SAFURE

D4.3
Final OS & RTE prototypes

Project number:	644080
Project acronym:	**SAFURE**
Project title:	SAFety and secURity by dEsign for interconnected mixed-critical cyber-physical systems
Project Start Date:	1st February, 2015
Duration:	40 months
Programme:	H2020-ICT-2014-1
Deliverable Type:	Demonstrator
Reference Number:	ICT-644080-D4.3 / 1.0
Work Package:	WP 4
Due Date:	M38 2018 - March
Actual Submission Date:	16th April, 2018
Responsible Organisation:	SYSGO AG
Editor:	Mikalai Krasikau
Dissemination Level:	PU
Revision:	1.0
Abstract:	This deliverable has presented the final implementation work in WP4.
Keywords:	mixed-criticality, OS, hypervisor, ARM, AURIX, RTEs, timing and temperature covert channels

This project has received funding from the European Unions Horizon 2020 research and innovation programme under grant agreement No 644080.

This work is supported (also) by the Swiss State Secretariat for Education, Research and Innovation (SERI) under contract number 15.0025. The opinions expressed and arguments employed herein do not necessarily reflect the official views of the Swiss Government.

Editor

Mikalai Krasikau(SYSGO AG)

Contributors (ordered according to beneficiary numbers)

Sylvain Girbal (TRT)
Jaume Abella, Francisco J. Cazorla, Enrico Mezzetti (BSC)
André Osterhues, Cheng Lu (ESCR)
Mikalai Krasikau, Sergey Tverdyshev (SYSGO)
Rehan Ahmed (ETHZ)
Mauro Marinonim, Marco Di Natale (SSSA)
Stefania Botta, Leonardo Giancippoli (MAG)

Disclaimer

Executive Summary

This deliverable presents a set of very diverse topics for embedded systems development. The driving idea for this work was to define a toolbox where each tool addresses an aspect for mixed-critical system design. The following topics were successfully investigated:

- Challenges in the usage of modern commercial of the shelf SoC for embedded system design
- Safety oriented scheduling (EDF) on hypervisor level
- Cryptography and usage of cryptography for mixed-critical systems
- Principles, benchmarks, and implementation of run-time engines for mixed-critical systems under time, space and energy constrains
- Controlling interference among mixed-critical applications
- Mixed-critical support for automotive systems based on AUTOSAR

Contents

List of Figures

List of Tables

Chapter 1 Introduction

This deliverable outlines the final implementation work in WP4. The work is based on the intermediate achievements reported in D4.1. The report is structured as follows. In Chapter 3 we describe the challenges project has encountered during the work on the modern COTS SoC as well as the analysis and the solutions we have developed. In Chapter 3 we present the porting of the PikeOS as well as extensions we have implemented in PikeOS. We demonstrate our approach for the platform security in the Chapter 4. Chapter 5 presents mixed-critical RTE running on PikeOS and the project ARM platform as well as the project results for mixed-critical approach on the automotive hardware. Chapter 6 and chapter 7 describe how timing dimension in the mixed-critical system has been addressed. We present research on mixed-criticality in the automotive context in Chapter 8. Finally, we conclude the document in the chapter 9.

Chapter 2 Platform Selection

2.1 Context of the Problem

A number of technologies developed as part of WP4 were intended to be integrated with the Telecom Use Case (WP6) for their evaluation in a real scenario. This imposed choosing a common hardware platform where to integrate the different technologies and the use case. The variety of constraints of the different technologies and software pieces led to a challenging decision process since a platform was needed including (at least) the following features:

1. Virtualization capabilities.
2. Being multicore (with at least four cores).
3. Performance Monitoring Unit (PMU).
4. Temperature sensors on each core.
5. Security-related hardware modules.
6. Having an industrial exploitation path (thus not purely being a development board).
7. Being affordable in price.

A number of candidate boards were identified and finally, the DragonBoard 810 (which includes the SnapDragon 810 processor) was initially selected due to the following reasons, related to the constraints above:

1. It included an ARM-based processor similar (a priori) to the one in the ARM Juno board, which was known to have the virtualization capabilities needed for the integration of PikeOS by SYSGO.
2. It included two clusters with four cores each.
3. It included a PMU partly documented in the public documentation from ARM.
4. It included temperature sensors for each of the 8 cores.
5. It included the minimum set of security-related modules.
6. It was not a development board but, instead, a board including features from industrial products.
7. Its price was within the range budgeted by the different partners.
8. The main disadvantage was a risk of accessing some advanced features (e.g. details on TrustZone implementation for HW virtualization) of the SoC, since they are seldom used in public projects and documented in public documentation. The consortium assessed this risk as reasonable and added and maintained this risk to the project risk table.

Unfortunately, as announced during the project review held in Brussels the 21st of November, a number of issues appeared during the integration of the different technologies in this platform. These issues relate mostly to the same problem: documentation available is scarce and, to some extent, erroneous.

This lack of complete and accurate documentation translated into the following problems, which we detail next:

- Information about the virtualization features is not available and they work differently than those in the ARM Juno board. Despite the many attempts performed by SYSGO to integrate PikeOS on this board, no success has been achieved since they have had to work blindly: to start the hypervisor one needs to install own software in the TrustZone or use the proprietary Qualcomm API, i.e. not ARM API, and this API is not public. The Qualcomm representative in the Advisory Board helped to setup contacts with the development department in San Diego. After that, we have contacted further labs such as the Android BSP development department and the CPU development department. However, no information could be obtained. Thus, the only option would be guessing how the hardware works and it is simply impossible configuring hardware features if the specific registers that need to be configured are not documented so that their addresses, contents and capabilities are unknown.
- Information about the PMU as well as other hardware details are incomplete and often erroneous. This processor has two clusters of four cores each, where each cluster shares a local L2 cache. Those L2 caches are connected to an interface where two memory controllers manage DRAM memory accesses. The PMU only provides information about cores, core-local caches and (limited) information about shared L2 caches. Moreover, some L2-related counters do not provide the information expected, thus reflecting a mismatch between the actual hardware and the documentation. Information about memory controllers is unavailable. Also, this processor has a pre-fetcher. The documentation describes how to disable it, so that this uncontrolled source of noise can be stopped. However, when trying to disable it as described by the documentation, the processor crashes. Again, since the only information available belongs to ARM, but the processor has been fabricated by Qualcomm and many elements are implementation-dependent, it seems that some default components have been modified. Therefore, the documentation available is limited and erroneous to some extent. This makes impossible characterizing the hardware platform and building on-chip contention models by BSC and TRT.
- This platform was planned as a springboard for the Telecom use-case because the selected phones were based on the same SoC. Thus, the plan to use the board as the springboard could not be implemented.

All in all, although the DragonBoard seemed to be the platform fitting everybodys needs, it failed to fulfil the requirements for its use in SAFURE. In the light of this, a number of actions have been taken as described in next section.

2.2 Undertaken Mitigation Actions

Initially, all these issues did not imply that the hardware platform did not meet the requirements. It could simply be the case that documentation, accurately describing the platform, is not publically available. Therefore, we first asked the Qualcomm representatives to grant us access to the documentation of the platform. For that purpose we tried several channels, including asking the Qualcomm representative in the Advisory Board (AB), Mr. Roberto Avanzi during our meeting in May 2016. Eventually, Mr. Roberto Avanzi put us in contact with appropriate Qualcomm representatives in San Diego, US, to follow the procedures needed to get access to this information. However, after a number of (very slow) iterations with Qualcomm representatives, we neither got the documentation required nor a commitment to get documentation in the future. Therefore, in November, we informed the Project Officer and the Reviewers about this situation during the Review Meeting, to make them aware of the situation and inform them that we were considering mitigation actions. Finally, partners agreed on looking for alternative hardware platforms despite the wasted time and resources on the DragonBoard, since simply waiting and polling Qualcomm was not providing any advance (nor a promise of future progress). The analysis, mostly led by SYSGO, brought on to the table a number of candidate platforms, which the partners assessed against their needs and constraints. A summarized result of this analysis is shown in the table below. This analysis required purchasing some additional boards to speed up technical assessments and skip whitepaper assessments that could result in erroneous assessment and introduce further delays.

SoC	**Docum -entation**	**HW access**	**PikeOS virt support**	**PMU**	**ETHZ (thermal protection)**	**TRT WP4 work (RTE + HWC)**	**BSC WP4 work**	**TRT WP6 work (RTE Integr.)**	**BSC WP6 work**	**TCS WP6 work**
Snapdragon 410 (Quad-core ARM Cortex A53)	no	locked	para_virt (hw_virt is blocked by vendor)	arm instructions (no registers inf)	Sensors available	very limited on HWC. Not possible RTE (missing PikeOS hw_virt support)	very limited	n.a.	n.a.	very limited
Snapdragon 810 (Octa core ARM Cortex quad A57, quad A53)	no	locked	para_virt (hw_virt is blocked by vendor)	arm instructions (no registers inf)	Sensors available	very limited on HWC. Not possible RTE (missing PikeOS hw_virt support)	very limited	n.a.	n.a.	full access (some compromises with performance expected)
juno SoC (ARM vendor) (ARM Cortex-A72 or A57 and Cortex-A53)	full	full	hw_virt ARMv8	Full up to the L2 cache. Nothing for memory controller	Sensors available. Technically doable. Budget not available for getting this board.	Current board used to develop the RTE. Some info missing about SoC HWC.	Limited but technically doable. Effort/time remaining largely insufficient	OK. Match WP4 current board for RTE.	To be seen if extrapolation from WP4 is possible	n.a.
HiSilicon Kirin 620 ARM Cortex-A53	poor (in chinese)	full	ARMv8	arm instructions (no registers inf)	Sensors available. Technically doable given PikeOS support	Can only start once PikeOS is available on the platform. Any SoC or HWC info?	At best very limited	Extrapolation possible for budget values. But not for porting the RTE. Requires PikeOS driver and native pers. Support.	To be seen if extrapolation from WP4 is possible	n.a.
Mediatek X20 (2x ARM Cortex-A72 4x Cortex-A53 4x Cortex-A53)	no	full (not tested)	ARMv8 hw_virt expected (not yet tested, missing firmware doc from vednor)	not yet assessed	Same as HiKey	Same issue as HiKey.	At best as for Juno	Same as HiKey	To be seen if extrapolation from WP4 is possible	n.a.
Infineon TC27x	full	full	n.a.	full	n.a. This board includes low power microcontrollers where thermal protection scheme is likely not relevant.	n.a.	Technically doable. Effort/time remaining may be insufficient	Same as HiKey	Should be done for MAG use case, not TCS. Still to be assessed with MAG	n.a.

Table 2.1: Boards Comparison Table

As a summary, we concluded the following about each one of the platforms above:

- All SnapDragon-based boards (e.g. DragonBoard) suffer from the same problem related to the lack of detailed (and correct) documentation available. Since there is no reason to expect that this information is made available to the consortium anytime soon, we can only dismiss these boards.
- The ARM Juno board allows porting PikeOS, as needed by SYSGO, since virtualization features are in place. TRT also bought this board and is now familiar with it. BSC does not own this board and lacks the time and resources to start from scratch with it. On the other hand, PMU documentation, while being more precise than that for SnapDragon-based boards, is still incomplete. Finally, since it is a development board, it is not suitable for TCS, so the Telecom use case cannot be ported on top of it. Due to the cost of this board, ETHZ does not have budget for it.
- HiSilicon and Mediatek boards allow accessing all hardware features needed. However, either they lack of publicly available documentation or it is in Chinese. Requests for getting detailed documentation from the chip vendors have not been answered satisfactorily so far (e.g. some of them need a bootloader, which will be developed and documented by the open-source community in the future and the vendor cannot provide the release date), and there is no reason to expect anything better than in the case of the SnapDragon-based boards.
- The Infineon AURIX TC27x board is the one to be used in the automotive use case by MAG. BSC technology could be ported to this board, but porting PikeOS or the Telecom use case on top of it is not doable since its architecture is too fine tuned for automotive and does not provide support (e.g. MMU, enough RAM) for many of the features in PikeOS and the Telecom use case.

As shown, no single board satisfies all partners needs. While this was anticipated by TCS earlier in the project, the first ambitious attempt was to integrate all technologies in the lowest number of different platforms. It was clear that the automotive use cases and the Telecom one could not be built on top of the same hardware platform due to their (too) different nature, however the target was to use a single platform for the telecom use case and a large fraction of the WP4 technologies. However, as shown before, this is not possible.

2.3 Final Decision

Since we did not have a single platform that satisfies requirements of all partners, a number of additional mitigation actions had been done. The consortium decided to use three different boards to cover all technologies planned for WP4 as well as make an overall WP4 results assessment in the SAFURE Framework in WP6. The following boards have been used in the WP4 implementation tasks:

- **ARM Juno board:** this board has been used to integrate results from SYSGO, ESCRYPT, TRT RTE scheduling and the WP4 mixed critical prototype (BSC, EZH).
- **Sony Xperia:** this platform, by being industrial, has been selected by TCS for the use-case in WP6. The porting of WP4 results (e.g. hypervisor and BSPs) will be described in WP6 to keep it in one context.
- **Infineon AURIX TC27x:** this board has been used to implement the automotive part of the WP4.

Chapter 3 PikeOS Support

As it has been described in SAFURE-D4.2-delay-justification-M24, it is hard to get to hypervisor mode on the DragonBoard 810 platform due to closed documentation and poor communication from Qualcomm side. In this delivery we introduce the results of porting PikeOS on the Juno board.

Since intermediate results of this work have already been presented in Deliverable D4.1, for the sake of completeness the current chapter will contain an updated content from Deliverable D4.1.

3.1 Juno Board

Within WP4, SYSGO provides support for PikeOS on the ARMv8 architecture. We have structured the porting work in two phases. The first phase, early development, has been done on Fixed Virtual Platforms simulator provided by ARM. The second phase has been done on the ARM Juno board. The ARM Juno board has been chosen as a target for the porting considering the following arguments:

- It is the only fully open development ARMv8 board with industrial support available on the market (at the time of making the decision on the board)
- It provides required debug facilities (JTAG and trace, serial port) for developing an operating system and hypervisor
- The support is provided directly by ARM Company.

The characteristics of the board are:

- Compute Subsystem
  - Dual Cluster, big.LITTLE, big.LITT configuration
  - Cortex-A57 MP2 cluster (r0p0), Overdrive 1.1GHz operating speed, Caches: L1 48KB I, 32KB D, L2 2MB
  - Cortex-A53MP4 cluster (r0p0), Overdrive 850MHz operating speed, Caches: L1 32KB, L2 1MB
  - Caches: L2 128KB
  - Quad Core MALI T624 r1p0, Nominal 600MHz operating speed
  - CoreSight ETM/CTI per core
  - DVFS and power gating via SCP, 4 energy meters
  - DMC-400, dual channel DDR3L interface, 8GB 1600MHz DDR
  - Internal CCI-400, 128-bit,533MHz
- Rest of SoC
  - Internal NIC-400, 64-bit, 400MHz
  - External AXI ports: using Thin-Links

  - DMAC : PL330, 128-bit
  - Static Memory Bus Interface : PL354
  - 32bit 50MHz to slow speed peripheral
  - HDCLCD dual video controllers: 1080p

- Debug
  - ARM JTAG: 20-way DIL box header
  - ARM 32/16 bit parallel trace

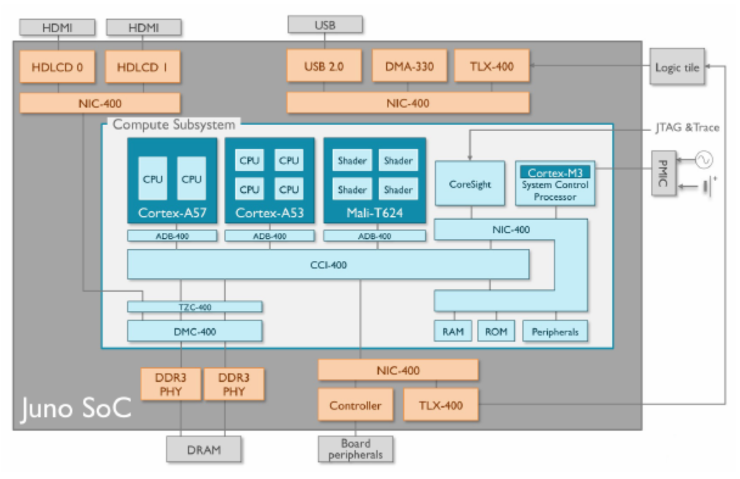


Figure 3.1: Juno SoC architecture

3.1.1 Porting Results for the Fixed Virtual Platforms simulator

As described before, the first porting has been done with using ARM Fixed Virtual Platform simulator for A53-A57 cores. The software models provide Programmers View (PV) models of processors and devices. The functional behavior of a model is equivalent to real hardware. PV models sacrifice absolute timing accuracy to achieve fast simulated execution speed. This means that you can use the PV models for confirming software functionality, but you must not rely on the accuracy of cycle counts, low-level component interactions, or other hardware-specific behavior [7].

The following features were developed on this simulator:

- **Hardware Virtualization**
  Hardware virtualization or *platform virtualization* refers to the creation of a virtual machine that acts like a real computer with an operating system. Software executed on these virtual machines is separated from the underlying hardware resources [36]. It allows PikeOS to run different operating systems on the same hardware simultaneously with just a little (or even without) modification of the operating system. It also provides almost no overhead in comparison to software virtualization, so performance of the operating system running under hypervisor supporting hardware virtualization is comparative to the performance of the same native operating system.

- **Trust Zone**
  TrustZone technology is programmed into the hardware, enabling the protection of memory and peripherals. Since security is designed into the hardware, TrustZone avoids security vulnerabilities caused by proprietary, non-portable solutions outside the core. Security can be maintained

as an inherent feature of the device, without degrading system performance, enabling device manufacturers to build security applications, such as DRM or mobile payment as protected applications that run on the secure kernel [1]. PikeOS supports this technology for ARMv7 architecture and now it has been ported to ARMv8.

3.1.2 Porting Results Juno

Currently the Juno board is supported officially in PikeOS 4.2 with the following features and interfaces:

- Hardware Virtualization
- TrustZone
- Serial driver
  Serial driver provides console support and allows to establish basic communication between target and host. Usually it is used for debugging and administration purposes.
- Ethernet driver
  Ethernet driver provides network support.
- ElinOS BSP
  Board Support Packages (BSPs) contain the necessary necessary adaptations to be able to run a Linux kernel on a specific target platform. ElinOS kernels contain some modifications to ensure a smooth operation with the ElinOS tool.

3.1.3 Testing Juno

Juno board support has been tested with the PikeOS Generic BSP Test Suite. This test suite is used for a regression testing as well as for a product testing. The results are listed in Table 3.1.

Test	Start	End	Operation	Result
arm-config-test	10:04:51	10:05:39	Run	OK
basic-linux-test	14:18:37	14:23:52	Run	OK
coherency-test	10:14:02	10:14:52	Run	OK
context-switch-bench	10:12:38	10:14:01	Run	OK
decode-test	10:16:17	10:17:03	Run	OK
demo-linux-guest-test	10:14:52	10:16:17	Run	OK
directio-test	10:18:30	10:21:22	Run	OK
fpu-test	10:21:22	10:22:41	Run	OK
hello-world	10:22:42	10:23:46	Run	OK
hwvirt-debug	10:08:04	10:09:15	Run	OK
interrupt-forward-test	10:37:15	10:37:59	Run	OK
memory-test	10:34:14	10:35:05	Run	OK
network-bench	10:35:05	10:37:14	Run	OK
p4bus-bench	10:40:37	10:44:06	Run	OK
p4bus-console-test	14:27:37	14:28:33	Run	OK
p4bus-enumerate-test	10:45:05	10:46:39	Run	OK
p4bus-mmaplseek-specvmem-test	10:59:49	11:02:03	Run	OK
p4bus-mmaplseek-test	11:04:33	11:06:55	Run	OK
p4bus-multiApp-vmchar-test	11:07:11	11:08:08	Run	OK
tiFD-vmchar-test	11:10:04	11:11:46	Run	OK

Continued on next page

Table 3.1 – *Continued from previous page*

Test	Start	End	Operation	Result
p4bus-muxa-test	11:13:04	11:15:38	Run	OK
p4bus-network-test	11:19:10	11:23:15	Run	OK
p4bus-nonblock-vmchar-test	18:29:40	18:31:04	Run	OK
p4bus-severalMem-test	11:26:04	11:27:55	Run	OK
p4bus-sigkill-vmchar-test	11:28:57	11:34:54	Run	OK
p4bus-test	11:34:55	11:37:47	Run	OK
p4bus-vmapi-sysfs-test	11:37:47	11:38:55	Run	OK
p4bus-vmapi-test	11:38:56	11:40:05	Run	OK
p4bus-vmblock-test	11:41:18	11:44:56	Run	OK
p4bus-vmchar-fp-test	11:44:56	11:46:15	Run	OK
p4guest-console-test	11:47:01	11:47:58	Run	OK
p4guest-multiApp-test	11:19:06	11:19:57	Run	OK
p4guest-vmfp-test	11:18:18	11:19:05	Run	OK
preempt-test	11:48:46	11:52:47	Run	OK
stress-test	11:53:33	11:54:46	Run	OK
stress-test-smp	11:54:47	11:56:21	Run	OK
trace-perfomance-test	11:56:22	12:03:42	Run	OK
virtio-test	11:52:47	11:53:32	Run	OK

Table 3.1: Benchmarks

3.1.4 Conclusion

Within the SAFURE project SYSGO has provided full support for a new ARMv8 64bit architecture in PikeOS hypervisor. The Juno board which is the official ARM reference platform for ARMv8 architecture is now officially supported by SYSGO's PikeOS hypervisor. By providing support for state-of-the-art technologies, SYSGO supports innovations and improves its stability on the European market.

3.2 Fixed-priority priority and EDF for mixed mixed-critical critical VM/-task scheduler

3.2.1 PikeOS Scheduling

PikeOS is a virtualising embedded real-time operating system. Its basic mechanism for ensuring isolation of different applications is partitioning, which enforces isolation both spatially and temporally. This way, mixed-criticality systems can be constructed based on the guarantees the PikeOS partitioning provides.
Partitioning in PikeOS is two-fold: to partition CPU time, an ARINC653-based time partitioning is available, while other resources like memory, I/O access, and communication rights management are handled by resource partitioning. Here, we will concentrate on the scheduling functionality provided by PikeOS. The top-level system scheduling in PikeOS is time-partitioning, which uses a static round-robin fixed sequence schedule to determine which partitions are scheduled at which point in time. For some kind of dynamics to react to different system states or situations, PikeOS supports different time partition schedules that can be switched by a system partition, but there is no way to construct new time partition schemata at run-time. This static configuration is done intentionally to ease argumentation about guarantees needed for certification of the temporal isolation and timing properties of the system.

The time partition scheduling in PikeOS has an extension over the ARINC653 partitioning: a time partition that is always active, which is called tp0. At any time, threads from the current time partition

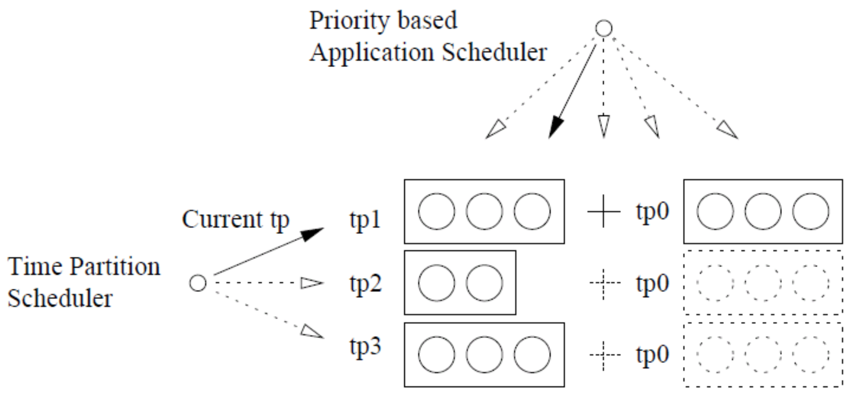


Figure 3.2: PikeOS time partitioning

plus those from TP0 (shown as tp0 in Figure 3.3) are eligible to thread scheduling. Figure 3.2 shows how PikeOS time partitioning works: the time partition scheduler selects, based on the static round robin schedule, one of the time partitions, and additional to that, TP0 is also active. From the set of threads selected this way, the second level scheduler, which is priority based, selects the thread with the highest priority.

TP0 is an important concept in PikeOS, because it can be used for two major tasks:

- To run background tasks at low priority, i.e., a Linux partition that should be active only whenever the system has nothing else to do. This way, free CPU resources can be utilised without allocating explicit CPU time. This possibility is best-suited for non-time-critical tasks. Figure 3.3 shows this possibility.
- To run high-priority error handlers that are mostly inactive except in critical situations. E.g., a power failure handler could be allocated to tp0 with high priority, so that it can react quickly if necessary. Again, no CPU time would have to be pre-allocated to such tasks, because in the normal case, the error handler is expected not to run, and in the exceptional case, it can still react with minimal delay. Because such error handlers have highest priority, they are automatically in the highest criticality class of the system, because of TP0, they are always active and could disrupt the whole system in case they went out of control. Figure 3.4 shows this use case of TP0 in PikeOS.

3.2.2 Event Handling in PikeOS

Earliest deadline first (EDF) as described in Section 3.2.3 is most attractive for sporadic, event-driven tasks, and for background activity that is not particularly time-critical, but for which starvation must be ruled out. It makes less sense for purely periodic activities, as such activities can be budgeted for in the time partition table. Therefore, EDF is most attractive for latency-sensitive threads in TP0 that should be able to preempt threads in other time partitions.

Application threads are routinely assigned to TP0 to serve as low-latency event handlers (e.g., interrupt threads). Such threads are typically lightweight, but there is no enforcement mechanism. In the case of an interrupt storm, high-priority event handlers assigned to TP0 could starve other time partitions (in the current system).

By coupling EDF with mandatory budget enforcement, it is possible to assign threads to TP0 that previously would have been too dangerous to allow the ability to run at any time: starvation of threads

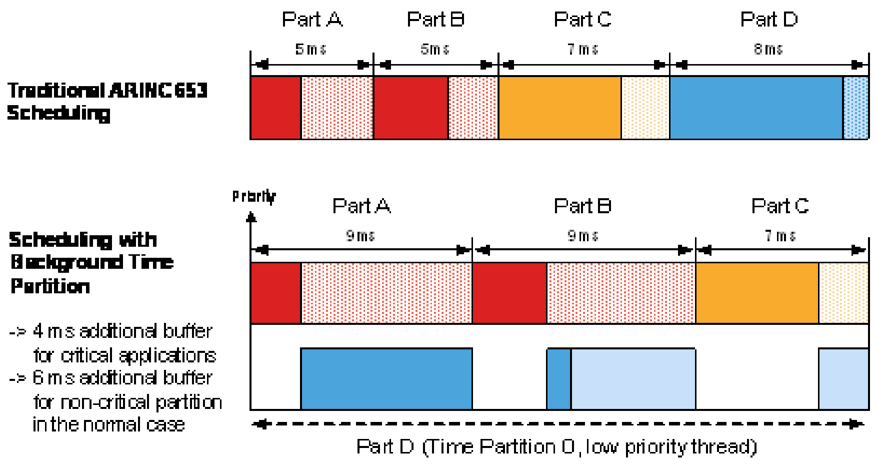

Part A
Part B
Part C
Part D
5 ms
5 ms
7 ms
8 ms
Traditional ARINC653 Scheduling
Priority
Part A
Part B
Part C
9 ms
9 ms
7 ms
Scheduling with Background Time Partition
-> 4 ms additional buffer for critical applications
-> 6 ms additional buffer for non-critical partition in the normal case
Part D (Time Partition 0, low priority thread)

Figure 3.3: Background task in tp0, as an extension to traditional ARINC653 time partitioning.

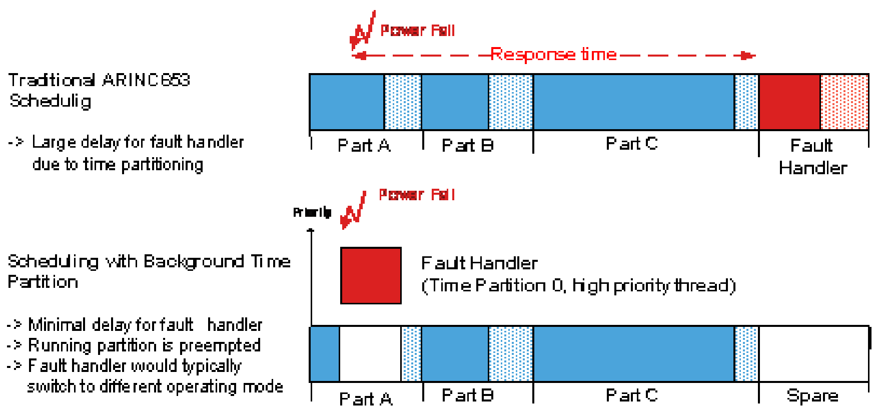

Power Fail
Response time
Traditional ARINC653 Schedulig
-> Large delay for fault handler due to time partitioning
Part A
Part B
Part C
Fault Handler
Power Fail
Priority
Scheduling with Background Time Partition
-> Minimal delay for fault handler
-> Running partition is preempted
-> Fault handler would typically switch to different operating mode
Fault Handler (Time Partition 0, high priority thread)
Part A
Part B
Part C
Spare

Figure 3.4: Fault handler in tp0, another extension to ARINC653 time partitioning.

in other time partitions can be prevented. In particular, this makes sense if the replenishment period of an event-driven thread in TP0 is shorter than the partition cycle. If the budget of an event-driven thread is larger than a time partitions slice, starvation is still possible. By making the replenishment period (substantially) shorter than the length of a time partition, it is ensured that no partition is starved in its entirety.

3.2.3 Earliest Deadline First (EDF)

Earliest deadline first (EDF) is a dynamic scheduling algorithm used in real-time operating systems to place processes in a priority queue. Whenever a scheduling event occurs (task finishes, new task released, etc.) the queue will be searched for the process closest to its deadline. This process is the next to be scheduled for execution [35].

3.2.4 EDF Reservations

Our overall solution involves introducing EDF-scheduled reservations, analytically sound containers for threads, at specific priority levels, called EDF bands, within TP0. The advantage of this approach is that from the perspective of the fixed-priority scheduler, we are simply deferring the queuing logic for certain priority levels to a secondary EDF scheduler, instead of following the default FIFO scheme.

Thus customers can simply choose to not designate any priority levels as EDF bands, or choose not to assign tasks to an EDF priority level. Also, EDF bands and all reservations can be configured statically what is important in certified safety-critical systems. Since reservation-based scheduling is a well-studied problem, it opens the possibility to analytically determine the temporal correctness of the system. Moreover, it has a low barrier to entry since the placement of certain threads in reservations is transparent to application developers (i.e., no special support needs to be implemented by the tasks themselves), and system integrators can rely on automatic analysis support in the integration toolchain courtesy to the large body [26, 34, 33, 2] of supporting theory for reservation-based scheduling. The integration of EDF bands within the existing fixed-priority scheduler ensures that zero low-latency low-criticality tasks interference can be trivially guaranteed to tasks that run at priorities higher than any EDF band.

3.2.5 Implementation

In PikeOS 4.0, SYSGO introduced pluggable kernel drivers. The design is such that at configuration time, the final PikeOS kernel binary is linked together from the core kernel binary, plus the platform support package (PSP), containing lowest level timer and boot support, plus user-defined kernel driver modules. This mechanism is available to customers, so no special support needs to be given when developing a kernel driver.

In the following paragraphs, we provide a brief overview of the implementation.

Plugin Framework

A scheduler plugin framework allows to easily implement extensions to the core scheduler. There are two key advantages to this approach. First, by providing such a framework with a fixed interface, once it has been certified, the only certification that needs to be performed is that of the plugins using that interface, and not the entire scheduler itself. Second, it allows for an efficient implementation; callbacks are bound only once (during initialization), and following this, the only added overhead is that of few additional conditionals.

The scheduler plugin framework was implemented via the poke interface in PikeOS, a mechanism to allow regular kernel drivers to provide, apart from the regular I/O functionality, any additional special functionality to the whole kernel binary (e.g., trace points, monitoring, and security loggers). This mechanism was used to implement a series of scheduler callbacks in a specially prepared PikeOS hypervisor kernel, allowing a scheduler plugin to insert its own logic into the control path of the main fixed-priority scheduler. The advantage of using the poke mechanism to implement this is that if no scheduler plugin is linked into the kernel, the PikeOS scheduler works with its default implementation with negligible overhead (as the poke mechanism has been optimized to be extremely lightweight when disabled). To support future extensions to the scheduler, the scheduler plugin framework exports generic a callback interface.

We now describe the callback API in detail. Table 3.2 summarizes the callback interface exported by the scheduler plugin framework built upon PikeOSs poke interface. These can be broadly classified into four categories: per-thread callbacks, timer-related callbacks, thread-admission and auxiliary callbacks, and finally, scheduling-related callbacks.

Scheduling-Related Callbacks
requeue()
get_next()

Timer-Related Callbacks
next_update()
timer_check()
timeout_expire()
Per-Thread Callbacks
wakeup()
wait()
stop()
Thread Admission & Auxiliary Callbacks
admit()
resolve()
is_owner()
prio_cmp()

Table 3.2: Scheduler plugin callback functions.

Scheduling-Related Callbacks: These callback functions pertain to core scheduling events, which need to be handled by a scheduler plugin.

Timer-Related Callbacks: These callbacks are used by the kernel to (i) determine the next timer expiration it should program in hardware, and (ii) to inform the plugin when a timer has expired.

Per-Thread Callbacks: These callbacks are used by the kernel to interact with a scheduler plugin to inform and be informed of thread state changes. Thus, each of the corresponding functions in the plugin framework invokes corresponding callbacks within the scheduler plugin that must return state changes (described below), which are used to correctly update the fixed-priority schedulers bitmap pertaining to EDF priority levels.

Thread Admission and Auxiliary Callbacks: These various callbacks are used to admit tasks into the plugin and acquire auxiliary information from the scheduling plugin. Specifically (i) determining EDF priority levels and (ii) comparing priorities managed by the PikeOS FP scheduler and the EDF priority levels managed by the scheduling plugin.

Reservation-Based EDF Plugin with Modular Reservation Policies

A reservation-based EDF plugin is implemented in PikeOS by hooking into the scheduler plugin framework described earlier. The scheduler itself was implemented using standard techniques, with the one key difference that, instead of scheduling threads, it schedules reservations.

The plugin maintains a global data structure that comprises the state of all reservations pertaining to it. The structure of this global data structure is shown below:

```
struct sched_plugin {
    unsigned int priority;
    struct list_head l_active;      /* Ordered by priority */
    struct list_head l_depleted;    /* Ordered by next_replenishment */
    struct list_head l_inactive;    /* No specific order */
    P4_time_t next_update_time;
    P4_time_t last_update_time;
    P4_time_t sched_begin_time;
};
```

The EDF plugin keeps track of all reservations at a particular priority level in one of three internal queues: (i) the queue Qactive keeps track of all reservations containing one or more ready threads (i.e., thread that are not blocked and currently runnable), (ii) the queue Qinactive holds all reservations that currently have no runnable threads within them, and finally (iii) Qdepleted stores all reservations that have one or more ready threads but have depleted their budget and are awaiting replenishment. At runtime, as events occur for threads associated with reservations managed by the plugin (e.g., a thread inside a reservation waking up or blocking), the state of reservation is changed, and the reservation is moved to the appropriate queue (e.g., if the sole thread within an active reservation blocks, the reservation is moved to the inactive queue). The EDF scheduler schedules reservations within Qactive based on their absolute deadline, which is updated by the reservation policy.

The plugin maintains three queues internally: *l_active* is a list of active reservations ordered by priority (this means ordering reservations by earliest-deadline-first). *l_depleted* is a list of reservations that have eligible threads within them but have currently depleted their budget and are awaiting replenishment (ordered by increasing replenishment time). Finally, *l_inactive* is a list of all inactive reservations in the system, that is, reservations that do not currently have any eligible threads (i.e., because all assigned threads are currently blocked or none have been assigned to these reservations). Currently, we assume only one such structure is present and the global priority field is initialized to SCHED_PLUGIN_PRIO.

Reservation logic is abstracted as a set of reservation-specific callbacks that define the semantics when certain events in the system occur.

```
struct reservation_ops {
    admit_t           admit;
    thread_wakeup_t   wakeup;
    thread_wait_t     wait;
    pick_t            pick;
    timeout_t         timeout;
    requeue_t         requeue;
    burn_t            burn;
};
```

A detailed description of each callback is provided in the inlined comments below:

pick_t: This callback selects a thread to be scheduled from an internally maintained ready-queue. For most reservation schemes that we consider, this is simply a round-robin scheme. (An exploration of different queueing schemes within reservations is left to future work.) The time slice variable in the sched info structure is used by the callback to return how long the thread should be run for. This is useful for setting up one-shot timers for budget enforcement. Note that returning a zero time slice means the thread should be scheduled indefinitely.

thread_wakeup_t: This callback is invoked when a new thread is (i) added to a reservation or (ii) an existing thread is woken up.

wait_t: This callback is invoked when a thread belonging to a reservation is (i) suspended due to I/O or (ii) terminated. Note that this callback is invoked when a new thread is stopped, as PikeOS behaves similarly in both cases.

requeue_t: This callback is invoked in order to requeue a runnable thread back into the queue of the reservation that it belongs to. (It is not invoked if the thread blocked and is no longer runnable.)

burn_t: This callback is invoked in order to update the reservations budget to reflect any execution or idling time.

admit_t: This callback is invoked in order to admit a thread into a reservation. This is only performed once during system initialization for statically configured systems.

Based on the callback function pointers just described, an individual reservation is defined as shown below.

```
struct reservation {
    /* Pointer to reservation-specific ops structure */
    struct reservation_ops *ops;
    unsigned long id;
    res_state_t state;
    P4_time_t budget; /* The configured (static) budget */
    P4_time_t period; /* The configured (static) "period" */
    P4_time_t deadline; /* The relative deadline */
    P4_time_t cur_budget; /* The current budget (at runtime) */
    P4_time_t cur_deadline; /* The current deadline */
    P4_time_t next_replenishment; /* Next replenishment time */
    /* Pointer to global plugin data */
    struct sched_plugin *plugin;
    void *plugin_data;
    /*
    * This structure may be part of two lists: (i) the list of
    * threads assigned to the reservation of this thread and (ii)
    * one of the three queues in the global scheduler plugin data.
    */
    adt_list_t threads; /* The list of threads in this reservation */
    adt_node_t list;
    /* For queueing in scheduler plugin */
};
```

Each reservation is specified by a structure containing, among other things, an ID, its static configuration parameters (*budget*, *period*, and (relative) *deadline*), as well as its runtime state (its current budget, current deadline and next replenishment time in the fields *cur_budget*, *cur_deadline*, and *next_replenishment*, respectively).

Reservation Policy: Deferrable Servers

A deferrable servers policy was implemented upon the reservation-based EDF scheduler.

Deferrable servers [26, 34] are a real-time server algorithm that provides an analytically sound encapsulating container around aperiodic tasks. Deferrable servers are periodically provided a specified budget. That budget is consumed whenever a task within the reservation executes, and no more execution can occur once the budget has been depleted. Replenishment of the budget occurs periodically based on a configured period. Under deferrable servers, the budget is replenished to the full configured amount, regardless of whether the budget was depleted before the end of a given period or not. Multiple tasks may be assigned to a deferrable server, and these are served via a round-robin policy. Figure 3.5 shows how EDF reservations can be sporadically executed within a budget.

The advantage of deferrable servers is that it is relatively simple to implement: it requires less runtime overhead compared to other types of servers (e.g., sporadic servers [33] incur memory overhead to keep track of task suspensions), and provides flexibility compared to the simpler polling server algorithm. In particular, polling servers deplete reservation budget regardless of whether a task is executing or not, while under deferrable servers, tasks may arrive at any time during the period, and as long as sufficient budget is available, may execute. On the other hand, if a task arrives during the period of a polling server after its budget has been depleted, it must wait until the next period to execute.

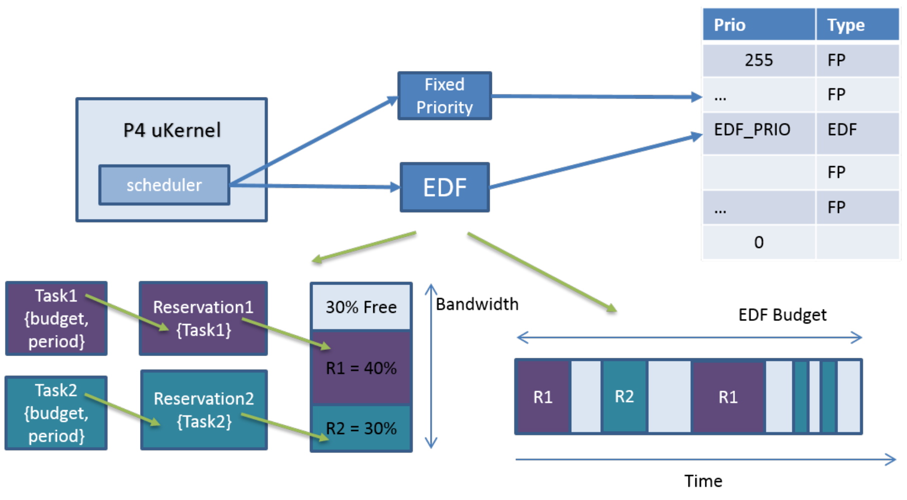


Figure 3.5: Reservation-Based EDF Plugin in PikeOS

Deferrable Servers: Implementation Under PikeOS

As can be seen below, we first define a set of reservation operations via a *reservation_ops* structure described above. We now detail the implementation of each of these functions for realizing a deferrable server policy under the reservation-based EDF scheduler.

```
struct reservation_ops ds_ops = {
     .admit =      ds_admit,
     .wakeup =     ds_thread_wakeup,
     .wait =       ds_thread_wait,
     .pick =       ds_pick,
     .timeout =    ds_timeout,
     .requeue =    ds_requeue,
     .burn =       ds_burn_budget,
};
```

ds_admit(): this callback is invoked by the reservation-based EDF scheduler whenever a new task is assigned to a deferrable-server-based reservation. In the deferrable servers implementation, no action is performed upong thread admission.

ds_thread_wakeup() and *ds_thread_wait*(): these callbacks are invoked by the EDF plugin whenever a threads state changes. *ds_thread_wakeup*() is invoked when a thread is woken up while blocking results in the *ds_thread_wait*() callback being invoked.

ds_burn_budget(): this callback is invoked whenever a thread has executed for a certain amount of time, and the budget of the reservation needs to be subtracted.

ds_requeue(): this callback is invoked whenever a runnable thread needs to be requeued before a scheduling phase begins.

ds_timeout(): Under deferrable servers, the only timers that are needed is a single replenishment timers for each reservation. *ds_timeout*() callback is invoked whenever a timer fires.

ds_pick(): Recall that when multiple threads are present within a deferrable server, they are serviced in a round-robin manner. *ds_pick*() gets the next thread to be executed and returns it to the EDF plugin.

Chapter 4 Security for Mixed-Criticality

4.1 CycurLIB on PikeOS

In order to meet the security requirements defined in the D1.2 (i.e., system integrity that both the operating system and the run-time environment are not manipulated), standard cryptographic algorithms (like AES, SHA-2, RSA, and ECC) are required. These cryptographic algorithms provide application programmers an easy and standard way to add security to applications, and SAFURE will integrate the most relevant algorithms into the run-time environment of PikeOS.

4.1.1 Cryptographic Algorithms

Security-critical applications typically need to protect the confidentiality (i.e., an attacker cannot read messages) and/or the integrity and authenticity (i.e., an attacker cannot modify messages) of data. To protect the confidentiality of messages, block ciphers are the most common choice. To protect the integrity and authenticity of data, either Message Authentication Codes (MACs) or digital signatures can be used.

4.1.1.1 Block Ciphers

Block ciphers like the Advanced Encryption Standard (AES) ensure the confidentiality of messages. If additionally also the integrity and authenticity need to be protected, an authenticated mode of operation like Galois/Counter Mode (GCM) can be deployed.

4.1.1.2 MACs

There are different ways to implement MACs, notably based on block ciphers (CMACs) and on hash functions (HMACs). Typically, HMACs offer a better performance (i.e., a higher data throughput) than block ciphers and are therefore often preferred. For older hash functions like SHA-1 and SHA-2, the HMAC construction, which requires two calls of the hash function, with an inner and an outer padding, has to be used. Alternatively, the KMAC algorithm, which is based on SHA-3 and requires no additional padding and just one loop, can be used.

4.1.1.3 Digital Signatures

Digital signatures are asymmetric algorithms based on hard mathematical problems like the integer factorization problem (RSA) or the discrete logarithm problem (DSA, ECDSA, EdDSA). While the keys for ”traditional” algorithms like RSA and DSA need to be quite large in order to be secure (at least 2048 bits), signature schemes based on Elliptic Curve Cryptography offer a similar security with much smaller keys (i.e., between 160 and 256 bits).

For instance, attacking 256-bit ECC with the currently best known methods requires about 2^{128} operations, which is infeasible even for large organisations with huge financial resources. Traditional asymmetric algorithms like RSA and DSA require about 3072-bit keys to offer the same level of security, because other attacks are possible.

4.1.1.4 CycurLIB

All above mentioned cryptographic algorithms are included in ESCRYPT's modular CycurLIB library, which is optimized for high performance for resource-constrained embedded devices. It can be optimized either for minimum code size (at the expense or slower execution speed) or for maximal execution speed (at the expense of a larger code size). Furthermore, the RAM usage is kept as small as possible. Also, due to its modular design, it can be configured to only contain the required algorithms, yielding an executable without any overhead.

4.1.2 Integration of Cryptographic Algorithms into the PikeOS Run-Time Environment

A Cryptographic File Provider for PikeOS has been created that includes the above mentioned cryptographic algorithms. This can be seen as a wrapper for the CycurLIB. From a developer's point of view, the cryptographic algorithms can be accessed with normal file operations.

PikeOS supports partitions with strong isolation. This means that security-critical applications can be separated from non-critical software. A traditional operating system (e.g. Linux) can run in a non-critical partition as a (para-)virtualized guest operating system. If an attacker manages to gain root access in the Linux system (e.g. through a vulnerability that enables privilege escalition), he still cannot get access to data in other security-critical partitions.

Thus, a secure system can be designed that includes the cryptographic algorithms (as a PikeOS File Provider) in one security-critical partition, a secure application in a second security-critical partition, and a non-critical partition for the virtualized guest operating system. The secure application contains all secret keys and uses the Cryptographic File Provider for all cryptographic operations like encryption, decryption, signature/MAC generation, and signature/MAC verification. This is similar to systems with hardware protection (e.g. Hardware Security Modules, HSMs), where all security-critical operations are carried out in a (hardware) module that is separated from the main processor. Hence, if this design pattern is followed, a protection similar to hardware security can be achieved against online attacks (i.e., assuming that the attacker has no physical access to the platform). However, HSMs offer additional protection against physical access attacks (e.g., temper-evidence or temper-resistance) that cannot be realized by a software-only solution.

Figure 4.1 shows the architecture of such a secure system. PikeOS provides the basic operating system functions (scheduling, resource management) and strong isolation of partitions. The cryptographic file provider based on CycurLIB could also be seen as part of the run-time environment. However, it is shown as a separate partition here to stress the strong isolation from the other partitions. Security-critical applications can be run in a separate partition. The idea is that all sensitive material (i.e., secret keys) is stored here and that this partition serves as an interface to the virtualized Linux system on one hand and the cryptographic file provider on the other hand. It is important to note that all operations on keys have to be run this partition. In addition to security-critical applications, safety-critical applications can also run in parallel, benefiting from strong isolation.

4.2 Secure Boot

A Secure Boot system is usually implemented using Message Authentication Codes (MACs): When flashing the firmware, the MAC authentication tag of the firmware is stored in a special hardware register or a dedicated memory region. The key used for MAC calculation also needs to be stored in a dedicated, secure memory region. Thus, the MAC key and the MAC authentication tag need to be stored in hardware-depend memory regions. Furthermore, the MAC verification routine has to be stored in a memory region that is protected against manipulation (otherwise, an attacker could simply modify the verification to always return ”passed”).

As it has been already mentioned in 2, it is hard to get access to all hardware components as well as to hypervisor mode on DragonBoard 810 platform due to closed documentation and poor commu-

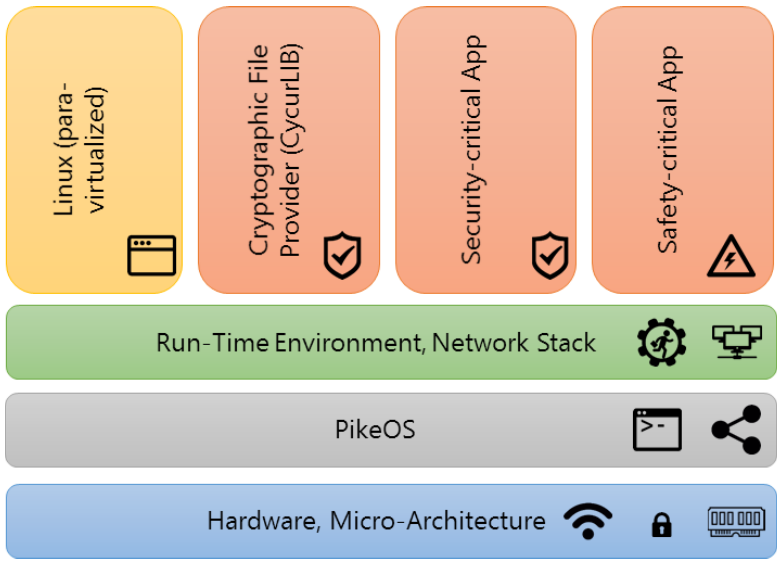

Linux (para-virtualized)
Cryptographic File Provider (CycurLIB)
Security-critical App
Safety-critical App
Run-Time Environment, Network Stack
PikeOS
Hardware, Micro-Architecture

Figure 4.1: PikeOS Run-Time Environment with Cryptographic File Provider

nication from Qualcomm side. Considering this, we decided to focus on Secure Update technology which requires only a cryptographic module or even a software library available on the target platform and no hardware-dependencies.

4.3 Secure Update

4.3.1 Architecture Overview

Currently, software updates for automotive ECUs are predominantly performed during service intervals within a service station. However, in order to react timely against emerging vulnerabilities and new threats, it is desirable to perform updates more frequently, ideally using a wireless connection to a remote server. To prevent attackers from modifying the firmware, security becomes a crucial part of the firmware update process.

The firmware update system consists of a Backend Server (operated by the OEM or a service provider) and the firmware update components within the embedded system. First, the firmware has to be transferred from the Backend Server to the embedded ECU. This can be done using a regular Internet connection (e.g., via GSM) to the Backend Server. In order to preserve the integrity and authenticity of the firmware, digital signatures or MACs can be used. Optionally, the firmware can also be encrypted in order to preserve the confidentiality and thus prevent the reverse-engineering of the software functions. Hence, the transferred data include the firmware itself (possibly encrypted) and a digital signature or MAC value which is attached to the end of the firmware.

Second, the firmware update is forwarded to a Secure Update application that (optionally) decrypts the firmware update. Then, the verification of the digital signature or the MAC authentication tag takes place. To accomplish this, the transferred digital signature is verified using standard cryptographic algorithms (like RSA, ECDSA, EdDSA) or in the case of MACs the MAC value of the message is calculated and checked against the transferred value. If both values are identical, the firmware update is considered authentic and the flashing of the firmware can be performed.

The architecture allows both asymmetric and symmetric algorithms. Depending on the platform (RAM size, ROM size, CPU speed), either the first or the latter is better suited. In the case of asymmetric cryptography, only the public key has to be stored in the embedded system. It is no problem if a potential attacker can read this public key. However, it has to be ensured that this key cannot be modified, because otherwise an attacker could replace it with his own public key. Similarly, the key used to generate the MAC value has to be securely stored and even protected from read-out by an unauthorized party. The rationale here is that an attacker could use the key to calculate a legitimate MAC value for a manipulated firmware update.

4.3.2 Implementation

The implementation of secure update described in the section above consists of remote server and a target platform. The server communicates with the target over Ethernet channel. The detailed architecture is depicted at Figure: 4.2.

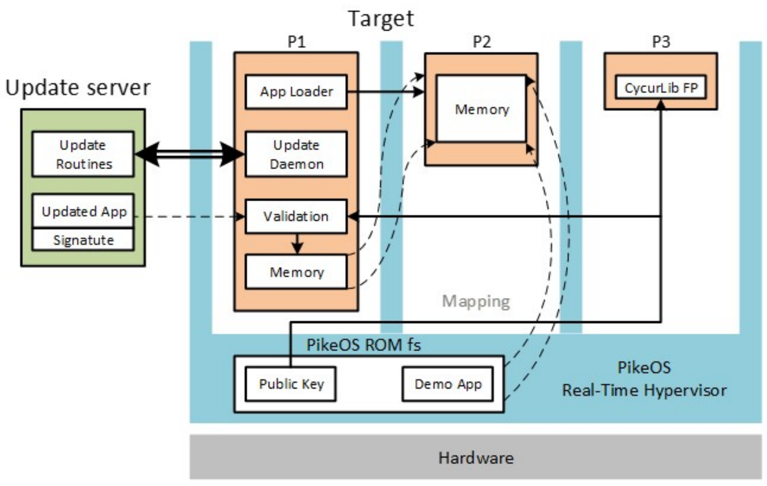


Figure 4.2: Secure Update Setup

An update package is stored on a server. The packet contains a binary application to be updated appended with its cryptographic signature. A signing process is shown at the picture 4.3. The signature is generated with a pair of public and private key using elliptic curve cryptography (ECC). The public key from the key pair then is stored on the target platform and used for a signature validation. The server communicates with a target and provides information about available updates.

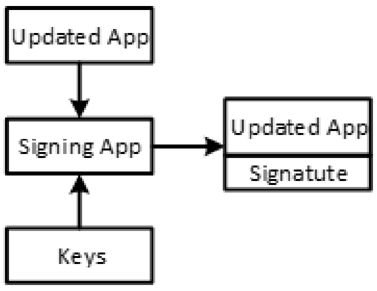


Figure 4.3: Generating an update packet

The target consists of three partitions:

- P1: Application Manager
  The *Application Manager* is responsible for requesting, receiving, validation and applying updates from the server. In turn it consists of three blocks:
  - Update daemon
    It receives an update request from the update server and downloads an update package.
  - Validation routines
    This block performs the validation of the received package. It loads the public key stored

in the file system. It sets up the CycurLIB file provider with the application, signature and the public key. After the preparation is finished, it requests the file provider to validate the signature.

  – Application Loader
    This block is responsible for the start and the update of the user application in P3. It has the ability to manipulate the execution state of P3 and also has writable access to the user application memory.

- P2: User application to be updated
  This partition is configured to run an application stored in RAM. This RAM is accessible by the Application Manager where Application loader.

- P3: CycurLIB file provider
  Please refer to section 4.1 for a detailed description.

The update workflow consists of the following steps:

- Initialization
- Getting an update package
- Validation
- Applying the update

At the *Initialization* step the P3 stays in idle mode. The Application Loader puts the initial user application stored in the ROM file system to the RAM and starts the partition.

After the initialization step is complete, the Application Loader starts the Update daemon to *get available updates*. It waits for an update request from the update server. When the new packed is available on the server the server notifies the Update Daemon with the request for update. The Update Daemon downloads the update package.

The *Validation* step is performed by the *Validation block* which receives the downloaded Update Package. The Validation block retrieves the pointer to the signature from the package and also loads the public key storing on the file system. It sets up the CycurLIB file provider with the application, signature and the public key. After the preparation is finished, it requests the file provider to validate the signature.

After the successful validation of a received update the Application Loader performs an *update procedure*. It stops P3 and replaces its RAM with the new application. Then it restarts P3. The new application starts its execution. In case of a failed validation result the update will not be applied and the Application Partition (P2) will not be affected.

4.3.3 The Secure Update Demonstrator

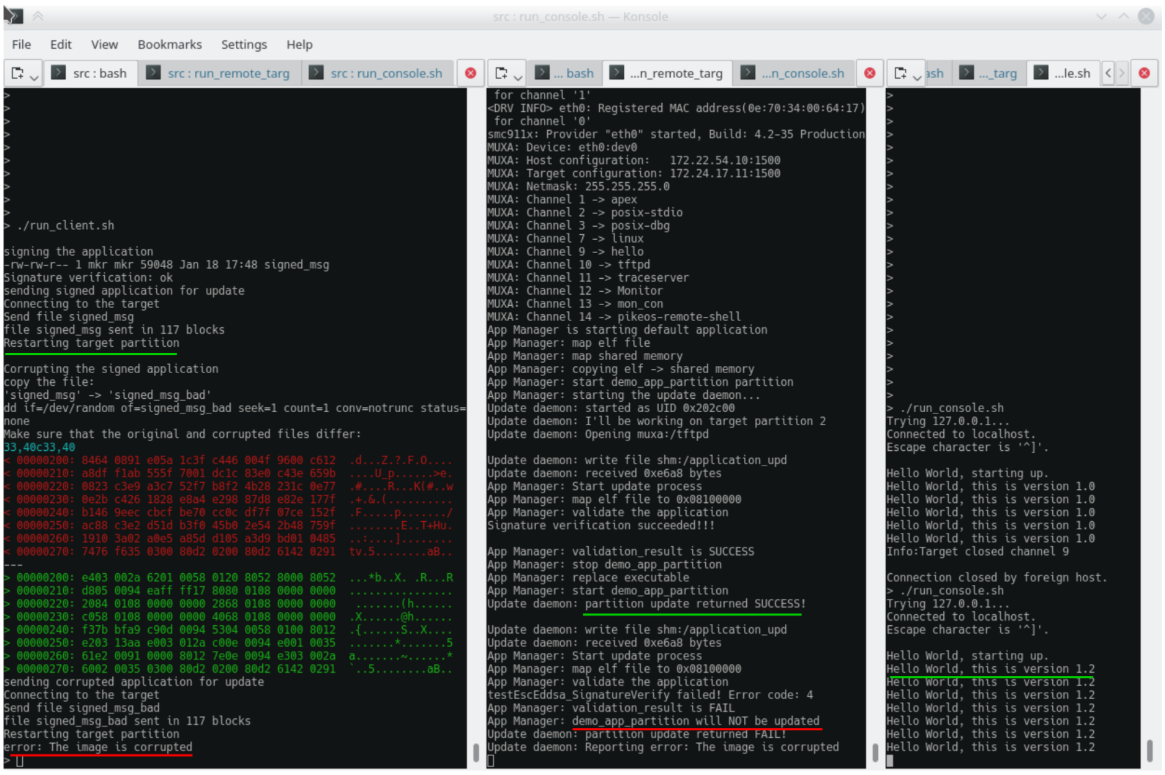
At the HiPEAC workshop in Manchester the Secure Update has been demonstrated. Figure 4.4 shows the demo resulting screen. There are three terminal there:

- The left one is a is the server output.
- The middle one is a Application Manager output.
- The right one is a User Application output.

The server requests an update process twice. The first Update package is correct and the second one is intendedly corrupted.

Green lines highlight result of the correct update packet update. The Application manager reports successful validation of the package and the printed version of the User Application partition is changed to 1.2.

Red lines highlight result of update with a corrupted package. The Application manager reports an validation error and returns this error to the server. The User Application partition continues its execution without a stopping or update.



```
>
>
>
>
>
>
>
>
>
>
> ./run_client.sh

signing the application
-rw-rw-r-- 1 mkr mkr 59048 Jan 18 17:48 signed_msg
Signature verification: ok
sending signed application for update
Connecting to the target
Send file signed_msg
file signed_msg sent in 117 blocks
Restarting target partition

Corrupting the signed application
copy the file:
'signed_msg' -> 'signed_msg_bad'
dd if=/dev/random of=signed_msg_bad seek=1 count=1 conv=notrunc status=
none
Make sure that the original and corrupted files differ:
33,40c33,40
< 00000200: 8464 0891 e05a 1c3f c446 004f 9600 c612  .d...Z.?.F.O....
< 00000210: a8df f1ab 555f 7001 dc1c 83e0 c43e 659b  ....U_p......>e.
< 00000220: 0823 c3e9 a3c7 52f7 b8f2 4b28 231c 0e77  .#....R...K(#..w
< 00000230: 0e2b c426 1828 e8a4 e298 87d8 e82e 177f  .+.&.(..........
< 00000240: b146 9eec cbcf be70 cc0c df7f 07ce 152f  .F.....p......./
< 00000250: ac88 c3e2 d51d b3f0 45b0 2e54 2b48 759f  ........E..T+Hu.
< 00000260: 1910 3a02 a0e5 a85d d105 a3d9 bd01 0485  ..:....].......
< 00000270: 7476 f635 0300 80d2 0200 80d2 6142 0291  tv.5........aB..
---
> 00000200: e403 002a 6201 0058 0120 8052 8000 8052  ...*b..X. .R...R
> 00000210: d805 0094 eaff ff17 8080 0108 0000 0000  ................
> 00000220: 2084 0108 0000 0000 2868 0108 0000 0000   .......(h......
> 00000230: c058 0108 0000 0000 4068 0108 0000 0000  .X......@h......
> 00000240: f37b bfa9 c90d 0094 5304 0058 0100 8012  .{......S..X....
> 00000250: e203 13aa e003 012a c00e 0094 e001 0035  .......*.......5
> 00000260: 61e2 0091 0000 8012 7e0e 0094 e303 002a  a.......~......*
> 00000270: 6002 0035 0300 80d2 0200 80d2 6142 0291  `..5........aB..
sending corrupted application for update
Connecting to the target
Send file signed_msg_bad
file signed_msg_bad sent in 117 blocks
Restarting target partition
error: The image is corrupted
> 
```

```
 for channel '1'
<DRV INFO> eth0: Registered MAC address(0e:70:34:00:64:17)
 for channel '0'
smc911x: Provider "eth0" started, Build: 4.2-35 Production
MUXA: Device: eth0:dev0
MUXA: Host configuration:   172.22.54.10:1500
MUXA: Target configuration: 172.24.17.11:1500
MUXA: Netmask: 255.255.255.0
MUXA: Channel 1 -> apex
MUXA: Channel 2 -> posix-stdio
MUXA: Channel 3 -> posix-dbg
MUXA: Channel 7 -> linux
MUXA: Channel 9 -> hello
MUXA: Channel 10 -> tftpd
MUXA: Channel 11 -> traceserver
MUXA: Channel 12 -> Monitor
MUXA: Channel 13 -> mon_con
MUXA: Channel 14 -> pikeos-remote-shell
App Manager is starting default application
App Manager: map elf file
App Manager: map shared memory
App Manager: copying elf -> shared memory
App Manager: start demo_app_partition partition
App Manager: starting the update daemon...
Update daemon: started as UID 0x202c00
Update daemon: I'll be working on target partition 2
Update daemon: Opening muxa:/tftpd

Update daemon: write file shm:/application_upd
Update daemon: received 0xe6a8 bytes
App Manager: Start update process
App Manager: map elf file to 0x08100000
App Manager: validate the application
Signature verification succeeded!!!

App Manager: validation_result is SUCCESS
App Manager: stop demo_app_partition
App Manager: replace executable
App Manager: start demo_app_partition
Update daemon: partition update returned SUCCESS!

Update daemon: write file shm:/application_upd
Update daemon: received 0xe6a8 bytes
App Manager: Start update process
App Manager: map elf file to 0x08100000
App Manager: validate the application
testEscEddsa_SignatureVerify failed! Error code: 4
App Manager: validation_result is FAIL
App Manager: demo_app_partition will NOT be updated
Update daemon: partition update returned FAIL!
Update daemon: Reporting error: The image is corrupted
```

```
>
>
>
>
>
>
>
>
>
>
>
>
>
>
>
>
>
>
>
>
>
>
>
>
> ./run_console.sh
Trying 127.0.0.1...
Connected to localhost.
Escape character is '^]'.

Hello World, starting up.
Hello World, this is version 1.0
Hello World, this is version 1.0
Hello World, this is version 1.0
Hello World, this is version 1.0
Hello World, this is version 1.0
Info:Target closed channel 9

Connection closed by foreign host.
> ./run_console.sh
Trying 127.0.0.1...
Connected to localhost.
Escape character is '^]'.

Hello World, starting up.
Hello World, this is version 1.2
Hello World, this is version 1.2
Hello World, this is version 1.2
Hello World, this is version 1.2
Hello World, this is version 1.2
Hello World, this is version 1.2
Hello World, this is version 1.2
```

Figure 4.4: Secure Update Result

Chapter 5 Mixed-Critical Run Time Engine

In this chapter we will present the Budget-Based RunTime Engine (BB-RTE), a mixed-critical runtime engine aiming at ensuring safe time-critical behavior of high-critical applications running concurrently with low-critical applications.

For the sake of completeness, Section 5.1 will recall the safety-critical and time-critical context and requirements already presented in Deliverables D1.2 and D3.2. The principles of the runtime engine are presented again in Section 5.2, while the associated process is now further detailed in Section 5.3. The experimental setup is described in Section 5.4 and relies on the WP4 prototype presented in Chapter 6. Finally, evaluation results can be found in Section 5.5.

Also in this chapter we introduce the thermal protection extension for the BB-RTE (Section 5.6), as well as the assessment of time-protection capabilities of the AURIX TC275 automotive platform (Section 5.7).

5.1 Context

In previous deliverables, we already pointed out that safety-critical and time-critical applications are characterized by stringent real-time constraints, making **time predictability** a major concern with regards to the regulation standards [20, 21, 30].

The recent shift of these industries toward multi-core COTS (component off-the-shelf) processors for size, weight and power (SWaP) [9] as well as efficiency reasons, introduced new sources of time variations, and the solution providers can no longer rely on resource over-provisioning to enforce time predictability. As a consequence, the industry is facing a trade-off between performance and predictability [23, 27].

As depicted in Figure 5.1, multi-core processors are characterized by shared hardware resources such as some levels of caches, the interconnect, the main memory or I/O controllers.

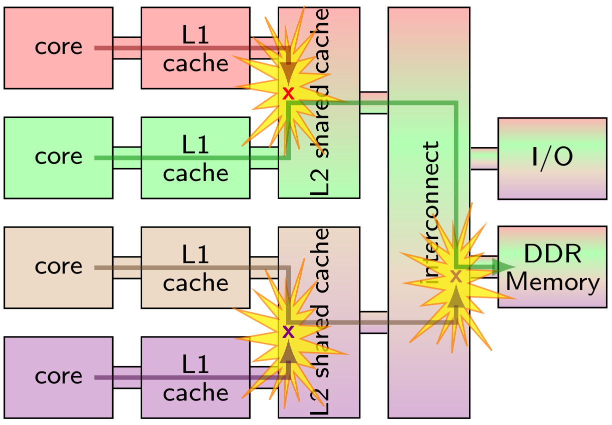


Figure 5.1: Timing Interference in multi-core architectures

At hardware level, concurrent accesses on these resources are arbitrated, introducing jitter at application level defined as **timing interference** [14]. Such interference, caused by electronic competition on shared hardware resources, are breaking the timing isolation principles required by the industry standards [20, 21, 30] of time-critical software.

Several papers [10, 28] have studied the impact of timing interference on the Worst-Case Execution Time (WCET). Their authors have shown that not trying to tackle the interference problem leads to a performance loss for worst-case that can go far beyond the expected performance gain of using a multi-core processor.

In a survey on **Deterministic Platform Solutions** [13] we presented various solutions allowing some level of time-deterministic usage of non-deterministic hardware platforms. Two families of solutions are presented: **control solutions** that aim at eliminating all interference by restricting the usage of the hardware platform, and **regulation solutions** that are focusing on keeping the impact of timing interference below a harmful level.

By introducing over-provisioning or complex runtimes, control solutions such as Marthy [22] or deterministic adaptive scheduling [12] fail to efficiently exploit the multi-core performance. Other solutions relying on execution models restrict too much the usage domain. For instance, The AER model [14] is easily applicable to distributed memory systems but lacks strong guarantees with regular shared memory systems.

Regulation solutions, on one hand, offer a better performance efficiency but lack the strong guarantees required by domains such as avionics. On the other hand, they are perfectly adapted to less critical environments such as mixed-critical systems where high-critical tasks are running conjointly with low-critical tasks. In such systems, regulation solutions offer the possibility to degrade low-critical tasks to guarantee the timing behavior of high-critical tasks. To provide time guarantees, regulation solutions usually rely on budgeting the time with early deadlines or shifting time windows [25].

Within the SAFURE project, we developed the **Budget-Based RunTime Engine**: a regulation solution relying on budgeting the number of shared hardware accesses to guarantee time properties. To be able to do so, first we perform at design time an automatic standalone characterization of the critical applications to figure out the available budgets; and second we rely on this budget to take scheduling decision about non-critical applications at runtime.

5.2 Principles of the Budget-Based RunTime Engine

The principles of the approach, already presented in Deliverable D3.2, consist in determining, per timeslot, a maximum budget to allocate to low-critical applications in terms of resource accesses. When this budget is spent, low-critical applications are suspended until the next timeslot, as depicted in first timeslot of Figure 5.2.

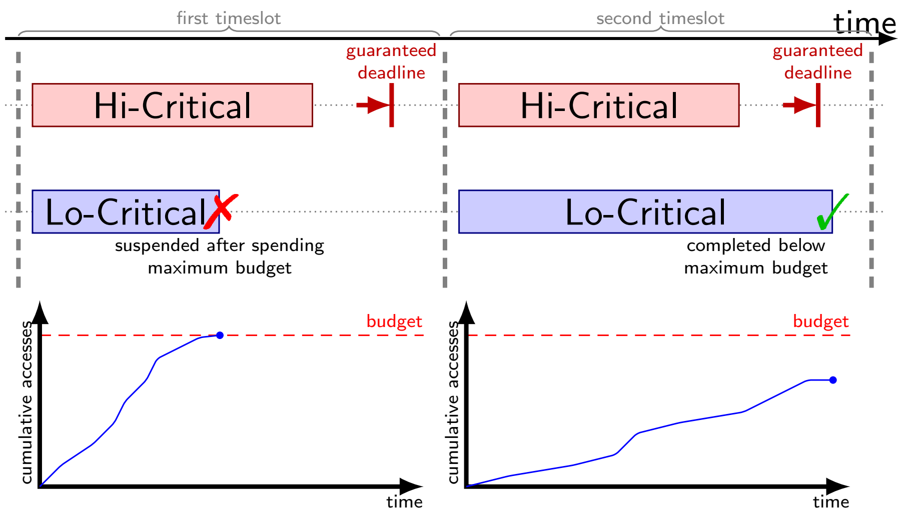


Figure 5.2: BB-RTE Principles

The budget is computed per timeslot and per hardware resource to ensure that the critical applications will be matching their deadline in each particular timeslot.

During the second timeslot of Figure 5.2, the low-critical applications manage to terminate without spending the whole allocated budget, again guaranteeing that the critical applications will match their deadlines during this timeslot.

With such a process based on budgeting, the major challenge consists in **determining the budgets** that guarantee the time behavior of high-critical applications.

5.3 Characterization & Regulation Process

The process to determine resource budgeting, depicted in Figure 5.3, is performed with two major steps: First, both the architecture and the high-critical applications are characterized in an **offline characterization** step with regards to every hardware resource 1) to determine the total available resource budget; 2) to quantify the resource requirements of the high-critical task; and 3) to figure out the maximum level of extra accesses to the resource before hampering the high-critical task.

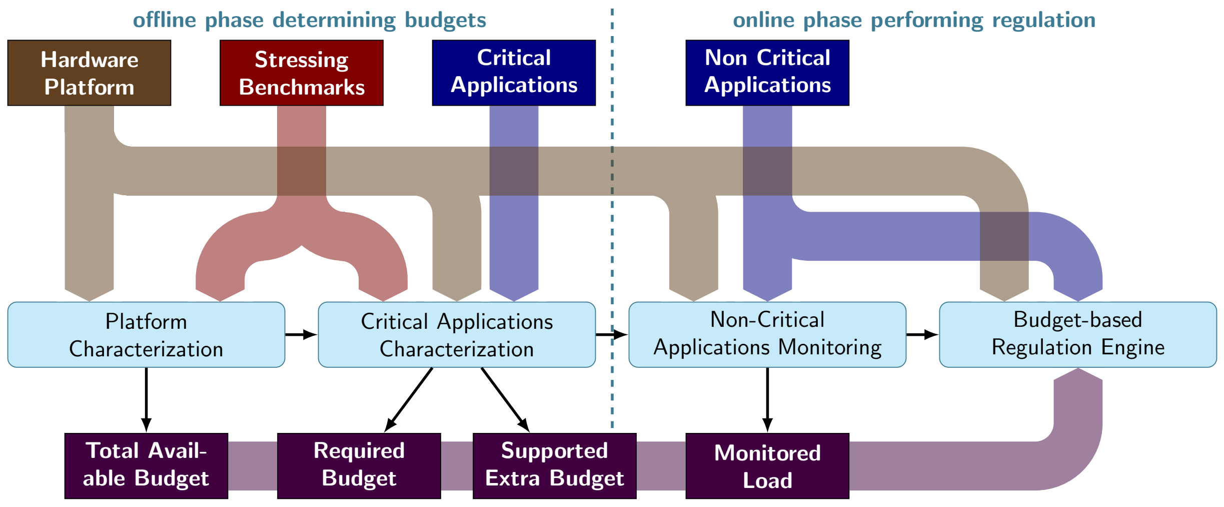

offline phase determining budgets
online phase performing regulation
Hardware Platform
Stressing Benchmarks
Critical Applications
Non Critical Applications
Platform Characterization
Critical Applications Characterization
Non-Critical Applications Monitoring
Budget-based Regulation Engine
Total Available Budget
Required Budget
Supported Extra Budget
Monitored Load

Figure 5.3: Timing integrity process for mixed time-critical systems

Second, in a **runtime regulation** step, from the characterization information is inferred the maximum number of resource accesses allowed for non critical tasks, these tasks being suspended by a **Budget-Based RunTime Engine (BB-RTE)** once they reach this maximum number of total access during a given time slot.

5.3.1 Hardware characterization & budgeting

As introduced in Figure 5.3, in the first sub-step of the offline characterization phase, we perform a characterization of the target hardware platform. This hardware characterization consists in defining a set of low-level (assembly code) Stressing Benchmarks. Each of these stressing benchmarks is responsible for stressing a particular hardware resource of the selected multi-core target, by multiplying the number of accesses to this particular resource.

By progressively stressing each resource while monitoring both the execution time and the effective number of access to this resource thanks to Performance Monitor Counters (PMC) [32], we are able to determine the maximum available bandwidth in terms of accesses to this resource, and that corresponds to the **total available budget** for the resource.

By iterating over all the potentially shared hardware resources, we obtain a vector of such total budgets that fully characterize the hardware limitations of the selected platform.

5.3.2 Critical application characterization & budgeting

During the second characterization sub-step of Figure 5.3, we characterize the usage made by high-critical applications of the shared hardware resources. To do so, we run the high-critical applications concurrently with the stressing benchmarks described above, progressively increasing the stressing level, and only monitoring the effect in terms of runtime of the high-critical applications.

It allows us to extract two different kinds of information: First the **required per-resource budget** needed by our high-critical applications; and second, the level of extra resource access supported by our high-critical applications before being significantly slowed down. This **supported extra accesses** is the access budget that can be safely used by the low-critical applications.

The process to determine the acceptable level of slowdown and the associated extra access budget is depicted in Figure 5.4. The y-axis represents, during the current timeslot, the maximum observed runtime of the monitored application. The x-axis represents the extra access load performed on the associated hardware resource by the stressing benchmarks.

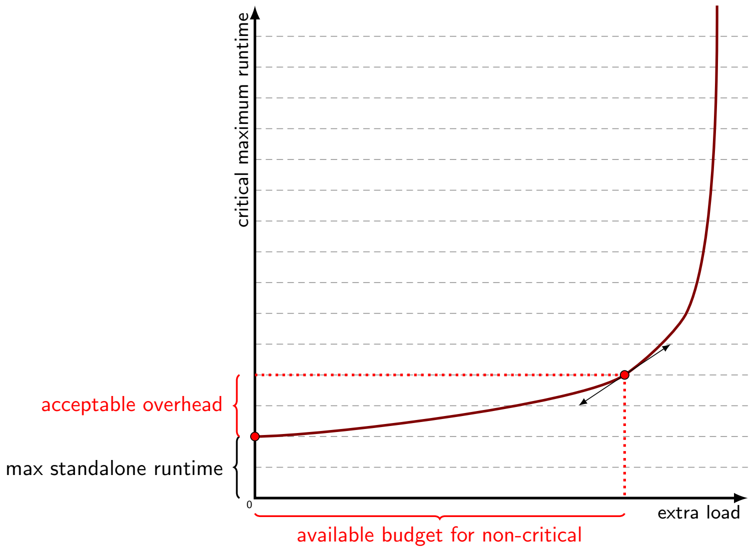


Figure 5.4: Determining an acceptable level of slowdown and the associated extra access budget

The leftmost point in the chart corresponds to the application running alone in isolation (with a null stressing benchmark activity). It therefore corresponds to the classical WCET of the application running in isolation. The rightmost point in the chart corresponds to a permanent maximum load from the stressing benchmark actually preventing the monitored application to access the required resource.

Selecting an acceptable level of slowdown and the associated extra access budget is performed by selecting a point on the chart. The projection of this point on the y-axis then corresponds to the acceptable level of slowdown and the projection on the x-axis directly provides the extra access budget available for non-critical applications with an acceptable impact on critical applications.

We used two different techniques to select this point: First by directly selecting a maximum acceptable slowdown, but doing so for every hardware resource led us to too much over-provisioning, failing to fully exploit the multi-core efficiency. Second, we defined a maximum slope for the curve. This second solution allowed us to vary the level of slowdown relatively to the shape of the curve, focusing on the hardware resources the high-critical application was the most sensitive to.

By repeating this procedure for each shared hardware resource, we obtain again a budget in the form of a vector of both the required amount of access by the critical application (the leftmost point of the curve) and the number of supported extra access (the selected point of the curve).

Characterization steps involve large-scale experimentation due to the limited monitoring resources of multi-core processors. These architectures usually provide from tens to hundreds of hardware events that can be monitored, but only allow to monitor a few PMC at a given time (6 for ARM-v8 architectures for instance). As a consequence, testing all the countable PMC events involves many days of experimentation, and this characterization phases have to be performed off-line. Anyhow, this characterization only needs to be performed once per critical application for a particular hardware target, and such a characterization could also provide useful information for the qualification or certification documents.

5.3.3 Budget-Based RunTime Engine (BB-RTE)

Once the characterization phases are over and all budgets have been gathered, both high-critical and low-critical applications can be deployed on the hardware target together with the runtime engine. As shown in Figure 5.3, the online regulation phase happens during the execution and consists in two sub-steps.

First, using the same process as the characterization phases, the low-critical applications are monitored with PMC counters, this time not to compute a budget, but to monitor the load in terms of hardware resource accesses.

Second, the BB-RTE runtime engine compares this load with the maximum extra budgets to decide if, during the current timeslot, non-critical tasks may continue executing or if they need to be suspended until the next timeslot. Doing so makes sure the slowdown of critical tasks does not hamper their ability to match their deadlines.

One of the specific challenges of the BB-RTE runtime engine for time-critical systems is that time intrusiveness of the associated monitoring features has to be kept minimal, not to bias the time characterization results. Also, the BB-RTE itself should make a minimal usage of shared hardware resources to not impact the resource access budgets. The final intrusiveness footprint of both the monitoring features and the BB-RTE engine will be presented in the results section.

5.4 Experimental Setup

The BB-RTE engine has been developed on top of PikeOS [4], which is both a hypervisor and a real-time operating system relying on partitioning. We relied on the METrICS toolsuite [15], described in Deliverable D4.2, to perform the characterization steps and implemented the runtime engine as a native PikeOS partition, altering the scheduling in real-time as the monitoring information is gathered.

5.4.1 Target Architecture

The results presented in the Section 5.5 were evaluated on an ARM Juno board [8] embedding a big.LITTLE architecture composed of a cluster of 2 high-performance Cortex A72 cores and a cluster of 4 more predictable Cortex A53 cores. The block diagram of the architecture is presented in Figure 5.5.

The highlighted parts in the figure correspond to the shared memory path from the cores towards the main memory that are prone to timing interference. Other initiators like the MALI GPU, the PCI express links and the DMA controllers can also create contention on these shared resource, but for this work we will only focus on memory contention from the cores.

Alongside this memory path, the shared hardware resources are: The shared L2 cache that is shared by all the cores of each cluster. The CCI-400 cache-coherent interconnect (a high-bandwidth crossbar interconnect combining routing and coherency functions), and two DDR3 controllers served through the AMBA-compliant DMC-400 memory controller.

The sizes of the memory structures alongside this path are presented in Table 5.1. This information will be useful when trying to design stressing benchmarks dedicated at stressing a particular hardware resource.

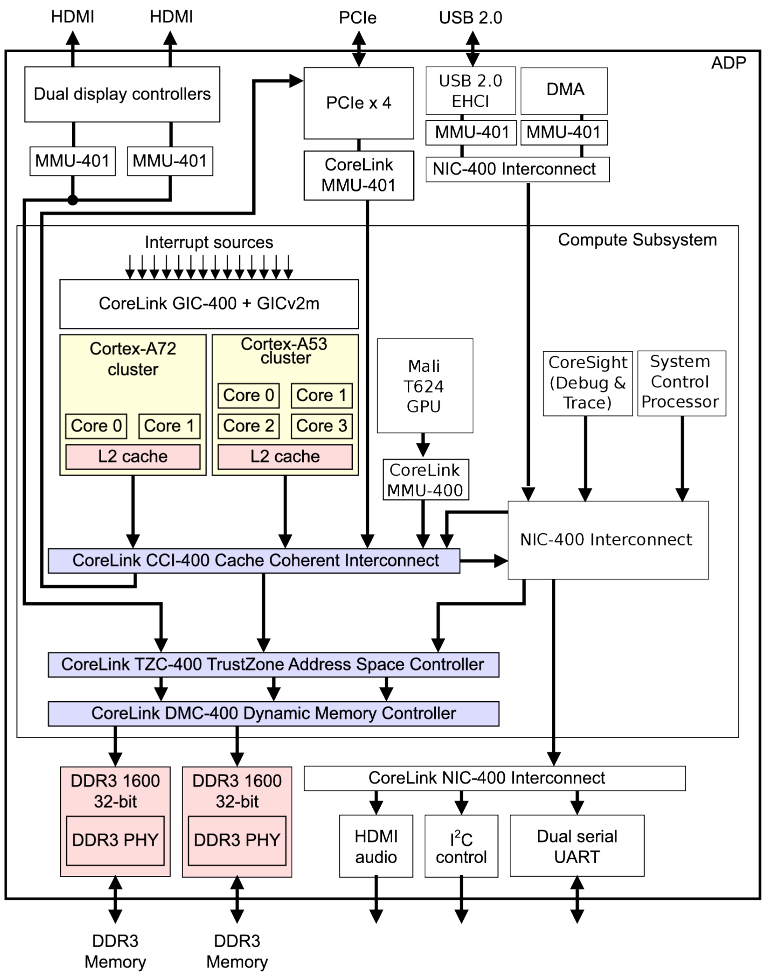


Figure 5.5: Simplified block diagram of the ARM Juno Board

	A72 cluster	A53 cluster
Number of cores	2	4
IL1 cache size	48KB	32KB
IL1 cache line size	64B	64B
IL1 associativity	3	2
DL1 cache size	32KB	32KB
DL1 cache line size	32B	64B
DL1 associativity	2	4
L2 cache size	2MB	1MB
L2 cache line size	64	64
L2 associativity	16	16

Table 5.1: JUNO Board memory structure sizes

5.4.2 Monitoring facilities

In Section 5.3, we stated that the budget characterization is performed by using Performance Monitor Counters (PMC).

The ARM-v8 architecture includes a Performance Monitors Unit (PMU), a non-invasive resource primarily used for debugging that provides information about the internal operations in the core. It includes a 64-bit cycle counter and 6 performance monitor counters able to count the occurrence of around 60 different events.

Among these events we selected a subset of 13 events that correspond to either private or shared hardware resources composing the memory path, listed in Table 5.2. The description appearing in the table is the level of details provided by the documentation on each countable event.

Performance counter	Counting
inst_retired	Instruction architecturally executed
cpu_cycles	Cycles
L1I_cache	Level 1 instruction cache access
L1I_cache_refill	Level 1 instruction cache refill
L1D_cache	Level 1 data cache access
L1D_cache_refill	Level 1 data cache refill
L1D_cache_wb	Level 1 data cache write-backs
L2D_cache	Level 2 cache access for data
L2D_cache_refill	Level 2 cache refill for data
L2D_cache_wb	Level 2 cache write-backs for data
bus_access	Bus access
mem_request	Memory access
prefetch	Linefill because of prefetch

Table 5.2: Hardware events measured with Performance Monitor Counters

A trial and error experimental process was therefore necessary to understand the exact meaning of each of these counters. `inst_retired` corresponds to the number of executed instructions whereas `cpu_cycles` corresponds to the number of CPU cycles measured so far. Therefore the ratio of the two correspond to the regular instruction per cycle (IPC) measurement commonly used to determine an application performance.

The events `L1D_cache`, `L1I_cache` and `L2_cache` indicate the number of accesses to the L1 data cache, L1 instruction cache and L2 cache by load/store requests respectively. The `refill` counterparts indicate how many times a cache line was obtained from a higher memory structure (e.g. the number of times a cache line was provided to the L1 cache by the L2 cache) so it is a good measure of the cache misses that indicates that a higher memory level in the datapath is accessed.

The `write-back` counterparts indicate how many times a modified cache line was written back to higher level memory so that the local cache line could be freed to fit a new data. This write back traffic, not appearing directly in the source code could be a significant part of the memory traffic.

The events `bus_request` and `mem_request` are respectively counting the requests on the local bus connecting core and caches, and the number of requests leaving the core to be sent to the CCI-400 interconnect. This local bus traffic does not only correspond to memory accesses in the code, but also to instruction fetches and to the coherency traffic.

Finally, `prefetch` indicates the number of times a cache line of the L2 cache is filled due to an automatic hardware prefetch of the cache line. This feature has proven to be very problematic for our stressing benchmarks later presented in this section as the prefetching was limiting the stress level, as shown in the evaluation section of the chapter.

Some of the other hardware resources appearing in Figure 5.5 also provide some debugging support, but the documentation is often lacking or only available under NDA, and restricted to third-party providers of debugging probes such as Lauterbach. We definitely want to monitor these SoC-level events in the future, but we considered them out-of-scope for this work.

5.4.3 Target Environment

In order to measure precise execution times and to sample hardware performance counters, we used our METrICS toolsuite. Its main concept is to sample performance counters (including the cycle-count register) with a very short timing overhead, before and after the code sequences to monitor.

The METrICS toolsuite is composed of several elements: a kernel driver used to configure the hardware performance counters (as this requires privileged instructions), a library providing measurement probes to the application, and a Collector performing various initialization and transmission of measurement results. We heavily modified this latter component to include the part corresponding to the Run Time Engine. In addition, the METrICS suite includes host-side scripts and tools for measurement campaign automation, post-processing of raw data, and visualization.

The latency of a METrICS probe has been evaluated to be less than 392ns at worst. This whole probe thus has a comparable latency to a system call, and is precise enough considering the millisecond deadlines we have in the mixed-critical prototype presented in Section 6.

The initial behavior of the collector component was to run out of the monitored application operational cycles, to minimize the impact it had on the application timings. More precisely, it was 1) performing initialization and performance counter selection prior to running the monitored applications, 2) being completely suspended during the application operational cycles, 3) being activated again at the end of the application to dump the collected data to the host.

As appearing on Figure 5.6, we transformed in the BB-RTE context the collector component into the runtime engine. It is now also running during the application operational cycle to monitor resource usage, and to suspend low-critical tasks when they have consumed all their budgets. In the experimental context, we kept the third dumping phase to also collect the applications runtime and performance counter data including the number of deadline misses observed during the applicative phase.

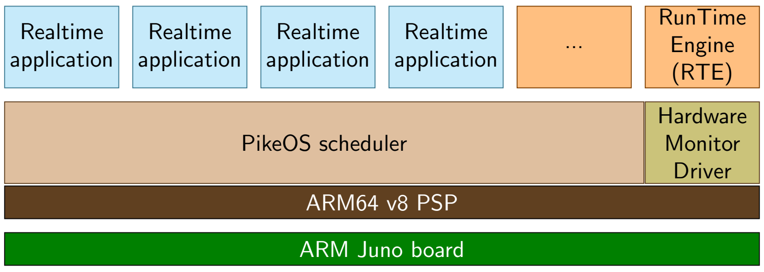


Figure 5.6: METrICS infrastructure in the context of BB-RTE

The application deployment also appears in Figure 5.6. We dedicated the two A72 cores to running both the operating system and the runtime engine. Several tests have shown that it was more efficient to run the RTE on the same core as the operating system due to the large number of system calls required to alter the scheduling. We therefore reserved the more deterministic A53 cores to the applications. Further deployment details for this cores will be provided in the evaluation section.

5.4.4 Stressing benchmarks

The mixed-critical applications we will use for the evaluation are described in Section 6, but the characterization steps described in Figure 5.3 also involve stressing benchmarks.

These benchmarks are simple applications performing repeated accesses to a shared resource. This involves executing a number of load or store instructions with regards to a memory region or a memory-mapped peripheral. In order to focus the stress applied by these benchmarks on a particular resource, we developed them in assembly language, thus allowing the greater control on the low-level behavior of the stressing benchmark.

The accesses are parametrized in four ways: the direction of the transfer, the amount of data transferred, the address offset between consecutive accesses, and the repetition rate. The first parameter is either read or write. The second parameter, representing the buffer size, spans from 1KB to 2MB in power of 2 increments. This allows us to experiment the behavior of data fitting or not in L1 and L2 caches. The address offset, called stride, is either 4 bytes for continuous accesses, 32 bytes to stress A72 L1 cache lines, or 64 bytes to stress A53 L1 and L2 cache lines. To vary the amount of accesses performed in a given time, we add a configurable number of NOP instructions (modeling calculations performed in isolation) per load or store instruction. This parameter spans from 0 (full stress) to 100 NOPs per access.

5.5 BB-RTE Evaluation

This section first reports the budget characterization results obtained following the process described in Section 5.3 and summarized by Figure 5.3. Then, it presents an evaluation of the runtime engine, by comparing the results in terms of deadline misses compared to an unregulated run of the same set of applications.

5.5.1 Platform characterization and total available budget

Performing the hardware platform characterization of Figure 5.3 involves concurrently running stressing benchmarks until some saturation phenomena are observed.

On the first A53 core, we run a monitored stressing benchmark with a fixed number of iterations and a full stressing profile (zero NOPs). On the other 3 A53 cores, we run the stressing benchmark as an infinite loop, starting it before the monitored one to be sure that the monitored benchmark is run under stress. For these stressing cores, we are varying the buffer size and the stride concurrently with the monitored core. We are also varying independently the number of NOPs (and therefore the amount of concurrent accesses to the resources).

By varying the stride, we impact which memory structure and therefore which part of the architecture will be exercised: With a 4-byte stride, we will exercise the L1 cache maximizing the number of L1 hits, while strides larger than the L2 cache line size will mostly produce L1 and L2 cache misses, actually exercising the DDR controller and the memory.

Table 5.3 shows the runtime results of such a characterization, varying the buffer size (how well the data will fit in the different cache structures), as the stressing level varying to maximum stress (no NOP instruction) to no stress (no load instructions).

		runtime (cycles)				
Buffer size	stress	min	25%	median	75%	max
16384	none	12338	12352	12360	12369	13101
16384	max	12338	12352	12360	12369	13175
32768	none	24680	24786	24839	24959	26129
32768	max	24679	24785	24838	24960	26272
65536	none	49340	49951	50300	50338	52499
65536	max	49339	49531	49752	50759	52795
524288	none	393962	394096	394219	394633	417123
524288	max	393978	394535	394785	395224	421488
1048576	none	789994	791536	792055	792817	827872
1048576	max	787995	791789	793538	795033	823978
2097152	none	1599528	1601997	1603697	1606266	1751393
2097152	max	1577078	1603203	1605720	1607560	1732193

Table 5.3: Run-time variation observed on the monitored core with a 4-byte stride

Selecting a 4-byte stride puts the L1 data cache under pressure, but this cache structure is private to each core. As a consequence, results presented in Table 5.3 show very small variations while increasing the number of NOPs, the runtime only varying as the buffer size grows. Due to the high L1 hit ratio, having large buffer size not fitting in the L1 does not impact the performance either.

Selecting larger stride values allows us to characterize further hardware components shared by all the A53 cores starting with the L2 cache and even with the A72 cores beyond the CCI 400.

As we were limited to count core-related events with the Performance Monitor Counters, we mainly focused on the level 2 cache, that is also the first hardware memory resource shared by all the cores of the A53 cluster.

Table 5.4 presents the running time distribution on the monitored core while varying again the buffer size and the number of NOP instructions, to compare standalone versus maximum stress deployments.

		runtime (cycles)				
Buffer size	stress	min	25%	median	75%	max
16384	none	818	826	832	839	2883
16384	max	818	824	831	839	4975
32768	none	1728	1866	1902	1955	9739
32768	max	1734	1883	1929	2006	7382
65536	none	6772	8237	8885	10131	19378
65536	max	6720	9080	9771	10777	20170
524288	none	79625	107638	108660	111344	179419
524288	max	97018	109665	110268	110897	177007
1048576	none	169753	210845	214317	229595	366017
1048576	max	207393	227506	237746	242661	353066
2097152	none	478261	522073	526574	529720	725379
2097152	max	500723	547663	551346	553397	705256

Table 5.4: Run-time variation observed on the monitored core while performing L2 cache misses

Whereas the results should allow us to observe timing interference with an expected runtime variability of fully-stressed versus standalone larger than x4, we still observe very little difference between the two standalone and stressed versions.

Further tests, including unrolling the stressing benchmark loop to further increase the load or store ratio did not significantly change the above results. We therefore focused on the results corresponding to the maximum stress condition (2MB buffer size, with 0 NOP per iteration on the unrolled version) and studied the associated hardware counters presented in Table 5.5.

counter	min	median	max
l2d_cache	33645	34654	35129
mem_request	32473	32636	32863
prefetch	30304	31716	32001

Table 5.5: Performance Monitor Counters related to the L2 cache under stressing condition

In the worst case, out of the 35K data accesses to the L2 cache, 32K accesses are going to the external memory interface, meaning that our configuration successfully maximizes the number of L2 cache misses in order to maximize the interference on the interconnect. However, we can also observe 32K occurrences of prefetch, meaning that the hardware prefetcher successfully manages to capture the access pattern of our stressing benchmark and was able to anticipate nearly all of the external memory accesses. As a consequence, our stressing benchmark fails to effectively continuously stress the hardware resource.

In such a configuration the hardware prefetcher, instead of worsening runtime variability, smooths the contention on memory accesses. This seems to be a case of unexpected positive contribution to determinism from a dynamic, non-deterministic mechanism.

Also, the behavior of the prefetcher for requests coming from a particular core seems to depend on the memory access activity from the other cores, leading to further thickening the runtime dispersion. We particularly observed that by running the benchmark concurrently with nothing, versus concurently with a benchmark only issuinf NOPs. Up to a 13% speedup was observed with the nop-only corunner.

We tried to disable the hardware prefetcher, unfortunately we observed that it is periodically turned back on. We suspect the System Control Processor to be responsible for that. However, from our real time system, we have no control nor access on this particular core.

An alternative approach would be to develop another stressing benchmark, performing accesses that are more difficult for the prefetcher to predict. This would for example involve pseudo-random address increments.

5.5.2 Critical application characterization and extra supported budget

The characterization of the critical application was performed by running the FMS application on the first A53 core. Again, thanks to METrICS, we instrumented the FMS application by inserting probes around each task composing the application, and at the level of a full operational cycle from the sensor task to the trajectory computation.

We first ran the the FMS application standalone, with the other cores being idle, to collect the budget in terms of L2 accesses required by each timeslot of the application. Contrary to the stressing benchmarks, that are regular in their behavior and throughput requirements, the observed runtimes and resource requirements vary a lot with the FMS application, as all the tasks are not triggered in every 200ms timeslots due to heterogeneous periods.

The LOC$_{C4}$ task for instance is only executed in 1 out of 5 periods, whereas the LOC$_{C1}$ task is executed every period. As expected the timeslots corresponding to the least common multiple of all the periods, during which all the tasks must execute, is the one with the larger observed runtime and L2 access count, as pointed out by Figure 5.7.

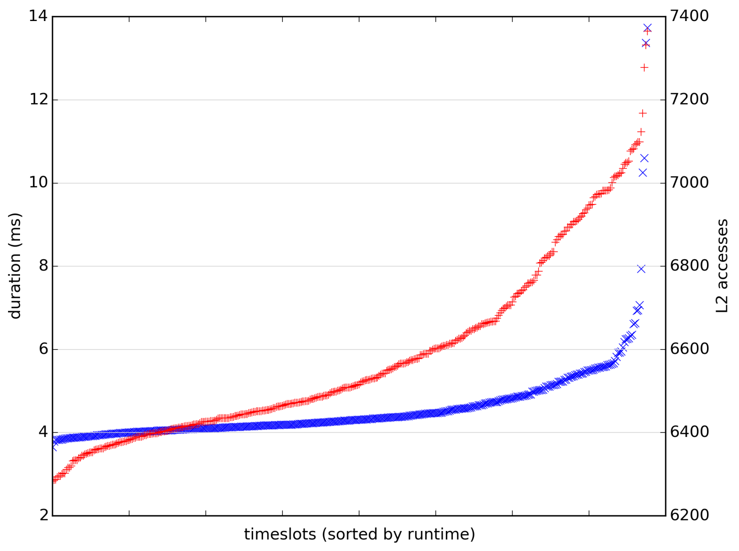


Figure 5.7: Variation of the runtime (in blue x) and L2 accesses (in red +) while running the FMS application standalone

The next step, as defined in Figure 5.3, would have been to run the FMS application on the first A53 core, while running stressing benchmarks on the other A53 cores to build a figure looking like Figure 5.4. Such experiments led to the same issue as presented in the previous section, with the benchmarks failing to successively stress the architecture.

As a consequence, we failed to build such a figure with well identified asymtotes, and had to rely on an alternative technique:

Let $R = [R_0, R_1, \ldots, R_N]$ be the set of all the required access budgets per timeslot for the critical FMS application running standalone without any activation of the asynchronous tasks. We set $E = [E_0, E_1, \ldots, E_N]$ the set of per-timeslot extra supported budget such as $E_i = (max(R) - R_i) \times 20\%$.

Doing so is not representative of the total budget available on the target platform, but this way the supported extra budget is inversely proportional to the resource usage performed by the critical application, as it should really be.

5.5.3 Evaluation of the RunTime Engine

As a baseline for the evaluation of the runtime engine we started to deploy the FMS application on the first A53 core, and the BiQuad applications on the remaining A53 cores. Doing so, we observed a deadline miss ratio (number of timeslot with a deadline miss compared to total number of timeslots) of 24.7%.

We then run the same application deployment, this time supervised by the Budget Based Runtime Engine. As a budget, we used the previously computed budget E. And again we ran the FMS application without any asynchronous tasks. The deadline miss ratio decreased to 2.6% while the number of slots with suspended non-critical tasks increased to 30.9%, proving the ability of the Runtime Engine to suspend low-critical tasks when they are endangering the high-critical ones.

The high-critical application normally also encompasses some asynchronous tasks that we ignored so far. However, some of these sporadic tasks have a huge memory footprint, especially the tasks from the Trajectory task group that are performing random accesses to the 100MB large navigation database.

It would make no sense to have a preset static scenario, forcing in which timeslot each asynchronous task would be triggered. It would be similar to considering these tasks as periodic with a very large period. On the other hand, a random scenario will raise a reproducibility issue as we are gathering the timeslot statistics.

As a consequence, we created randomly two static scenarios for asynchronous task triggering, one was used during the characterization phases to compute an R' extra supported budget set, and the second one was used while running under the supervision of the runtime engine. In such a scenario we observed a final miss ratio of 13.0% and a suspend ratio of 52.7%.

On one hand, the increase in suspend ratio is due to the over-provisioning of the timeslots with some asynchronous tasks in E', leading to unnecessary suspends at runtime. One the other hand, the increased deadline miss ratio could be explained by the lack of such provisioning in the timeslots were the asynchronous tasks finally occur.

5.5.4 Limitation of the approach

Beyond the issues related to the prefetcher preventing us to sufficently stress the hardware resources, we identified a set of issues and limitations while evaluating the runtime engine.

First, relying on Performance Monitor Counters restricts us to the hardware resources within the cores. As a consequence, we were not able to evaluate shared hardware resources such as the DDR controller. With the prefetchers disabled, furthermore increasing the stride of the stressing benchmarks would have caused memory page reloads at the level of the memory controller, causing extra delays. These sources of interference have not yet been evaluated.

A more fundamental limitation is the issue caused by the aperiodic tasks. When pre-computing budgets with characterization phases, we face the same issue as with machine learning with both over-learning, and outlier issues, and as a result we are facing either deadline misses or unnecessary suspends.

Also having per-timeslot budget is not practical for real applications running for hours. During our evaluation, the FMS ran for up to 5 minutes. With 200ms timeslots, which involves 1500 different budgets for this critical application. Also, because of unpredictable aperiodic tasks, it is not really possible to identify shorter repeating patterns.

We performed tests with an unified average timeslot budget, but doing so only reduces the number of deadline misses by 2.04%. Using a maximum timeslot budget on the other hand eliminates all the deadline misses but the low-critical tasks are systematically suspended.

5.6 BB-RTE Thermal Protection

This section provides an overview on how thermal protection can be added to BB-RTE. The scheme is based on the Thermal Isolation Servers scheme detailed in D3.2. Additional details and results will be provided in D6.3. Our strategy has the following components.

1. Using thermal callibration tests, determine the thermal model of the ARM Juno Board.

2. Using additional thermal callibration tests, determine the maximum system-wide temperature caused by the execution of the HI-Critical Application. We call this temperature $\mathbf{\Theta^{Hi}}$
3. Compute the thermal budget for Lo-Critical Applications. This budget equates to $\mathbf{\Lambda^{Lo}} = \mathbf{\Theta^{\Delta}} - \mathbf{\Theta^{Hi}}$. $\mathbf{\Lambda^{Lo}}$ characterizes the maximum allowed temperature increase caused by the execution of Lo-Criticality applications.
4. Using the thermal model computed in step1, ensure during run-time that $\mathbf{\Lambda^{Lo}}$ is not exceeded.

5.6.1 Application and platform specifics

As stated in section 5.5, BB-RTE has the high critical FMS application running in the core 0 of the A53 cluster. The less-critical BiQuad application is executed on cores $\{1, 2, 3\}$ of the A53 cluster. Thermal protection needs to be provided to the FMS application.

Juno board does not have a per-core temperature sensor and reports a single temperature value for the A53 cluster. Due to this limitation, the thermal callibration tests need to be more extensive and the thermal model will be more conservative, compared to the scenario where per-core thermal sensors were available. This is because, under the current setup, the thermal impact of an application (on its own core and neighbouring cores) will need to be inferred from this single temperature reading of the A53 cluster. The details of the thermal callibration tests and the corresponding thermal model results will be reported in D6.3.

5.7 Hardware Support for Mixed-Criticality Multicore Systems

In deliverable D4.1 we reported our hardware support for the SnapDragon 810 processor. Due to the known failure to integrate all technologies on that board, this technology was also integrated on a Juno Development Board, which implements a similar architecture, so the low-level software could be reused with minor modifications. However, for the sake of exploiting this technology in one of the use cases, we further integrated it in the Infineon AURIX TC27x processor family. We report next our experience in understanding the Performance Monitoring Counter (PMC) support in such state-of-the-art automotive processor (using only the available documentation as reference), with the goal of building a multicore-contention model as that presented in [11] to support the RTE. In particular, for this work we focus on the Infineon AURIX TC-275 board [18].

5.7.1 Platform

The TC-275 is a TriCore platform comprising an energy-efficient TC1.6E core (Core 0) and two instances of a performance-efficient TC1.6P core (Core 1 and Core 2), see Figure 5.8, with Cores 0 and 1 executing in parallel with two lockstep checker cores (not shown in the figure). The two core models slightly differ in terms of memory interface and PMC support but in this work we focus exclusively on the high-performance core as the candidate core to execute time-critical applications. The TC1.6P is a super-scalar core featuring 3 pipelines (Integer, Load-Store and Loop) that may enable multiple instructions to be issued in each cycle, in the best case. The TC1.6P core is equipped with local 32KB program and 120KB data scratchpads, complemented with relatively smaller 16KB instruction and 8KB data set-associative caches. The cores are connected to the shared memory interface through the Shared Resource Interconnect (SRI) crossbar, which potentially reduces the contention in accessing shared hardware resources as accesses to different peripherals can happen in parallel. The memory interface comprises a Local Memory Unit (LMU), providing 32KB of shared SRAM, a FLASH device (PMU0 - Program Memory Unit) with separate SRI slave interfaces, which can be accessed both in cacheable or non-cacheable mode. Each slave interface in the SRI crossbar comes with its own arbiter and, beside a common SRI arbitration cost, each peripheral guarantees different access latencies.

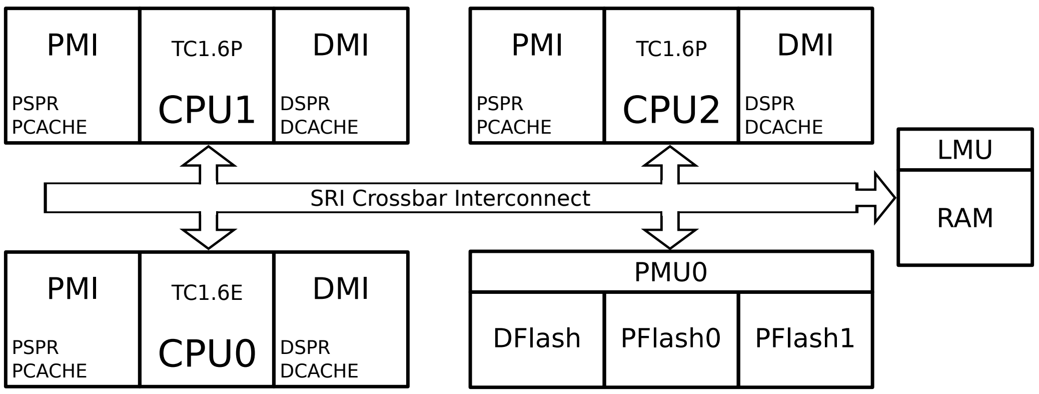


Figure 5.8: Module view of the AURIX TC27x.

The TC-27x processor supports the debug and testing process through several features, including an Emulation memory (EMEM), support to Overlay data accesses and a set of Debug registers and PMCs per core. However, in this work we only focus on PMC support. PMC support includes a local cycle counter (which can be used as a reference in addition to the global timer) and an instruction counter. In addition, each core has 3 general-purpose performance counters that can be configured to record up to 3 types of events simultaneously. Each counter can log only a fixed and disjoint subset of the available events to record, so that multiple runs may be required to collect comprehensive PMC information. Note that the TC1.6E core naturally features slightly different PMCs as it includes a data buffer instead of a data cache. The TC-27x manual provides a very concise description of each PMC mode [18], often limited to a single sentence.

5.7.2 Methodology

In order to better understand how to exploit the available PMCs fitness to build our contention-analysis model, we first examined the publicly available documentation, and the manuals coming with the board, to derive their semantics. Then we proceeded by designing a set of minimal experiments to trigger those events that, to the best of our understanding, each PMC was expected to count. This methodology allowed us to confirm our understanding of the relevant events for the PMCs and to assess whether they are effective indicators for a resource usage analysis.

Once we selected the events of interest, we enabled and configured properly PMCs by writing into the Counter Control Register (CCR). However, the processor manual is clear in stating that there is no assurance on the exact stage in the pipeline when counters will actually start counting. In fact, the TC1.6 processor manual recommends that each `mtcr` instruction – needed to configure the CCR – should be followed by an `isync` instruction, to flush the pipelines and ensure all instructions fetched after the `mtcr` see the effects of the CCR update [19]. We defined all experiments using assembly code and preventing any side-effects from the compilation process. We used a custom linker script to fetch code and data from different locations (PMU, LMU, Scratchpad RAM). In order to reduce the noise in our observations, we: (i) set and enable the counter; (ii) execute a small code snippet to trigger the target event; and (iii) disable the counters before reading the respective PMC values (see Listing 5.1). We conveniently added additional padding (`nops`) to enforce a proper alignment when addressing counters related to memory accesses.

Listing 5.1: Code pattern used in experiments

```
mtcr 0xFC00, %[config]  // enable counter
                        // with specific PMCs
isync                   // flush pipeline
...                     // code snippet
mtcr 0xFC00, %[zero]    // disable counter
isync                   // flush pipeline
```

5.7.3 Difficulties in PMC assessment

By only relying on the information at our reach, the assessment of the PMC support in the TC-27x required us to undergo a disproportionate reverse-engineering effort to try to fill the gap between the overly-concise specification of events and their observed behavior. In fact, despite our relatively good understanding of the platform, we were not able to assess the events with the desired degree of confidence. In the following we report the main difficulties we encountered in our evaluation. We observe that most of the impediments are straightforwardly curable with a more detailed description of the PMCs' mode of operation.

Shaving off measurement noise. A first problem we had was related to the precision of our observations, which is the main reason why we coded our experiments at the assembler level. The adoption of the CCR handling protocol (i.e. with pipeline flush) ensures that PMCs are enabled; however, it also introduces some undesired noise. We decided to execute an "empty" experiment (i.e. a sequence $\mathtt{mtcr} - \mathtt{isync} - \mathtt{mtcr} - \mathtt{isync}$) in view to consider the noise it produces as a constant to be subtracted to any other measurement. However, we found out that the results obtained on this small example do incur a (bounded) noise of ± 1 for PMCs read that is dependent on instruction alignment and selected CCR configuration. Some configurations seem to suffer less fluctuation than others. We observed that this phenomenon is not triggered with larger sequences of instructions. It is worth noting this effect is caused by the need of reverse-engineering itself and could be easily fixed with proper documentation.

Events or cycles? When reading the technical specification of traceable events, a critical aspect to understand is whether events are tacked down in terms of number of occurrence or cycles, as it makes a huge difference when building a resource usage model for timing analysis. Unfortunately, one-liner descriptions turned out to be too ambiguous for us to discern on this matter. Resource usage and contention analysis models usually rely on events (accesses or stalls) rather than cycles and this point should be clarified.

In our case, for example, we tried to understand whether the PMEM_STALL and DMEM_STALL counters were counting either single stall events or the number of cycles incurred because of memory stall events. This point was not clearly stated in the manual and both interpretations were possible. In fact, the reader is led to interpret PMEM_STALL and DMEM_STALL as the number of stall events incurred when fetching code and data, as other counters like LS_DISPATCH_STALL are explicitly described as counting cycles instead. We designed a simple experiment, triggering a load from external memory to unequivocally determine the semantics of the PMC. We empirically determined that, in contrast to our original expectations, PMEM_STALL and DMEM_STALL actually read cycles. In fact, we repeated the same experiments fetching data from different locations (LMU, PFlash, DFlash) and observed a variation in the PMCs that would not have appeared if they were counting occurrences of events. Again, we believe that this ambiguity could be avoided with proper documentation.

Granularity of observed events. Our assessment of PMC support also included the granularity of counted events, which consisted in comparing the PMC collected values against the expected ones (based on the specification). When focusing on the program cache performance, we had hard times in interpreting the PCACHE_HIT counter (while no problem was encountered in the complementary PCACHE_MISS). Our difficulties were only in part caused by the interaction of the different pipelines. In fact, we experienced some inconsistency in our experiments between cache statistics when dealing with different instruction sizes (16/32 bits or a mixture thereof). Reduced-size instructions have been introduced in some RISC ISAs as a means to reduce the program memory footprint [3, 24, 19]. Most instructions in the TC1.6 ISA are 16-bit instructions. The cache behavior with different instruction sizes depends on the dimension of the fetch interface and is complicated by the necessary alignment of the fetched instructions. In our platform we observed that the cache behavior (in terms of number of hits/misses) was as expected only when 32 bit instructions were exclusively used. The number of cache hits was instead particularly counter-intuitive when mixing different instruction sizes.

As an example, we considered a chunk of 256 bytes, worth of 128 sequential 2 bytes instructions and 512 bytes, worth of 128 sequential 4 bytes instructions. For the former our platform reports

17 hits and 9 misses, for the latter our platform reports 25 hits and 9 misses. While the number of misses always stays the same (stemming from fetching 256 bytes in eight 32 bytes lines, plus one additional miss for a cache line holding the counter disabling instruction) the number of hits reported differ depending on the instruction size, which further complicates analysis.

Using PMCs to break down maximum latencies. Analytical models also make use of maximum latencies to conservatively overapproximate the interference when accessing shared resources. Breaking down the observed latencies of different slave interfaces on the TC-27x SRI is fundamental for any contention model to avoid unnecessary sources of pessimism stemming from always assuming the absolute worst case.

To assess the cost of data fetches, we exploited the DMEM_STALL and LS_DISPATCH_STALL counter to measure the latencies incurred by diverse external memory. The latencies observed for LMU, DFLASH and other cores' Data Scratchpad-DSPR (local DSPR accesses incurs no latency) were in line with the specification. We empirically observed, however, how the first data transfer from any destination was incurring one DMEM_STALL and LS_DISPATCH_STALL less than the other accesses in a sequence: we were not able to track it back to its source (e.g., effects in the crossbar interconnect or prefetching mechanism) with reasonable confidence. Therefore we cannot exclude it is an (bounded) accuracy problem.

Understanding the latencies incurred by code fetches from the PFlash is complicated by the prefetching and buffering mechanism in place. We empirically observed both the full benefits of the prefetching mechanisms (cache line and PFlash interface), by fetching sequential code, and the worst-case latency, which we triggered by trashing the effect of line prefetching, with jumps from critical word to critical word in the Program Line Buffer. We observed that the prefetching mechanism in the PFlash is highly effective but we were not able to derive general conclusions on its performance. Without more detailed information on how prefetching is implemented and how it interacts with the PMU, breaking down the observed latencies was beyond our reach.

5.7.4 Summary

Overall, despite the difficulties, we managed to develop a suitable multicore contention model compatible with AURIX TC27x automotive platforms, as presented in D4.2. The timing characterization of the platform based on its PMCs has revealed to enable tight contention estimates, as indicated by the preliminary results of its integration in the automotive multicore use case. Those results will be conveniently detailed in Deliverable D6.6. However, we have already reached a number of valuable conclusions:

- Contention bounds are tight (within 10% of the maximum execution time observed). Thus, regardless of whether those bounds can be partially attributed to pessimistic choices, they are tight enough for an efficient use of the hardware platform.
- The ability to obtain those bounds based on an industrially-friendly approach justifies the feasibility of the adoption of multicores for safety-critical real-time systems.
- Proving that software can be run simultaneously in different cores improving single-core throughput, justifies the appropriateness of the adoption of multicores for safety-critical real-time systems.

In summary, the assessment of the hardware support for multicore contention modelling on the AURIX TC27x platform has been very successful despite all difficulties encountered. Hence, the corresponding RTE can reliably build on this PMC analysis to manage and schedule tasks accounting for tight bounds for contention in shared resources.

5.8 Conclusion

In this chapter, we presented a regulation solution based on budgeting that aims at guaranteeing the temporal behavior of high-critical tasks in a mixed critical system, while degrading the behavior of non critical tasks.

The budgeting approach is performed offline and per timeslot. While the approach worked well with purely periodic tasks, high-overhead aperiodic or sporadic tasks caused some major problems: either over-provisioning if considered during the offline characterization phase, causing very low budget for high-critical tasks and, as a consequence, a high rate of low-critical suspend rate; or causing high-critical deadline misses if occurring at runtime during an unprepared timeslot.

In single-core systems, it is common practice to have a dedicated periodic slot to deal with sporadic asynchronous tasks. The applicability of such a practice for multi-core architectures depends on the way the applications are deployed in the system to benefit from parallelism: an option is to parallelize inside applications / partitions, granting all the cores to a single application during each time slot. Such a deployment is compatible with the usual way of dealing with aperiodic tasks. But it also forces to re-write the applications, whose performance will then be constrained by the Amdahl's law [5].

Another option is to run different independent applications, running partitions in parallel. Such a scheme is good for software development that could carry on producing single-threaded applications, which could bring better performance, exploiting the Gustafson's law [17], and not being hampered by data dependency. However, it does not allow anymore to have dedicated timeslots for a particular kind of traffic or tasks, unless costly synchronization is introduced.

As a consequence, before selecting the most adequate control or regulation solution to deal with timing interference on multi-core, a first step should be to consider the possible deployment of the applications, figuring out which kind of parallelism will be exploited. This choice has consequences on the timing interference level, and on the visibility of interference (from white-box in case of intra-partition parallelism as they are coming from well known other tasks of the application, to black-box when coming from a potentially unknown independent application in case of inter-partition parallelism).

In this chapter we have also studied how to extend the RTE to account for thermal issues. While limitations of the hardware platform considered lead to some significant characterization cost, the approach has been proven doable and applicable to industrial use cases.

Analogously, specific automotive multicore architectures (AURIX TC275 processor) have also been considered for the control of contention in mixed critical systems. Our analysis reveals that, despite the intrinsic difficulties posed by this platform, multicore contention can be properly modelled and accounted for, so this solution is also integrated in the industrial use cases of the project.

Chapter 6 WP4 Time-Critical Prototype

As presented in Deliverable D4.1, the time-critical avionic prototype used in the context of WP4 encompasses high-critical and low-critical applications as well as the Budget-Based RunTime Engine (BB-RTE). Figure 6.1 details the updated target environment for this WP4 prototype using the ARM Juno board target.

Mixed-critical tasks as well as the RTE will run as single or multiple partitions applications or threads on top of the PikeOS operating system. We developed a specific driver for PikeOS to provide privileged access to the PMC hardware counters, and the whole collection infrastructure is based on METrICS, that is described in Deliverable D4.2.

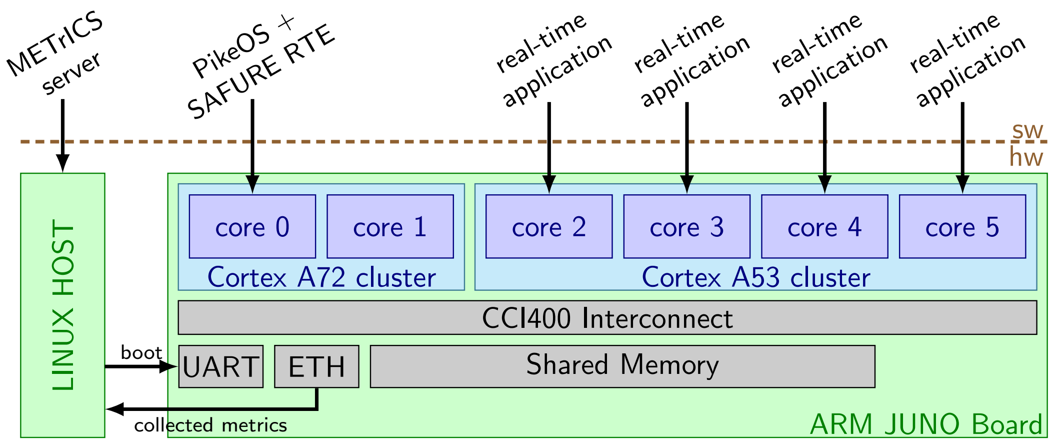


Figure 6.1: Target environment for the Time-Critical prototype

The tested time-critical applications are further presented in the next subsections, and are run on the more time-predictive A53 cores. Both PikeOS and the RTE are running on the first high-performance A72 core. We also tested running the SAFURE RTE a dedicated core (core 1), but it has proven to produce more interferences due to the high number of system calls performed by the runtime engine.

6.1 Hard Real Time High-critical Application: Flight Management System

The selected hard real-time high-critical application for the WP4 time-critical prototype is a mark-up FMS application from the avionics domain. The purpose of the FMS in modern avionics is to provide the crew with centralized control for the aircraft navigation sensors, computer based flight planning, fuel management, radio navigation management,and geographical situation information. Taking charge of a wide variety of in-flight tasks, the FMS allows us to reduce the workload of the flight crew allowing us to reduce crew size.

The FMS is especially responsible for services that allow in-flight guidance of the plane. From pre-set flightplans (take-off airport to landing airport), the FMS is responsible for plane localization, trajectory computation allowing the plane to follow the flightplan, and reaction to pilot directives.

6.1.1 Software Architecture

The FMS application is constituted by 25 time-critical tasks that are regrouped into different task groups as presented in Figure 6.2.

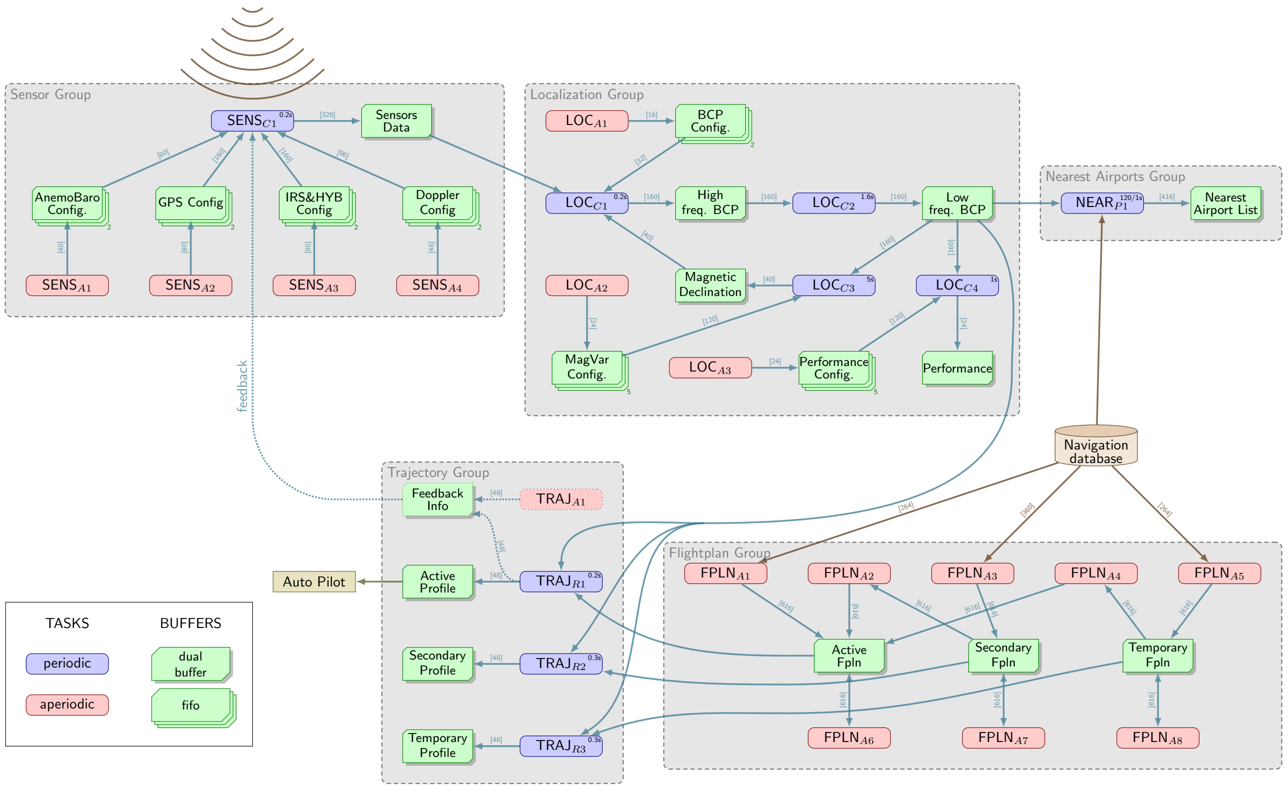


Figure 6.2: Software architecture of the Flight Management System (FMS)

The **Sensors task group** is in charge of generating all the localization data from various sensors (Anemo-barometric sensors, IRS (Pure Inertia Reference System), GPS (Global Positioning System), HYB (Hybrid Inertia Reference System), Doppler sensor).

The **Localization task group** is in charge of analysing outputs of sensors to generate the most probable position of the aircraft (BCP). This localization data is composed of: Position (latitude, longitude, and altitude), Attitude (Pitch, Roll and Yaw angular rates), Velocity (Ground speed and Vertical Speed), Acceleration (lateral and longitudinal), and Wind related data (speed and angle).

Note that a single sensor may not provide the full Localization information. The Doppler sensor for instance does not provide any position related information such as longitude and latitude. It however provides very accurate velocity (speed related) information. The role of the Localization task group is therefore to merge information from the sensors with different trustworthiness levels.

The purpose of the **Nearest Airports task group** is to continually build a list of the nearest airports, during the flight. This information is useful in case the pilot decides to have an impromptu landing for some reason. The tasks from this task group do not participate directly in flight management, and the computed output only has to be sent to the display.

The **Flightplan task group** is in charge of managing and processing modification requests on the flightplans that are pre-set routes used to guide the airplane. Three different flightplans coexist concurrently on the system:

- The **active flightplan** is the flightplan currently used to guide the aircraft.

- The **secondary flightplan** is an alternative route toward the destination. It could consider for instance an alternative-landing runway on the destination airport, which has a significant impact on the target airport approach procedure.
- The **temporary flightplan** is an intermediate flightplan allowing the crew to enter a new flightplan and check for the modification before applying.

The flightplan task group is only composed of aperiodic tasks that correspond to the pilots modifications to the pre-set flightplans.

The **Trajectory task group** aims at computing both lateral and vertical profiles for the three flightplans set by the flightplan task. The lateral profile is composed of waypoints as well as leg information (path before, after and between the waypoints). The vertical profile provides altitude information (cruise altitude interceptions, crossing altitudes and slope angles) as well as performance information (estimated time of arrival, estimated fuel on board).

Trajectory computation is performed for each of the three above defined flightplans. The inputs of trajectory computation are both the flightplan and the best computed position (BCP) of the plane that comes from the localization task group. The computed trajectory tries to tangent the pre-set flightplan while respecting passenger wellness (limiting roll and pitch) as well as physical limitation of the plane actuators such as flaps. The trajectory information is later used by the plane autopilot to actually interact with these actuators.

The FMS application also embeds a large Navigation Database that does not fit in any cache structure. It is both linearly and regularly accessed by task from the Nearest airport task group, as well as randomly and sporadically accessed by tasks of the Flightplan task group. Accesses to this database in the main memory is very interference prone.

6.1.2 Safety and Timing Requirements

All the tasks composing the FMS have stringent real-time requirements: Table 6.1 and Table 6.2 respectively show the time requirements of periodic and aperiodic tasks composing the application.

Periodic Task	Period / Deadline
SENSC1	200ms
LOCC1	200ms
LOCC2	1.6s
LOCC3	5s
LOCC4	1s
TRAJR1	200ms
TRAJR2	300ms
TRAJR3	300ms
NEARP1	1s

Table 6.1: FMS: Time requirements of periodic tasks

In the FMS, periodic tasks are characterized by an activation period as well as a deadline that always corresponds to the next activation period.

All aperiodic tasks are sporadic, and are characterized by a maximum number of activation per period of time. This period of time is usually defined by the period of the periodic task consuming the data produced by the aperiodic task. Aperiodic tasks also have to respect a deadline provided by Table 6.2.

Aperiodic Task	Maximum activations	Deadline
SENSA1	2 per 200ms	50ms
SENSA2	2 per 200ms	50ms
SENSA3	2 per 200ms	50ms
SENSA4	2 per 200ms	50ms
LOCA1	2 per 200ms	100ms
LOCA2	5 per 5s	50ms
LOCA3	5 per 1s	50ms
FPLNA1	once at initialization	1s
FPLNA2	1 per 1s	1s
FPLNA3	once at initialization	1s
FPLNA4	1 per 1s	1s
FPLNA5	1 per 1s	1s
FPLNA6	1 per 1s	50ms
FPLNA7	1 per 1s	50ms
FPLNA8	1 per 1s	50ms
TRAJA1	once at initialization	50ms

Table 6.2: FMS: Time requirements of aperiodic tasks

6.2 Soft Real Time Low-critical Application: Bi-Quadratic Distributed Control System

The selected soft low-critical real-time application for the WP4 time-critical prototype is a control-command application implementing a BiQuad.

6.2.1 Software Architecture

The tasks composing this applications, appearing in Figure 6.3, are:

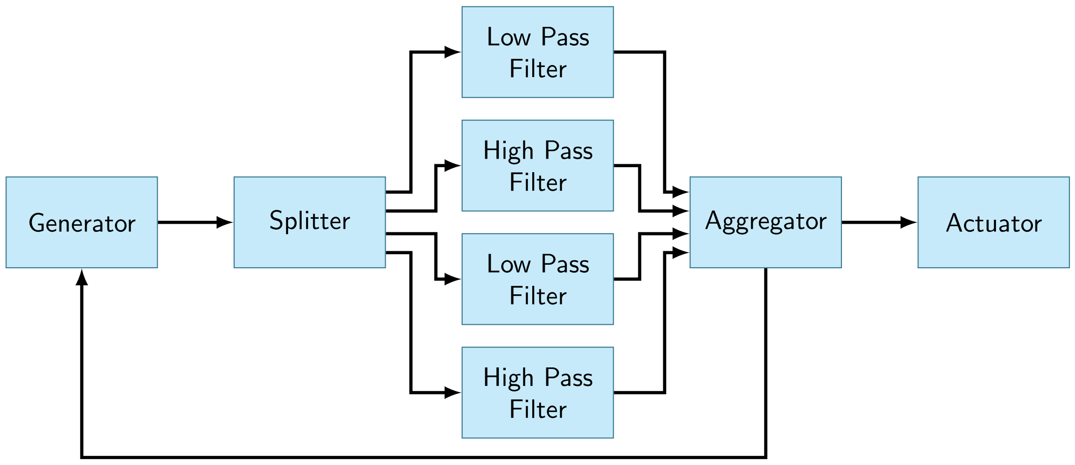


Figure 6.3: Partitioned BiQuad application implemented in PikeOS

- The Generator process is generating the input data. It either self-generates the data on the first pass, or iterates on the received data on the next passes.
- The Splitter process splits the data received from a FIFO to make it globally available to all the filtering tasks.
- The LoPass and the HiPass processes are applying some bi quadratic filtering to the data.

- The Aggregator process fuses the previously computed filtered data and sends it back as feedback to the generator task.
- The Display process finally displays the fused data.

Figure 6.3 illustrates the multi-partition PikeOS version of this application. It is composed of 3 different partitions and 6 PikeOS threads implementing the different tasks. This example exhibits the use of the possible communication mediums available in PikeOS, both for intra/inter-partition communication and for performing heavy load floating-point computation.

6.2.2 Safety and Timing Requirements

The tasks composing the BiQuad application have the real-time requirements presented in Table 6.3. All these requirements are soft real-time requirements dure to the required data throughput.

Periodic Task	Partition	Period	Deadline
Generator	1	100ms	100ms
Splitter	2	100ms	10ms
Lo-Pass Filter	2	100ms	20ms
hi-Pass Filter	2	100ms	20ms
Notch Filter	2	100ms	20ms
Band-Pass Filter	2	100ms	20ms
Aggregator	2	100ms	10ms
Display	3	100ms	100ms

Table 6.3: FMS: Time requirements of BiQuad tasks

The SAFURE runtime engine is expected to degrade the timing behaviour of the BiQuad application to ensure correct execution of the critical application, implying some of these deadlines will not be respected.

6.3 Multi-Core Low-critical Application: Drone Fleet Guidance

The third application composing the WP4 prototype is embedded Directed Rotodrone Operated Network (eDRON), an application dedicated at guiding a fleet composed of four quadricopter drones along a preset flight route. The purpose of this application is to mimic a representative behaviour of an avionic application, while exercising classical ARINC-653 communication services and proposing several multi-core deployment options.

6.3.1 Software Architecture

The software architecture of eDRON is presented in Figure 6.4. This application is composed of six ARINC-653 partitions. The first one sets up the preset flight route for all the drones, the last one displays all the drone positions, and each of the four remaining partitions is dedicated to pilot a particular drone. These later partitions are composed of 7 tasks, with most of the computation being performed in the four engine control tasks, each being dedicated to control the velocity for one of the four engine of a drone, so that it follows the preset route.

6.3.2 Safety and Timing Requirements

The tasks composing the eDRON application have the real-time requirements presented in Table 6.4. As per the BiQuad application, we consider these requirements to be soft real-time ones, even though deadline misses have an impact on the trajectory of each drone, with regards to the preset flightplan.

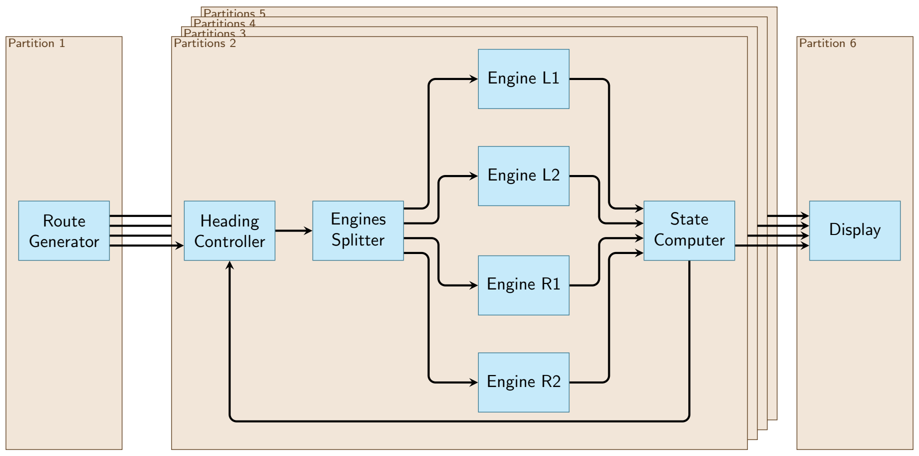


Figure 6.4: Software architecture of the eDRON application

Periodic Task	Period	Deadline
Route Generator	200ms	20ms
Heading Controller	200ms	15ms
Engine Splitter	200ms	10ms
Engine	25ms	25ms
State Computer	200ms	25ms
Display	200ms	30ms

Table 6.4: eDRON time requirements

6.3.3 Mapping and Timing Interference

When mapping such an application on a multi-core processor, we first need to decide what will run in parallel. The application offers two obvious parallelization schemes: **inter-partition** parallelism where each core will deal with a single drone, running the velocity control tasks sequentially for each drone; and the **intra-partition** parallelism where each core will focus on one particular engine, dealing with each drone sequentially and running all sequential tasks on core 0.

Some other parallelization options are available: for example parallelizing along the pipeline, or performing loop-level parallelization of the tasks, but those are beyond the scope of this project as they require deeper modifications of the application.

These two deployments have advantages and drawbacks: The Amdahl's law [5] may limit the performance of the intra-partition version, while the inter-partition version will benefit from the Gustafson's law [17], running independent applications in parallel. However, with regards to timing interference, the intra-partition version offers a white-box context where the partition scheduling can limit the level of interference between known tasks. The inter-partition parallelism on the other hand corresponds to a black-box context where no easy control is possible to limit the interference level of another independent application.

Chapter 7 Freedom from Interferences for mixed-critical systems

Freedom from Interferences OS-extension, implementation details
For SAFURE, the Automotive Multicore Use Case prototype guarantees at firmware level the freedom from interferences compliant with ISO 26262[21]. According to the SAFURE deliverable D4.1 (chapter 6.2), MAG has developed two firmware drivers (AUTOSAR like) to implement timing protection and memory protection to guarantee freedom from interferences between two applications that run on two different cores. This is an optimized alternative to RealTime OS support.

7.1 Memory Protection OS-extension

MAG has developed a firmware (FW) driver to support memory protection at Task level, to guarantee that data structures inside the protected Task (or part of it) cannot be accessed by untrusted operations. This implementation makes use of microcontroller MPU (Memory Protection Unit) device. This mechanism is ISO 26262 compliant and AUTOSAR-like (see D4.1 chapter - 6.3.2).

Here is the Memory Protection concept described in AUTOSAR figure 7.1:

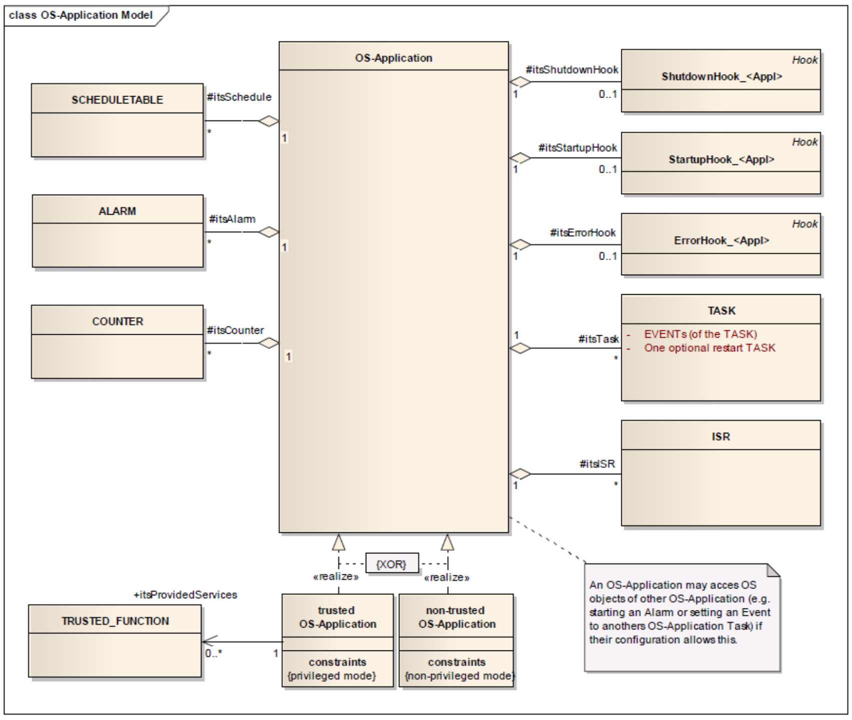

class OS-Application Model
OS-Application
SCHEDULETABLE
#itsSchedule
ALARM
#itsAlarm
COUNTER
#itsCounter
#itsShutdownHook
Hook
ShutdownHook_<Appl>
#itsStartupHook
StartupHook_<Appl>
#itsErrorHook
ErrorHook_<Appl>
0..1
TASK
- EVENTs (of the TASK)
- One optional restart TASK
#itsTask
ISR
#itsISR
{XOR}
«realize»
trusted OS-Application
constraints {privileged mode}
non-trusted OS-Application
constraints {non-privileged mode}
+itsProvidedServices
TRUSTED_FUNCTION
0..*
An OS-Application may acces OS objects of other OS-Application (e.g. starting an Alarm or setting an Event to anothers OS-Application Task) if their configuration allows this.

Figure 7.1: Memory protection applied at OS Application level

7.1.1 Implementation and test details

7.1.1.1 APIs description

The APIs implemented are the following:

- `void MPU_Init (<Pwd>)`
- `t_MPU_Partition <Pid> MPU_GetCurrentPID (<Pwd>)`
- `void MPU_SetCurrentPID (t_MPU_Partition <Pid>, <Pwd>)`
- `void MPU_RegCheck (void)`
- `void MPU_EX_ProtectionError (<ErrorAddress>, <Pid>)`

Where the parameters:

- **Pwd**: represents the password to avoid uncontrolled calls to this function
- **t_MPU_Partition Pid** (out parameter): is the partition ID.
- **ErrorAddress**: represents the address of instruction that has generated the Protection Error.
- **Pid**: is the partition where the error has been generated.

More in details, these routines are implemented to cover and achieve the following purposes:

- **MPU_Init**: It is important to initialize the driver to a default state at system startup and on the occurrence of an adverse event on the system. For this reason, an initialization routine must be provided. Initialization must be used to configure the common registers.
- **MPU_GetCurrentPID**: This purpose is achieved by MPU driver allowing the definition of Software Partitions. Software partitioning allows the co-existence of software partitions that use the same resources. It allows software components to be free from interference from other software components. Moreover, it allows changes to be made to one software partition without the need to re-verify the unmodified software partitions.
- **MPU_SetCurrentPID**: same description of MPU_GetCurrentPID.
- **MPU_RegCheck**: This function checks the flow control of execution. Check data protection of all the registers used during the initialization using CRC check and comparison of stored values with current values of registers.
- **MPU_EX_ProtectionError**: This exception must be called when a Protection Error has been raised.

7.1.1.2 Tests

For each API, respectively, the following specifications of tests have been carried out:

- **MPU_Init**: It is verified that in the initialization of the MPU driver, the partitions defined in static configuration are configured correctly and the MPU driver is enabled with access right of the partitions set to R/W/X for both SV and User Defined mode. The current Partition ID of the microcontroller is set to MPU_DEFAULT_PID.
- **MPU_GetCurrentPID**: It is verified that the driver is able to return the current active Partition ID and there is a password protection mechanism to avoid uncontrolled call to get the active Partition ID.

- **MPU_SetCurrentPID**: It is verified that the new software partition can be set to become the current active partition and there is a password protection mechanism to avoid uncontrolled call to switching of software partition.
- **MPU_RegCheck**: It is verified the capability to report error in case of data corruption in registers used in the initialization procedure.
- **MPU_EX_ProtectionError**: It is verified that the exception is called when a Protection Error has been raised by the MPU driver as a consequence of an unauthorized access.

From these specifications, the test implementation procedures have been derived and the results have been successfully achieved and checked.

7.2 Timing Isolation OS-extension

TPROT module must be a generic driver to realize Timing protection without any support from the Real Time OS.
TPROT enables us to monitor:

- the execution time of Task and ISR routines;
- the time spent by a higher priority task waiting for the resource occupied by a lower priority task;
- the periodicity of task.

The temporal protection system is used to guard against run-time over run. It covers three functionalities:

1. TASK/ISR Execution Time Monitoring.
2. TASK/ISR Lock time Monitoring.
3. TASK/ISR Time period Monitoring.

TPROT introduces the concept of TimerSet, which is a logical id bound to one or more microcontroller timing resources. The number of microcontroller resources needed may vary depending on the specific microcontroller model. TimerSet will also encapsulate all the timing protection budgets and measures. One task should always use the same TimerSet. Different tasks should always use different TimerSet.
Each TimerSet object must have three methods to configure, start and reset the measure.
Each functionality can be implemented by changing the use case, using the generic function of the TPROT driver.

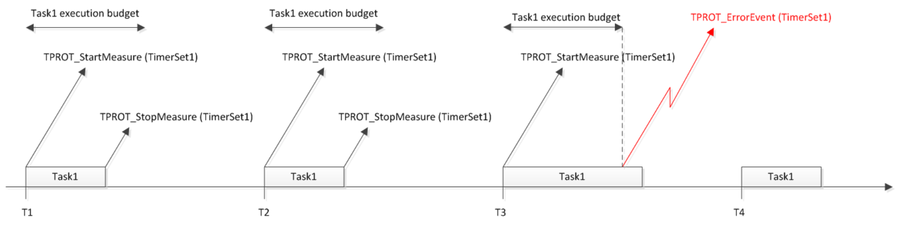

Task1 execution budget
Task1 execution budget
Task1 execution budget
TPROT_ErrorEvent (TimerSet1)
TPROT_StartMeasure (TimerSet1)
TPROT_StartMeasure (TimerSet1)
TPROT_StartMeasure (TimerSet1)
TPROT_StopMeasure (TimerSet1)
TPROT_StopMeasure (TimerSet1)
Task1
Task1
Task1
Task1
T1
T2
T3
T4

Figure 7.2: Execution Budget Monitoring use case

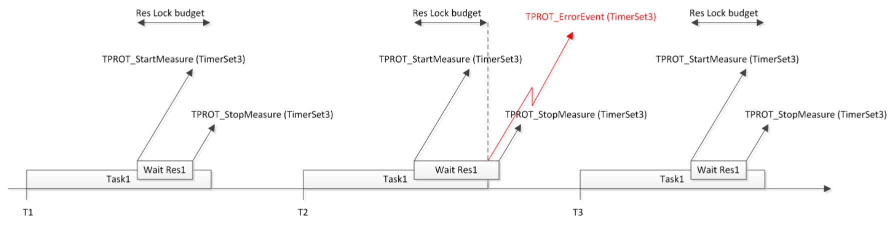

Res Lock budget
Res Lock budget
TPROT_ErrorEvent (TimerSet3)
Res Lock budget
TPROT_StartMeasure (TimerSet3)
TPROT_StartMeasure (TimerSet3)
TPROT_StartMeasure (TimerSet3)
TPROT_StopMeasure (TimerSet3)
TPROT_StopMeasure (TimerSet3)
TPROT_StopMeasure (TimerSet3)
Wait Res1
Wait Res1
Wait Res1
Task1
Task1
Task1
T1
T2
T3

Figure 7.3: Time Period Monitoring use case

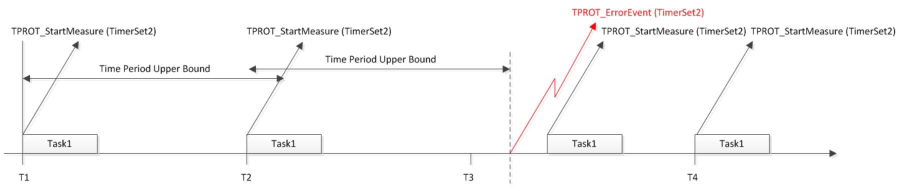

TPROT_ErrorEvent (TimerSet2)
TPROT_StartMeasure (TimerSet2)
TPROT_StartMeasure (TimerSet2)
TPROT_StartMeasure (TimerSet2)
TPROT_StartMeasure (TimerSet2)
Time Period Upper Bound
Time Period Upper Bound
Task1
Task1
Task1
Task1
T1
T2
T3
T4

Figure 7.4: Lock Time Monitoring

7.2.1 Implementation and tests details

This driver is composed of three different layers:

- INTERFACE: Its composed by types and methods exported to upper layers. The signature and the functional description of ALL the methods described in this specification refer to interface layer. Furthermore, at implementation level, the functional part of each method is coded in the kernel part.
- KERNEL: Its composed by internal types and methods, not visible to upper layers.
- STATIC CONFIGURATION: Its composed by channel database definition for the driver and other parameterizations.

In order to have a list of symbolic identifiers used by device driver for referring a physical signal and used by BIOS driver for understanding which channel must be addressed, scm_tprot_drv.h file must define the type t_IDN_tprot.
This type collects in an enumerative type all the signals present in the system and that can be addressed using a generic driver.
IDN channels of the same driver must be contiguous. At the end must be defined a tag (with the syntax ⟨driver name ⟩_MAX) to be used to calculate the number of IDNs related to a specific driver.
Bear in mind, the order in which IDN TPROT driver channel must be exactly the same in which they are declared into scm_tprot_drv.h file where IDN identifiers are associated to the internal hardware channels.
The enum is defined in the following way:

```
typedef enum {
        <IDN_TimerSet_0>,
        <IDN_TimerSet_1>,
        <IDN_TimerSet_2>,
        ...,
        <IDN_TimerSet_n>,
        TPROT_MAX
} t_IDN_tprot;
```

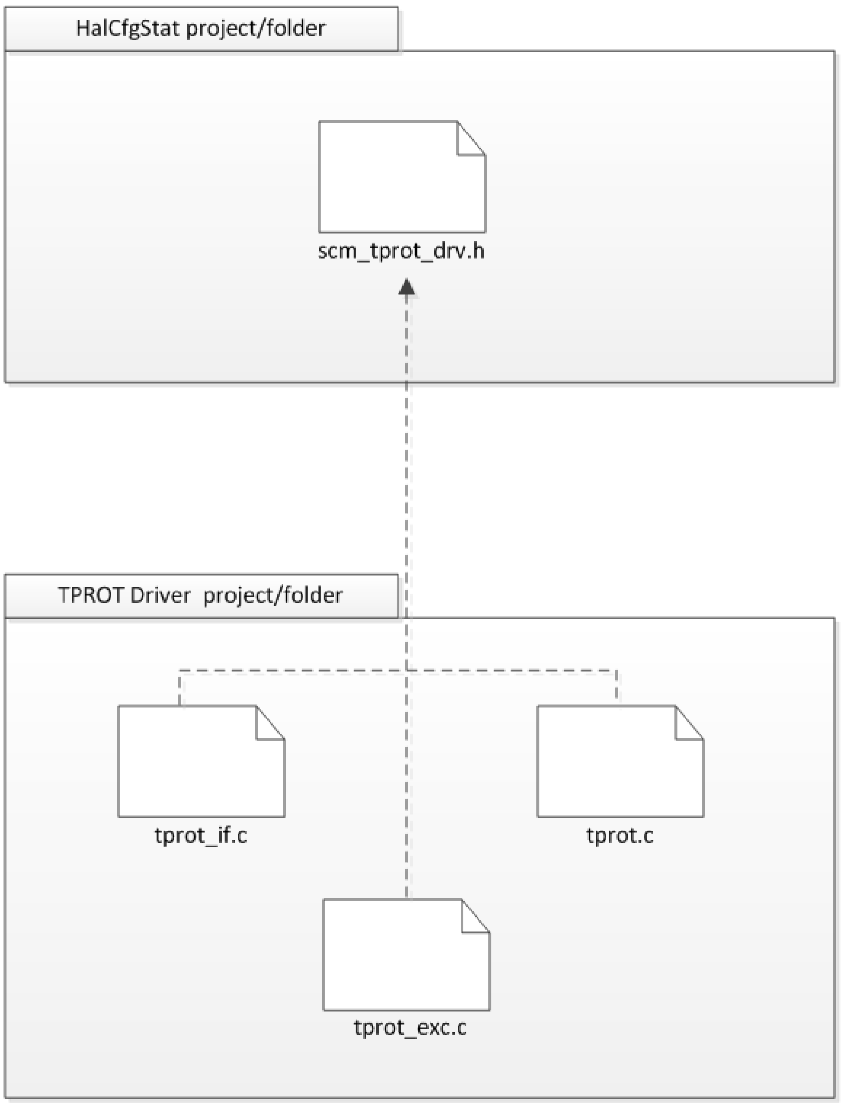


Figure 7.5: Layer structure

In order to define generically all the possible TPROT physical channels that could be associated to the external output frequential signals, the upper level module must foresee all the possible TPROT identifiers.
Only the identifier used must be really defined, but all the possible identifiers must be foreseen. Unused identifiers must be commented, as described below:

```
TPROT0_IDN
TPROT1_IDN
TPROT2_IDN
...
TPROTn_IDN
```

7.2.1.1 Driver enabling

The information of the presence of the TPROT drive into the BIOS is declared into the file scm_en_drv.h. If the TPROT driver is present, following define is enable, otherwise TPROT driver isn't present:

```
# define TPROT_IS_PRESENT
```

7.2.1.2 APIs description

The APIs implemented are the following:

- `t_MStatus  TPROT_Config (<DynCfg>)`
- `t_MStatus  TPROT_StartMeasure (<Idn>, <Timeout>)`
- `t_MStatus  TPROT_StopMeasure (<Idn>, <Measure>)`

Where the parameters:

- **t_MStatus** can assume the value of:
  - MS_OK is returned if the method has been executed successfully.
  - MS_PAR_NOT_OK is returned if the input parameters are wrong.
  - MS_NOT_OK is returned if the method has been called before the TPROT_Config method.
- **DynCfg** is the pointer to the structure used to dynamically configure the Timer and event resource of the microcontroller. It could have the following fields:
  - *idn*: is the TPROT channel identifier defined in the static configuration in the type definition t_IDN_tprot;
  - *excPtr*: is the callback to be call in case of Timing protection Error. This function must be defined in the upper layer (type is: void *);
  - *excObj*: is the object pointer to be passed as a parameter of the callback Timing protection error (tipe is: void *).
- **Idn**: represents the TPROT channel identifier defined in the static configuration in the type definition t_IDN_tprot.
- **Timeout**: is the timeout to be measured before calling the timing protection error.
- **Measure** (out parameter): is the time measured from the last called TPROT_StartMeasure and the current TPROT_StopMeasure.

More in details, these routines are implemented to cover and achieve the following purposes:

- **TPROT_Config**:
  1. initializes all the global variables and registers used for the timing protection functionality for the channel identified by *idn* field of pointer *DynCfg*;
  2. configures the HW mapped on the timing protection functionality of the channel *idn* to be ready to start the measure;
  3. configures the exception handler with the passed exception pointer and object pointer to be called in case of Timing protection error, this event must generate a no-maskable trap or maskable interrupt if HW support is not available;
  4. stops every ongoing activity on the channel *idn* and re-configure it to be ready to start again the functionality.

  If the field *excPtr* contains a null address, the method must link an internal function and only the measuring functionality must be implemented.
  Measures and event detection must be activated only after called the TPROT_StartMeasure.
  No pending activity must be tolerated after the config method has been called.
- **TPROT_StartMeasure**:

1. starts the measure of the time related to the timeSet **Idn**;
2. programs the callback event in case the **Timeout** elapses.

Timeout event and time measure must be restarted if the method is called before stopping the previous measure. This sequence is useful in the use case described in the Figure 7.4.

- **TPROT_StopMeasure**:

1. stops the measure of the time related to the timeSet **Idn**;
2. clears the pending status of the timeout to avoid protection error event;
3. return the measure of the time elapsed from the last TPROT_StartMeasure and the current TPROT_StopMeasure.

All the activities related to the TimeSet **Idn** must be cleared, after to call the stop measure method.

7.2.1.3 Test

The firmware driver TPROT is still in development phase and also the test specifications are under construction.

Chapter 8 AUTOSAR OS

8.1 RTE Generator

In the AUTOSAR workflow, the scheduling and communication implementation is generated automatically based on the specifications provided in the model for the application components.

When these specifications include the indication of a criticality level, as indicated by the modeling extensions presented in the Deliverable D2.2 of WP2, the generation of the RTE task implementation and scheduling instructions should include also the calls to the OS functions and configuration features that can be used to enforce timing isolation and protection.

In AUTOSAR, these OS API functions and OS configuration features allow to protect application tasks against excessive execution of higher priority tasks or when accessing resources (to protect against excessive blocking time) and to protect against an excessive use of time by ISR handlers.

8.1.1 Mechanisms for timing protection

The main AUTOSAR mechanisms to prevent timing interference in AUTOSAR are:

- AUTOSAR OS guarantees a statically configured upper bound, called the Execution Budget, on the execution time of tasks and category 2 ISRs
- AUTOSAR OS prevents guarantees a statically configured upper bound, called the Lock Budget, on the time that resources are held by tasks or Category 2 ISRs
- AUTOSAR OS enforces an inter-arrival time protection to guarantee a statically configured lower bound, called the Time Frame, on the time between a task being permitted to transition into the READY state due to to activation or release

Of those, the Execution budget and Lock budget protections are of interest for our work.

8.1.2 Overview of the RTE generator

This work adopted a generator for the AUTOSAR Run-Time Environment (RTE) developed by Evidence SRL in the context of the EU Hercules project, which is conceived to work for the open source ERIKA Enterprise real-time operating system. ERIKA is OSEK-compliant and de-facto representative of the typical behavior of an AUTOSAR operating system.

The RTE generator has been realized by extending the AUTOSAR Tool Platform (Artop), which is a free-of-charge implementation of common base functionality for AUTOSAR development tools. Artop is based on the Eclipse development environment. The RTE generator consists of Eclipse Java packages loaded in Artop that make use of a set of queries developed in the Acceleo template language and transforming the AUTOSAR models into code. The generator inputs an AUTOSAR `arxml` model and generates:

- the code of the tasks that calls the corresponding runnables;
- a set of functions to work with sender-receiver AUTOSAR ports between software components and the corresponding data structures;

- the configuration of the operating system(OIL file); and
- other accessory code and C headers.

8.1.3 Objectives

The main objective of this work is to extend the RTE generator described in the previous section to:

- handle the AUTOSAR modeling extensions related to mixed-criticality systems proposed in the SAFURE project; and
- support the timing protection mechanisms available in AUTOSAR-compliant real-time operating systems, which are configured as a function of the model information.

To prevent timing errors, AUTOSAR OSes employ execution time protection to guarantee a statically-configured upper bound, called *Execution Budget*, on the execution time of tasks. To preserve the different criticality levels of tasks, high-criticality tasks must not be affected by overruns (i.e., timing faults) experienced by high-priority tasks. To this end, for each task τ_i, the RTE has to verify whether there exists *at least* one high criticality task with a lower (or equal[1]) priority with respect to τ_i: if this task does exist, the timing protection mechanism is configured for τ_i.

Formally, let $lep(\tau_i)$ be set of tasks with lower or equal priority with respect to task τ_i, and let $c_i \in \{0, 1\}$ be the criticality level of task τ_i, where $c_i = 1$ means high criticality. Task τ_i must be subject to timing protection if

$$\exists \tau_j \in lep(\tau_i) \mid c_j > 0.$$

The timing protection is configured by setting the Execution Budget of τ_i equal to the sum of all the execution time bounds provided in the AUTOSAR model for the runnables that compose τ_i. According to the AUTOSAR OS specification, it is possible to setup a callback that is invoked whenever a task exceeds its Execution Budget. The RTE generator is also in charge of automatically generating a skeleton code for this callback as part of the RTE code. Finally, the RTE generator logs eventual errors in generating the configuration of the timing protection mechanism. These aspects are discussed in Sections 8.1.5, 8.1.6, and 8.1.7, after discussing in the next section the queries to the AUTOSAR model that are necessary to extract the required information.

8.1.4 Queries to the AUTOSAR model

Two main Acceleo queries have been developed to configure timing protection:

- `isSubjectToBudgeting`: returns TRUE if a task must be subject to timing protection, FALSE otherwise;
- `getWCET`: returns the sum of all the execution time bounds provided in the AUTOSAR model for the runnables that compose a task.

The queries are respectively discussed in the next two subsections.

8.1.4.1 The isSubjectToBudgeting query

For a given task τ_i, the `isSubjectToBudgeting` query has been implemented by exploring all tasks in the system and collecting only those in the set $lep(\tau_i)$ with a positive criticality level. In AUTOSAR models, the tasks are available in the `EcuConfiguration` AUTOSAR package within the `OS` child (that is an entity of type `EcuModuleConfigurationValues`). Task priorities and criticality levels are modeled as childs of type `EcucNumericalParamValue` for each task entity, which must

[1]Note that tasks with the same priority handled in first-in-first-out order can always interfere each other.

include a `NumericalValueVariationPoint` entity with the corresponding numerical value. Each `EcucNumericalParamValue` entity has a specific AUTOSAR definition string:

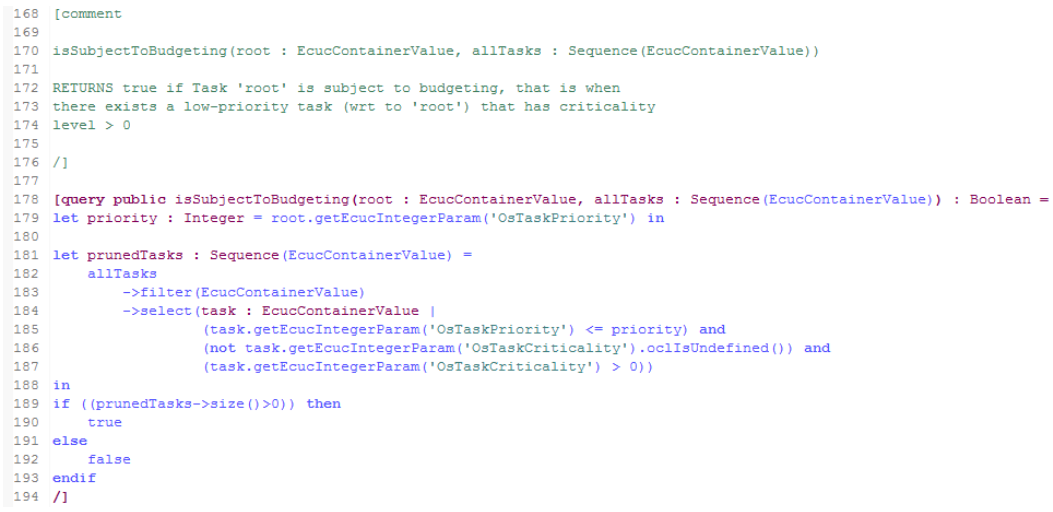
`/AUTOSAR/EcucDefs/Os/OsTask/OsTaskPriority`,

for task priorities, and

`/AUTOSAR/EcucDefs/Os/OsTask/OsTaskCriticality`,

for task criticalities.

The code snippet of the `isSubjectToBudgeting` query is reported in Figure 8.1. As it can be observed from the figure, the query uses a `select` statement to collect in `prunedTasks` the set of tasks $\tau_j \in lep(\tau_i) \mid c_j > 0$. Then, if `prunedTasks` is not empty, TRUE is returned.



```
[comment

isSubjectToBudgeting(root : EcucContainerValue, allTasks : Sequence(EcucContainerValue))

RETURNS true if Task 'root' is subject to budgeting, that is when
there exists a low-priority task (wrt to 'root') that has criticality
level > 0

/]

[query public isSubjectToBudgeting(root : EcucContainerValue, allTasks : Sequence(EcucContainerValue)) : Boolean =
let priority : Integer = root.getEcucIntegerParam('OsTaskPriority') in

let prunedTasks : Sequence(EcucContainerValue) =
    allTasks
        ->filter(EcucContainerValue)
        ->select(task : EcucContainerValue |
                  (task.getEcucIntegerParam('OsTaskPriority') <= priority) and
                  (not task.getEcucIntegerParam('OsTaskCriticality').oclIsUndefined()) and
                  (task.getEcucIntegerParam('OsTaskCriticality') > 0))
in
if ((prunedTasks->size()>0)) then
    true
else
    false
endif
/]
```

Figure 8.1: Code snippet for the `isSubjectToBudgeting` query.

8.1.4.2 The getWCET query

The `getWCET` query has to identify all the runnables associated to a task. Typically, the connection between tasks and runnables is defined in AUTOSAR models by mapping the RTEevent associated with the activation of each runnable to the task. As a result, several modeling entities must be traversed to list the runnables associated to a task.

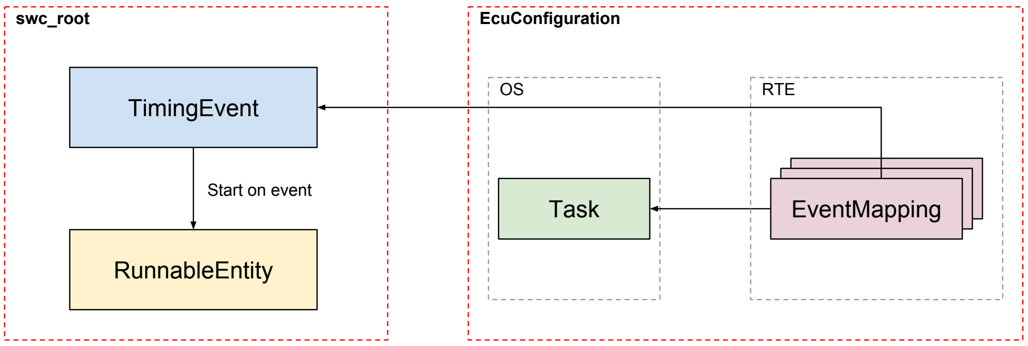


Figure 8.2: Path to identify the runnables associated to a task in AUTOSAR.

As it is illustrated in Figure 8.2, runnables are associated to *timing events* (e.g., a periodic stimuli), where the latter are assigned to tasks via *event mappings*. Each event mapping maps one task to a single timing event. A task can be referenced in multiple event mappings, hence it can manage more than one timing event.

Consequently, the following steps are required to identify all the runnables associated to a task τ_i:

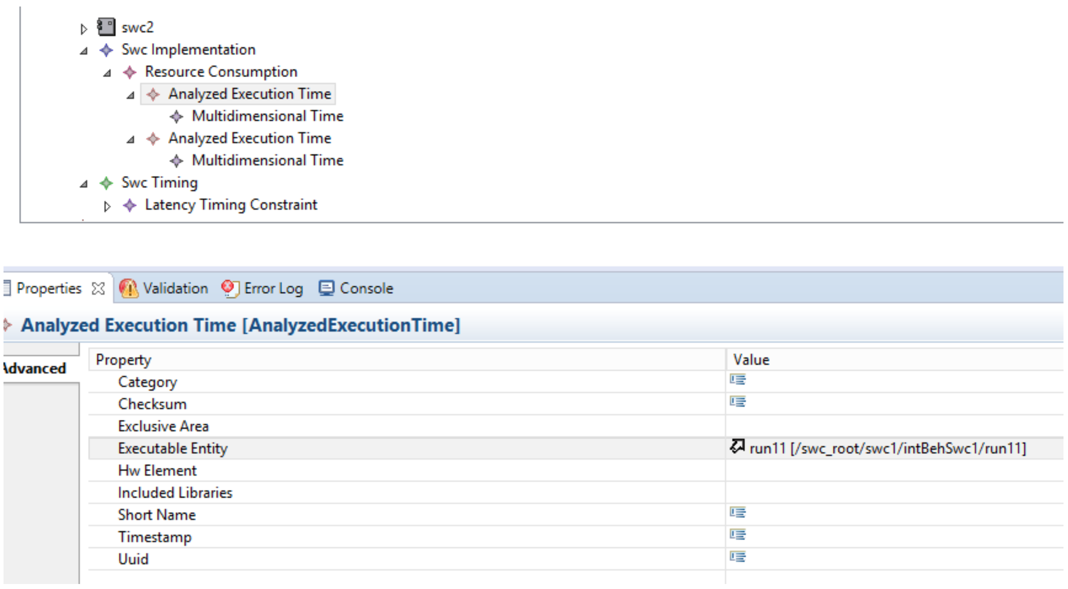
1. Collect the set $EM(\tau_i)$ of event mappings related to τ_i;
2. Identify the set $TE(\tau_i)$ of timing events related to any event mapping in $EM(\tau_i)$;
3. Construct the set $R(\tau_i)$ of runnables associated to the timing events $TE(\tau_i)$.

Event mappings can be identified with the AUTOSAR definition string

`/AUTOSAR/EcucDefs/Rte/RteSwComponentInstance/RteEventToTaskMapping`.

Within each event mapping, the related task is identified by the `RteMappedToTaskRef` child in the model tree, while the corresponding timing event is identified by the `RteEventRef` child. Finally, for each timing event, the corresponding runnable is indicated in the `startOnEvent` field.

Once these steps are performed, the `getWCET` query can simply return the sum of the execution time bounds of each runnable in $R(\tau_i)$. These bounds are not directly available as fields or children of each runnable entity. Rather, they are available in an `SwcImplementation` child of the `swc_root` AUTOSAR package. Specifically, an `SwcImplementation` entity must include a child of type `ResourceConsumption`, which in turn must include another child of type `AnalyzedExecutionTime`, which – again – must include another child of type `MultidimensionalTime`. The field `Cse Code` of the latter contains the execution time bound. The corresponding runnable is provided in the field `Executable Entity` of the `AnalyzedExecutionTime` child. This tree of entities is illustrated in Figure 8.3.


swc2
Swc Implementation
Resource Consumption
Analyzed Execution Time
Multidimensional Time
Analyzed Execution Time
Multidimensional Time
Swc Timing
Latency Timing Constraint

Properties | Validation | Error Log | Console

Analyzed Execution Time [AnalyzedExecutionTime]

Advanced

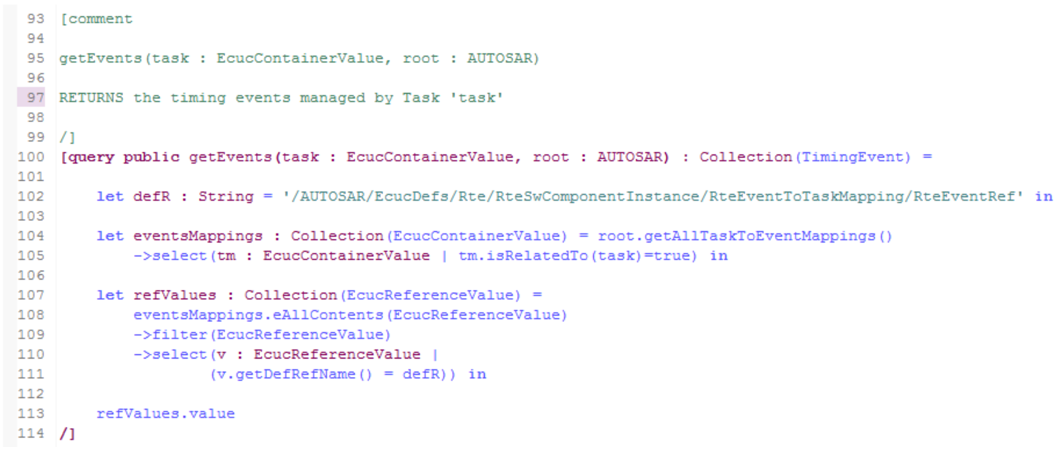
Property	Value
Category	
Checksum	
Exclusive Area	
Executable Entity	run11 [/swc_root/swc1/intBehSwc1/run11]
Hw Element	
Included Libraries	
Short Name	
Timestamp	
Uuid	

Figure 8.3: Tree of entities to model execution time bounds of runnables.

All the above steps have been implemented by decomposing the problem into multiple sub-queries. As an example, Figure 8.4 reports an example of one of such sub-queries, specifically the one that identifies all the timing events related to a task. Note that this query builds upon another sub-query named `isRelatedTo` that verifies whether an event mapping is related to a task.

8.1.5 Timing protection in the OS configuration

The two queries `isSubjectToBudgeting` and `getWCET` are used to configure the timing protection mechanism of the operating system. At the stage of the generation of the OSEK Implementation



```
[comment

getEvents(task : EcucContainerValue, root : AUTOSAR)

RETURNS the timing events managed by Task 'task'

/]
[query public getEvents(task : EcucContainerValue, root : AUTOSAR) : Collection(TimingEvent) =

    let defR : String = '/AUTOSAR/EcucDefs/Rte/RteSwComponentInstance/RteEventToTaskMapping/RteEventRef' in

    let eventsMappings : Collection(EcucContainerValue) = root.getAllTaskToEventMappings()
        ->select(tm : EcucContainerValue | tm.isRelatedTo(task)=true) in

    let refValues : Collection(EcucReferenceValue) =
        eventsMappings.eAllContents(EcucReferenceValue)
        ->filter(EcucReferenceValue)
        ->select(v : EcucReferenceValue |
                (v.getDefRefName() = defR)) in

    refValues.value
/]
```

Figure 8.4: Code snippet of a sub-query needed to implement the `getWCET` query.

Language (OIL) file, which defines the configuration of the operating system, a specific statement is generated whenever a task must be subject to timing protection. In the ERIKA operating system, the corresponding OIL statement is the following:

```
TIMING_PROTECTION = TRUE {
        EXECUTIONBUDGET = <Execution Budget>;
};
```

For each task in the system for which the `isSubjectToBudgeting` query returns TRUE, the above OIL statement is generated with Execution Budget equal to the result of the `getWCET` query.

8.1.6 Handling budget overruns

The AUTOSAR OS specfification provides the following requirement:

[SWS_Os_00064] If a task's OsTaskExecutionBudget is reached then the Operating System module shall call the `ProtectionHook()` with `E_OS_PROTECTION_TIME`.

The ERIKA operating system matches this requirement. Accordingly, the RTE generator automatically generates a skeleton for the `ProtectionHook()` function. The skeleton provides the code to handle the `E_OS_PROTECTION_TIME` exception and retrieves the identifier of the task that generated the exception. The skeleton is reported in Listing 8.1.

Listing 8.1: Code skeleton to handle timing faults.

```
ProtectionReturnType ProtectionHook (StatusType FatalError)
{

  ProtectionReturnType policy;
  TaskType task_id;
  ISRType  const isr_id = GetISRID();

  if (isr_id == INVALID_ISR)
  {
    (void)GetTaskID(&task_id);
  } else
  {
    task_id = INVALID_TASK;
  }

  switch (FatalError)
  {
    /* Execution Budget expiration */
    case E_OS_PROTECTION_TIME:
      /* This part of the code is reached when task 'task_id' exceeds its budget*/

      /* (----- FILL HERE -----) */

      /* Possible policies to handle the exception:
         policy = PRO_TERMINATETASKISR;
         policy = PRO_TERMINATEAPPL;
         policy = PRO_SHUTDOWN;
          */

      /* Any other value of policy will be translated in PRO_SHUTDOWN */
    break;

    /* Handled only Timing Protection Error. Other errors:
       MP, stack overflow are Shutdown. */
    default:
      policy = PRO_SHUTDOWN;
  }

  return policy;
}
```

8.1.7 Handling errors during the RTE generation

During the generation process, it may happen that a task that must be subject to timing protection is composed of runnables whose sum of execution time bounds is not positive. In this case, the generator logs this error in a dedicated file named `timing_protection_errors.txt`.

8.1.8 Example

This section illustrates an example on which the extended RTE generator has been executed. As it is illustrated in Figure 8.5, the example consists of two software components that comprise a total of five runnables mapped to three tasks. The parameters of the tasks are reported in Table 8.1, and the execution time bounds of the runnables are reported in Table 8.2.

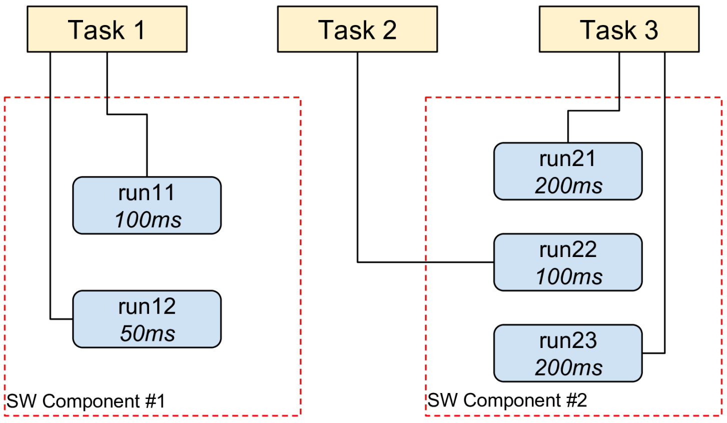


Figure 8.5: Illustration of the example task set.

Note from Table 8.1 that both Task 1 and Task 3 must be subejct to timing protection, as they both have a higher priority with respect to Task 3 that has a positive criticality level. The Execution Budget of Task 1 must be set to 55, which is the sum of the execution times provided for runnables run11 and run12 (when the execution time is not provided in the model, as in this example is the case for run12, zero is assumed). Similarly, the Execution Budget of Task 3 must be set to $150 + 250 = 400$.

Table 8.1: Example task set.

Task	Priority	Criticality	Runnables
Task 1	2	0	run11, run 12
Task 2	1	1	run 22
Task 3	3	0	run21, run23

Table 8.2: Execution times for the runnables of the example task set in Table 8.1.

Runnable	Execution time
run11	55
run12	-
run21	150
run22	-
run23	250

Listing 8.2 reports an excerpt of the OIL file (operating system configuration) generated by the RTE generator. Note that a TIMING_PROTECTION entry is generated for both Task 1 and Task 3 with the correct Execution Budget.

Listing 8.2: OIL snippet generated for the example task set.

```
TASK Task_1 {
            ACTIVATION = 1;
            PRIORITY = 2;
            SCHEDULE = FULL;
            TIMING_PROTECTION = TRUE {
                    EXECUTIONBUDGET = 55;
            };
            RESOURCE RTE_Resource_swc1_pport1_intVal1;
            RESOURCE RTE_Resource_swc1_pport1_intVal2;
    };

    TASK Task_2 {
        ACTIVATION = 1;
        PRIORITY = 1;
        SCHEDULE = FULL;
    };

    TASK Task_3 {
        ACTIVATION = 1;
        PRIORITY = 3;
        SCHEDULE = FULL;
        TIMING_PROTECTION = TRUE {
                EXECUTIONBUDGET = 400;
        };
    };
```

8.2 Spatial and Temporal Isolation on shared caches and DRAM memories for automotive platforms.

8.2.1 Introduction

In the analysis of most hypervisors, including those for mixed-critical automotive and communications systems, spatial and temporal concerns are handled in software at the level of the task partitions and at the level of the scheduling. However, in modern multicore architectures, interference can happen at low level, that is, at the level of the HW features shared between the tasks in the cores. The focus of the work reported in this section is to provide an analysis of possible actions to mitigate or remove the problem of:

- Spatial isolation on shared cache levels;
- Temporal isolation on DRAM memory accesses.

8.2.1.1 Contention due to shared cache levels

All multicore CPUs include a cache memory hierarchy to improve performance. Typically, the first level of the hierarchy consists of small and fast cache memories reserved for each single core, while the second level is commonly composed of a large cache memory shared between all the cores. Some designs even include hierarchies with more than two levels. In embedded real-time systems, the CPU cache memory hierarchy is one of the significant sources of unpredictability. In fact, the cores run simultaneously and one of them can replace the data placed in the LLC by another core, so generating a mutual interference that can be highly unpredictable (as well as strongly dependent on the application behavior). For instance, this is a problem when porting applications from single- to multi-core platforms. As long as caches are concerned, the execution time of a real-time task in a multicore CPU can be affected by different types of interferences:

- Intra-task: interference occurs when two memory entries in the working set are mapped into the same cache set;
- Intra-core: interference happens locally in a core. Specifically, when a preempting task evicts the preempted tasks cached data;
- Inter-core: interference is present when tasks running on different cores access a shared level of cache concurrently. When this happens, if two lines in the two addressing spaces of the running tasks map to the same cache line, said tasks could repeatedly evict each other in the cache, leading to complex timing interactions and thus unpredictability;

In the case of an hypervisor which assigns a core to each partition, it is possible to assign fractions of the shared cache to each partition to realize the second level of private cache, thus providing spatial isolation and reducing the problem of inter-core interference.

8.2.1.2 Memory bandwidth contention

Another primary shared resource for a multicore embedded system is the main memory (RAM). In this case, at the level of the hypervisor, we have two main problems that are spatial isolation and temporal isolation. As for spatial isolation, the hypervisor will have to ensure that the partitions have a separate memory space between them. To ensure the separation of memory space the proposed solution leverages the two stages translation capabilities of the MMU provided by the reference architecture (i.e., ARM-VE). The isolation problem is still present when access to uncached memory generates a cache miss and is needed access to DRAM memory through the DRAM memory controller which is unique, and hence the overall bandwidth is shared between the cores contending it. The problem occurs if one of the partitions, even non real-time, begins to have an abnormal behavior by starting a series of accesses to the main memory, even if not accessing the space of the other partitions. A possible way to limit this type of problem is to use a bandwidth reservation mechanism so that each partition has a maximum number of RAM access guaranteed within a given time window.

8.2.1.3 Integration with hardware-based security

The robustness of a system to face external software attacks is becoming a significant requirement for those systems that provide secure services and/or store confidential data such as cryptographic keys. The ARM TrustZone Technology is a hardware-based security built into SoCs by semiconductor chip designers who want to provide secure endpoints and a device root of trust. At the heart of the TrustZone approach is the concept of hardware separation between secure and non-secure worlds, with the non-secure one blocked from directly accessing secure resources, with the switch between these two worlds accomplished via a software referred to as the secure monitor. This concept of secure (trusted) and non-secure (non-trusted) worlds extends beyond the processor to encompass memory, software, bus transactions, interrupts and peripherals within a SoC. Virtualizing the two worlds allows stand-alone programs, with mixed-criticality and security levels, to coexist in the same Hardware platform. When merging this kind of systems, the biggest requirement is the strict isolation between them, that needs to be achieved not only in the Non-Secure World but also in the Secure World, even if the secure part of the software is certified or trusted. Figure 8.6 shows a possible application scenario.

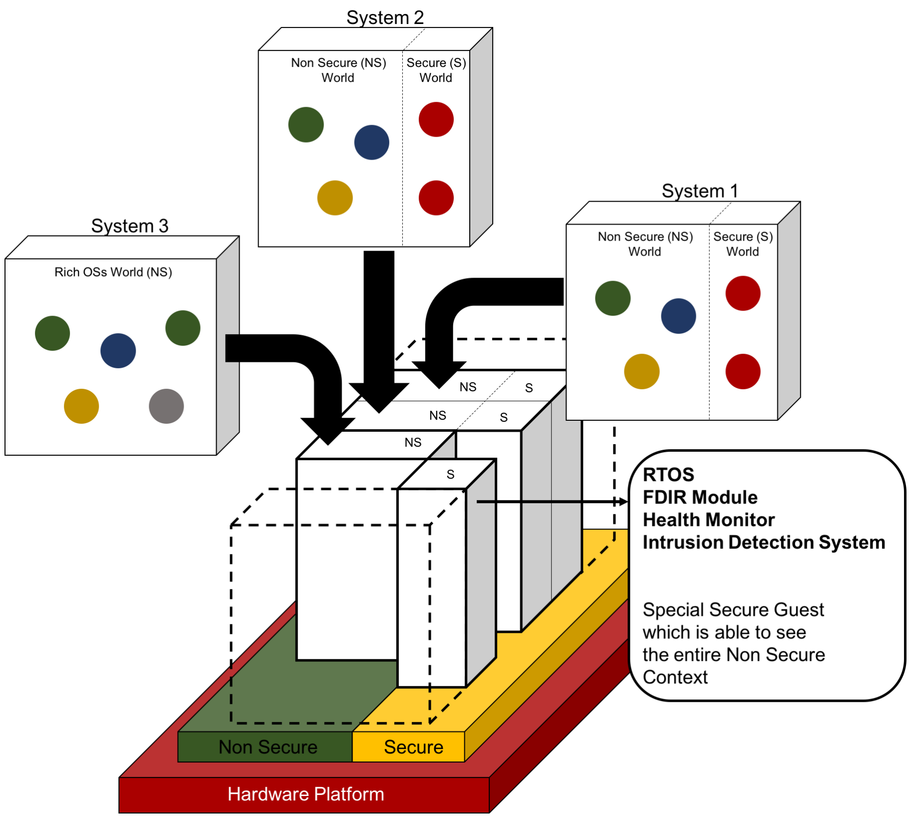


Figure 8.6: Merging Systems with mixed criticality and security levels.

8.2.2 Hypervisor setup

Xvisor [29], shown in Figure 8.7, is an embedded open-source monolithic Type-1 hypervisor that supports both full virtualization and para-virtualization. It aims at providing a lightweight hypervisor that can be used within embedded systems with small overhead and memory footprint. Xvisor provides fully virtualized guest, Hardware-assisted full-virtualization when possible (ARM Virtualization Extensions) and provides para-virtualization also in the form of optional VirtIO devices [31].

The Xvisor source code is light-weight and highly flexible and can be easily ported to most general-purpose 32-bit or 64-bit architectures as long as they have a paged memory management unit (PMMU) and a port of the GNU C compiler (GCC). It provides a high performance and low memory foot print virtualization solution for ARMv5, ARMv6, ARMv7a, ARMv7a-ve, ARMv8a, x86_64, and other CPU architectures.

The most important advantage of Xvisor is its single software layer running with the highest privilege, in which all virtualization related services are provided. Xvisors context switches are lightweight

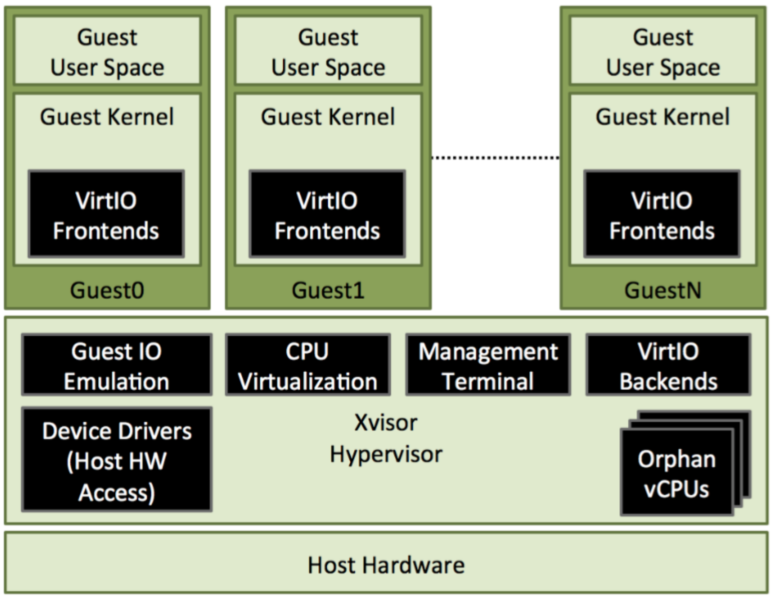


Figure 8.7: Xvisor System Architecture (picture from [29])

resulting in the fast handling of nested page faults, special instruction traps, host interrupts, and guest IO events. Furthermore, all device drivers run directly as part of Xvisor with full privilege and without nested page table ensuring no degradation in device driver performance. Also, the Xvisor vCPU scheduler is per-CPU and does not do load balancing for multiprocessor systems. The multiprocessor load balancer is a separate entity in Xvisor, independent of the vCPU scheduler. Both, vCPU scheduler and load balancer are extensible in Xvisor.

Xvisor provides Hardware-Assisted and Para-Virtualization (in case of no Virtualization Extension).

Figure 8.8 shows how Xvisor in ARM with Virtualization Extensions implements the Guest IO Emulation. The scenario starts at (1) when a guest IO event is trapped by Xvisor ARM and (2) handles it in a non-sleepable normal (or emulation) context. The non-sleepable normal context ensures fixed and predictable overhead.

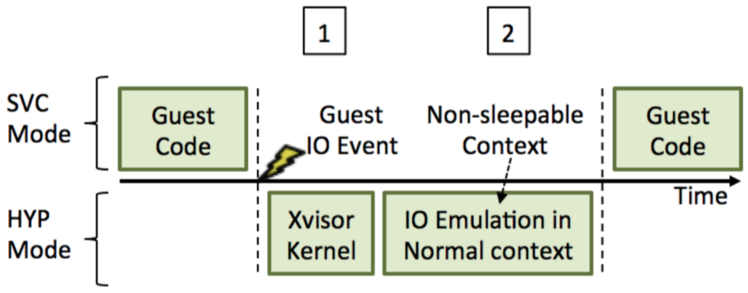


Figure 8.8: Emulated Guest IO event (picture from [29]).

Xvisors host device drivers commonly run as part of Xvisor with the highest privilege. Figure 8.9 shows how Xvisor in ARM with Virtualization Extensions handles the Host Interrupts while a Guest OS is running. A scheduling overhead only incurs if the host interrupt is routed to a guest which is not running.

8.2.3 Memory Management, Virtual CPUs

The ARM architecture with Virtualization Extension provides two-staged translation tables (or nested page tables) for memory virtualization. The guest OS is responsible for programming stage1 transla-

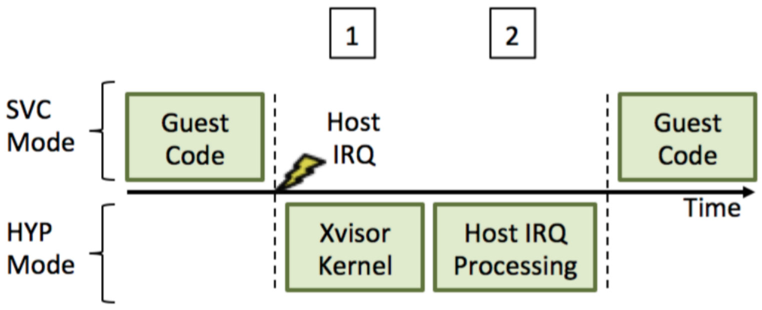


Figure 8.9: Host interrupts handling (picture from [29]).

tion table which carries out guest virtual address (GVA) to intermediate physical address (IPA) translation. The ARM hypervisors are responsible for programming stage2 translation table to achieve intermediate physical address (IPA) to actual physical address (PA) translation. Translation table walks are required upon TLB misses. The number of stage2 translation table accesses affects the memory bandwidth and the overall performance. To reduce TLB-miss penalty in two-staged MMU, ARM hypervisors create bigger pages in stage2 translation table.

Virtual machines are separated into two major categories (based on their use):

- System Virtual Machine: A system virtual machine provides a complete system platform which supports the execution of a complete operating system (OS).
- Process Virtual Machine: A process virtual machine is designed to run a single program, which means that it supports a single process.

Xvisor refers system virtual machine instances as ”Guest” instances and virtual CPUs of system virtual machines as ”VCPU”.

8.2.3.1 Guest Management

Each Guest Region has a unique Guest Physical Address (i.e. Physical address at which region is accessible to Guest VCPUs) and Physical Size (i.e. Size of Guest Region). Further a Guest Region can be one of the three forms:

- Real Guest Region: A Real Guest Region gives direct access to a Host Machine Device/Memory (e.g., RAM, UART).
- Virtual Guest Region: A Virtual Guest Region gives access to an emulated device (e.g., emulated PIC, emulated Timer, etc.).
- Aliased Guest Region: An Aliased Guest Region gives access to another Guest Region.

8.2.3.2 Hypervisor Scheduler

The Xvisor scheduler is generic and pluggable concerning the scheduling strategy (or scheduling algorithm). It updates per-CPU ready queues whenever it gets notifications from hypervisor manager about VCPU state change. The hypervisor scheduler uses per-CPU hypervisor timer event to allocate a time slice for a VCPU. When a scheduler timer event expires for a CPU, the scheduler will find next VCPU using some scheduling strategy (or algorithm) and configure the scheduler timer event for next VCPU. For Xvisor a Normal VCPU is a black box (i.e. anything could be running on the VCPU) and exception or interrupt is the only way to get back control. Whenever Xvisor is running, it could be in any one of following contexts:

- *IRQ Context*, when serving an interrupt generated from some external device of the host machine.
- *Normal Context*, when emulating some functionality or instruction or emulating IO on behalf of Normal VCPU in Xvisor.
- *Orphan Context*, when running some part of Xvisor as Orphan VCPU or Thread (Note: Hypervisor threads are described later.)

Xvisor has a special context called Normal context. The hypervisor is in Normal context only when it is doing something on behalf of a Normal VCPU such as handling exceptions, emulating IO, etc. The Normal context is non-sleepable which means a Normal VCPU cannot be scheduled-out while it is in Normal context. In fact, a Normal VCPU is only scheduled-out when Xvisor is exiting IRQ Context or Normal Context. This helps Xvisor ensure predictable delay in handling exceptions or emulating IO.

The expected high-level steps involved in architecture specific VCPU context switching are as follows:

- Save arch registers (or arch_regs_t) from the stack (saved by architecture-specific exception or interrupt handler) to current VCPU arch registers (or arch_regs_t).
- Restore arch register (or arch_regs_t) of next VCPU on the stack (will be restored when returning from an exception or interrupt handler C code).
- Switch context of architecture specific CPU resources such as MMU, Floating point subsystem, etc.

The possible scenarios in which a VCPU context switch is invoked by scheduler are as follows:

- When time slice allotted to current VCPU expires we invoke VCPU context switch. We call this situation as VCPU preemption.
- If a Normal VCPU misbehaves (i.e. does invalid register/memory access) then architecture specific code can detect such situation and halt/pause the responsible Normal VCPU using APIs from hypervisor manager.
- An Orphan VCPU (or Thread) chooses to voluntarily pause (i.e. sleep).
- An Orphan VCPU (or Thread) chooses to voluntarily yield its time slice.
- The VCPU state can also be changed from some other VCPU using hypervisor manager APIs.

8.2.3.2.1 Scheduling algorithm

Currently, Xvisor supports two classic scheduling algorithm:

- Fixed Priority Round-Robin (PRR) (default)
- Rate Monothonic (RM)

8.2.3.3 ARM platform description

The ARM architecture used as a reference platform for the implementation of the proposed solution is a 32-bit ARMv7-A Cortex-A15 with the Virtualization Extension and the Security Extension.
In this section further details are provided for the ARM platform for two main reasons:

- The extensions described in the following sections have been implemented by modifying the ARM implementation of Xvisor. In addition, the specific features of the ARM platform are leveraged to obtain the implementation of some techniques (cache coloring) described in the following.

- The ARM platform is among the most popular choices for automotive applications.

Of course, the problem of avoiding interference at the memory level is highly specific to the features of the memory hierarchy and the control of the acess to the memory. In our prototype implementations we used the ARM platform.

Processor modes As shown in Table 8.3, an ARMv7 processor has up to 9 different modes depending on if optional extensions have been implemented. The usr mode that has a privilege level 0 is where userspace programs run. The svc mode that has a privilege level 1 is where most parts of kernel execute. However, some kernel modules run at special modes instead of svc. For example, when a data abort exception happens, a processor switches to the abt mode automatically. Processor mode change can be triggered by exceptions, such as the aforementioned data abort exception. Or, privileged program can directly write CPSR by calling a *MSR CPSR_c, #imm* instruction, where c stands for the control field that includes processor mode bits and interrupt mask bits.

Mode	**Abbr.**	**Privilege level**	**Security state**
User	usr	PL0	both
Supervisor	svc	PL1	both
System	sys	PL1	both
Abort	abt	PL1	both
IRQ	irq	PL1	both
FIQ	fiq	PL1	both
Undefined	und	PL1	both
Monitor*	mon	PL1	Secure only
Hyp**	hyp	PL2	Non-secure only

* only implemented with Security Extension
** only implemented with Virtualization Extension

Table 8.3: Processor modes.

Processor states With the security extensions, a processor has two security states, namely the secure state (s) and the non-secure state (ns). The distinction between the two states is orthogonal to the mode protection based on privilege levels, except that the mon mode is only available in the secure state and the hyp mode that is implemented with virtualization extensions only exists for the non-secure state. The current processor state is determined by the least significant bit of the secure configuration register (SCR) in the CP15 coprocessor.

Core registers Figure 8.10 compares the ARMv7 architecture core registers between the application level view and system level view. From the application-level perspective, an ARMv7 processor has 14 general-purpose 32-bit registers (R0 to R14), a 32-bit program counter R15 also known as PC, and a 32-bit application program state register (APSR). Two of the 14 general-purpose registers can be used for special purposes: R13 also known as SP is usually used as the stack pointer; R14 also known as LR is usually used to store return address. APSR is an application level alias for CPSR, and it must be only used to access condition flags.

From the system level view, these registers are arranged into several banks, which means a register name is mapped to a collection of different physical registers, governed by the current processor mode. As shown in Figure 8.10, each mode except the system mode of the processor has:

- its banked copy of stack pointer SP;
- a register that holds a preferred return address for the exception (a banked copy, such as LR_mon, for LP1 modes or a special register ELR_hyp for the hyp mode);
- a copy of saved program status register SPSR to save the copy of CPSR made on exception entry (except the usr and system).

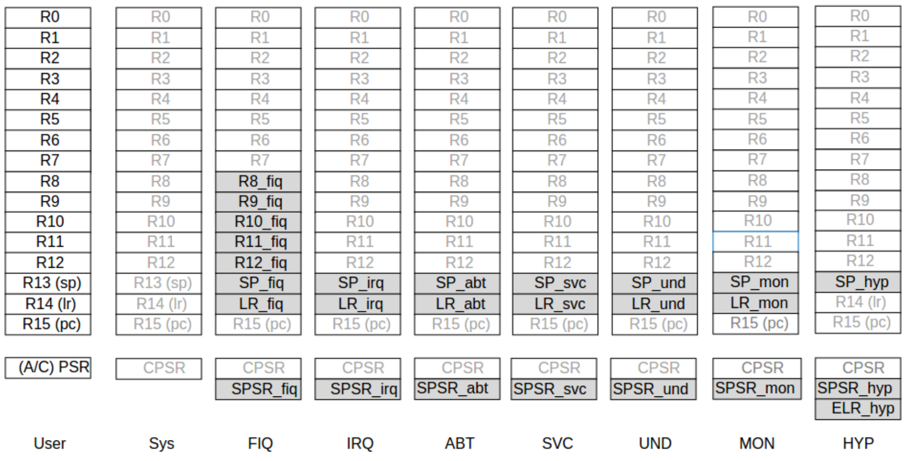


Figure 8.10: ARMv7-A Core Register (from [6]).

Saving the value of CPSR in banked SPSR registers means the exception handler can immediately restore the CPSR on exception return and examine the value of CPSR when the exception was taken, for example, to determine the previous process mode when the exception took place. Also, the fiq mode has banked copies of R8 to R12. For example, when a processor is executing in the fiq mode, R0 refers to R0_usr, but R12 refers to R12_fiq instead of R12_usr.
Note that processor core registers and program status registers are not banked between the secure state and non-secure state. Therefore, a program can use registers to pass parameters between states. Also, during a processor state switch, a privileged program mostly running in Monitor mode will save the old state's register values and restores the new states register values.

Coprocessors and System Registers The ARM architecture supports sixteen coprocessors, namely CP0 - CP15, in which CP15 (System Control coprocessor) is reserved in the architecture for the control and configuration of the processor system. Hardware manufacturer can define other coprocessors for their purposes.
The system registers in CP15 are categorized in many groups that include

- virtual memory control registers function group (SCTLR, DACR, TTBR0, TTBR1, PRRR)
- PL1 Fault handling registers
- cache maintenance operations
- address translation operations
- security Extensions registers.

Given the special purpose of CP15 system registers, many of them are banked between secure and non-secure states. However, the registers that configure the global system status, such as SCR, are not banked. Table 8.4 lists some CP15 system registers that are used in this work.

8.2.3.3.1 ARM Virtualization Extension

The ARM Architecture virtualization extension and Large Physical Address Extension (LPAE) enable the efficient implementation of virtual machine hypervisors for ARM architecture compliant processors. To handle complex software with potentially large amounts of data, connected consumer devices and cloud computing demand energy efficient, high-performance systems. The virtualization extensions provide the basis for ARM architecture compliant processors to address the needs of both client and server devices for the partitioning and management of complex software environments into

Register	Name	Security state
VBAR	Vector Base Address Register	Banked in both states
MVBAR	Monitor Vector Base Address Register	Secure state, monitor mode
ISR	Interrupt Status Register	
SCR*	Secure Configuration Register	Secure state
TTBRx	Translation Table Base Register (0), (1)	Banked in both states
TTBCR	Translation Table Base Control Register	Banked in both states
DACR	Domain Access Control Register	
SCTLR	System Control Register	Banked in both states
NSACR*	Non-Secure Access Control Register	Secure state
SDER	Secure Debug Enable Register	

* only implemented with Security Extension

Table 8.4: Some CP15 Registers on ARMv7-A

virtual machines. The Large Physical Address extension provides the means for each of the software environments to utilize the available physical memory efficiently when handling large amounts of data. The basic idea of this architecture is based on the presence of an additional higher privileged mode: HYP Mode.

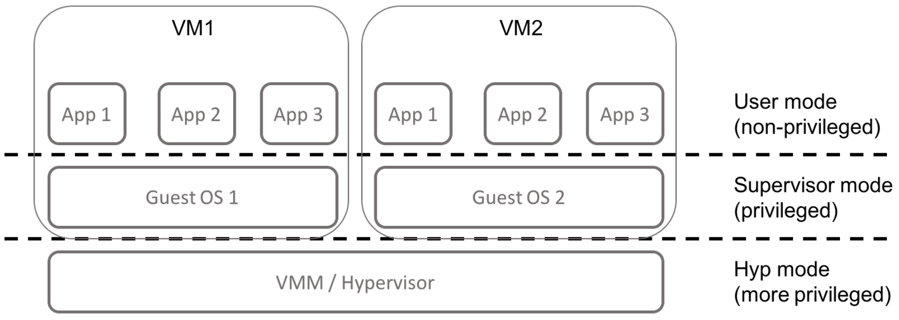

VM1
VM2
App 1
App 2
App 3
App 1
App 2
App 3
User mode (non-privileged)
Guest OS 1
Guest OS 2
Supervisor mode (privileged)
VMM / Hypervisor
Hyp mode (more privileged)

Figure 8.11: ARM Virtualization Extensions Modes.

The basic model of a virtualized system involves:

- a hypervisor, running in Hyp mode, that is responsible for switching Guest operating systems;
- a number of Guest operating systems, each of which runs in the PL1 and PL0 modes (respectively Supervisor and User);
- for each Guest operating system, applications that usually run in User mode.

A Guest operating system, including all applications and tasks running under that operating system, runs on a virtual machine and the hypervisor switches between virtual machines. However, the Guest OS's view is that it is running on an ARM processor. Normally, a Guest OS is completely unaware that it is running on a virtual machine.
Each virtual machine is identified by a virtual machine identifier (VMID), assigned by the hypervisor. The key features of this extension are:

- Hyp mode is implemented to support Guest OS management. Hyp mode operates in its virtual address space, that is different from the virtual address space accessed from PL0 and PL1 modes.
- The Virtualization Extensions provide controls to:
  - Define virtual values for a small number of identification registers. A read of the identification register by a Guest OS or its applications returns the virtual value.

  - Trap various other operations, including accesses to many other registers, and memory management operations. A trapped operation generates an exception that is taken to Hyp mode.

  These controls are configured by software executing in Hyp mode.

- With the Security Extensions (TrustZone), the Virtualization Extensions control the routing of interrupts, and asynchronous Data Abort exceptions to the appropriate one of:
  - the current Guest OS;
  - a Guest OS that is not currently running;
  - the hypervisor;
  - the Secure monitor (better detailed in the next section).
- When an implementation includes the Virtualization Extensions, it provides independent translation regimes for memory accesses from:
  - Hyp mode, the PL2 translation regime;
  - Supervisor and User modes, the PL1&0 translation regime.
- In the PL1&0 translation regime, address translation occurs in two stages:
  - Stage 1 maps the Virtual Address (VA) to an Intermediate Physical Address (IPA). Typically, the Guest OS configures and controls this stage, and believes that the IPA is the Physical Address (PA);
  - Stage 2 maps the IPA to the PA. Typically, the hypervisor controls this stage, and a Guest OS is completely unaware of this translation.

8.2.3.3.2 Trustzone

The ARM TrustZone is a hardware security extension technology, which aims to provide secure execution environment by splitting computer resources between two the normal and the secure execution contexts. As ARM is widely deployed on the majority of mobile and micro-controller devices, TrustZones goal is to provide security for those platforms.
A system is usually only secured at the software level. However, a greater level of security can be achieved by building security checks into the hardware of the system. This idea is implemented by the concept of Trusted Execution Environments (TEE).
The Trusted Execution Environment (TEE) is a secure area of the main processor. It guarantees code and data loaded inside to be protected concerning confidentiality and integrity. The TEE as an isolated execution environment provides security features such as the isolated execution and the integrity of Trusted Applications along with confidentiality of their assets. In general terms, the TEE offers an execution space that provides a higher level of security than a Rich OS.

Trustzone provides support for an hardware-based TEE. Formal definition and specification of a TEE has been defined by GlobalPlatform [16]. The security of TrustZone is based on the idea of partitioning all of the System on Chip (SoC)s hardware and software into two worlds: Secure World and Normal World. Hardware barriers are established to prevent normal world components from accessing secure world resources; the secure world is not restricted since it has full control of the entire system (the non-secure world as well). Specifically, the memory system prevents the normal world from accessing:

- regions of the physical memory designated as secure;
- system controls that apply to the secure world;
- state switching outside of a small number of approved mechanisms.

ARM introduced two versions of TrustZone: one for ARM-A and one for ARM-M. The one used in this work, and so described in this document, is the ARM-A version.

Modes view and System IPs

The trustzone idea is based on having a sort of additional bit checked in every instruction and transaction: the NS Bit which identifies which world is currently running. We can erroneously say that a 32-bit processor, with the Security Extensions, is going to be a 33-bit processor where the 33rd is the NS bit. However, this is a wrong definition, but it gives a good overview of the mechanism. In fact, the NS bit is in a unique register (Secure Configuration Register - SCR) in the CP15 only accessible by the Secure World.

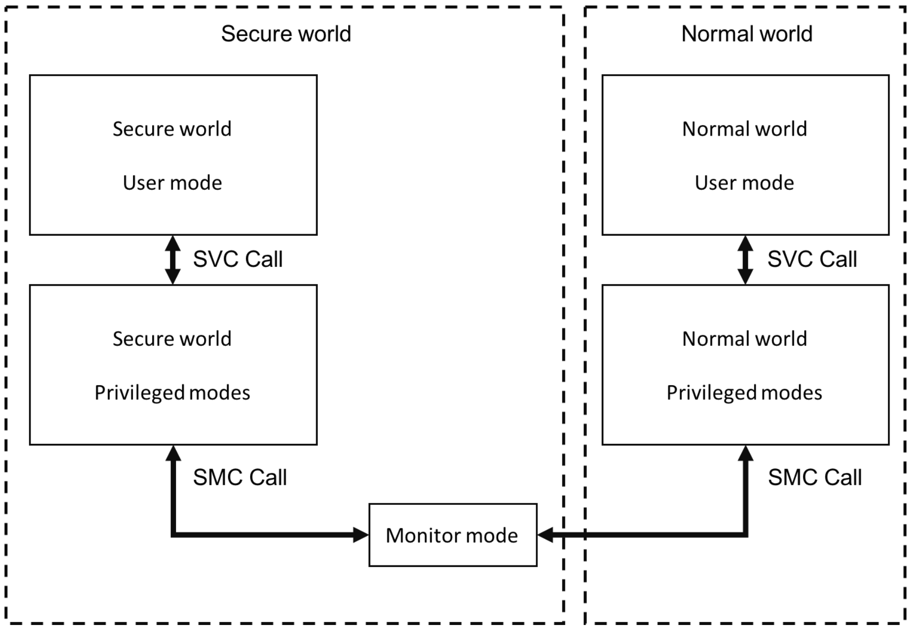


Figure 8.12: Modes overview in a TrustZone-enabled Processor.

As shown in the Figure 8.12, the traditional User and Privileged Modes are orthogonally split into the two different worlds. A new entity is introduced which has the highest privileges and is mainly responsible for the context switching between the two worlds: the Monitor.

ARM has implemented this split-environment processor with various system IP additions. These unique components are used to enforce security restrictions while preserving the low power consumption and other advantages of ARMs designs. Some of the features are described in the specification for ARMs Advanced Microcontroller Bus Architecture version 3 (AMBA3). The main additional system IPs of the Security Extensions are a APB Bridge (AXI-to-APB Bridge), a modified Cache Controller, a modified DMA Controler, a TrustZone Address Space Controller (TZASC), a modified Generic Interrupt Controller (GIC), and a TrustZone Protection Controller (TZPC). The AMBA3 AXI to APB Bridge allows for secure communication between a CPU and peripherals. The Advanced eXtensble Interface (AXI) bus, which is the main system bus, contains an active-high non-secure (NS) bit that indicates whether a read/write operation is directed to secure or non-secure memory. The Advanced Peripheral Bus (APB), whose low bandwidth reduces power consumption, connects to the AXI bus via a bridge. As the APB does not check for security due to backward-compatibility concerns, the bridge checks for appropriate permissions and blocks unauthorized requests.
Like the AXI to APB bridge, the Cache Controller also looks for an NS bit. This bit is treated like a 33rd address bit: the first 32 bits provide the location, and the NS bit indicates which world is referring. Since both worlds share the same physical cache, the same location may have two distinct addresses, requiring a controller to look up the correct location. This also includes L2 cache and other smaller locations.

The Direct Memory Access (DMA) Controller is used to transfer data to physical memory locations instead of devoting processor cycles to this task. This controller, which uses AXI, can handle Secure and Non-secure events simultaneously, with full support for interrupts and peripherals. It prevents non-secure access of secure memory.

The TrustZone Address Space Controller (TZASC) allows dynamic classification of AXI slave memory-mapped devices as secure or non-secure. Controlled by the secure world, the TZASC allows partitioning of a single memory unit rather than requiring separate secure and non-secure units. The TZASC allows an arbitrary number of partitions to be created.

The Generic Interrupt Controller (GIC) is a single hardware device that supports both Secure and Non-secure prioritized interrupt sources. Attempts by Normal world software to modify the configuration of an interrupt line configured as a Secure source will be prevented by the GIC hardware. Additionally, Non-secure software can only configure interrupts in the lower half of the priority range, preventing denial-of-service attacks.

Finally, the TrustZone Protection Controller (TZPC) is a signal-control unit. It has three 2-bit registers to control up to 8 signals.
The TrustZone Hardware, where hardware extensions enforce a separation of secure and non-secure software, is more resource-efficient than the use of two separate processors.

Figure 8.13 shows the block diagram of AXI Bus TZ-Enabled.

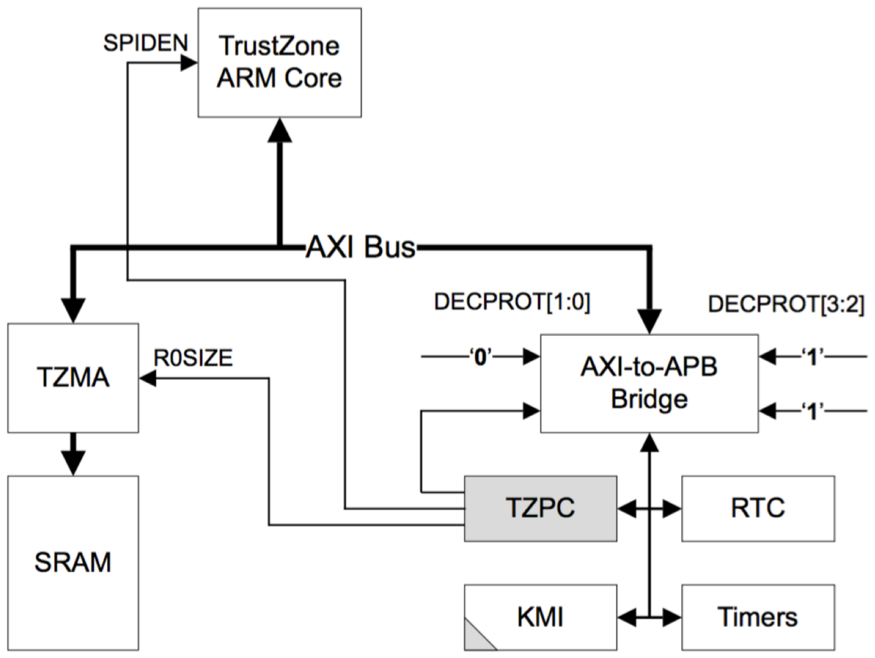


Figure 8.13: AXI Bus TZ-Enabled [6].

8.2.3.3.3 World Switching Mechanism

To switch synchronously between worlds, TrustZone for ARM-A introduced a special form of Software Generated Interrupt which is called Secure Monitor Call (SMC) (see Figure 8.12). When the processor executes the Secure Monitor Call (SMC) the core enters Secure Monitor mode to execute the Secure Monitor code. This instruction can only be executed in privileged modes, so when a User process wants to request a change from one world to the other, it must first execute an SVC instruction. This changes the processor to a privileged mode where the Supervisor call handler processes the SVC and executes a SMC. The Secure Monitor mode is typically responsible for switching between worlds. The recommended way to return from an SMC call is to:

1. Toggle the NS bit in the SCR (so setting it if we are going to the Non Secure world or clearing it if we are going to the Secure world).
2. Execute a MOVS, SUBS or RFE.

All ARM implementations ensure that the processor can not execute the prefetched instructions that follow MOVS, SUBS, or equivalents, with Secure access permissions.

However the world's switching mechanism is also supported asynchronously by the Hardware, for instance when an Interrupt for the Secure World is raised while the Non-Secure World is running, and/or vice-versa.

8.2.4 Cache partitioning

The inter-core interference problem can be solved by partitioning the shared cache in smaller subsets assigned to individual cores or partitions. There are two types of cache partitioning: index-based or way-based partitioning. In the index-based partitioning (also called horizontal slicing), the partitions are formed by aggregation of the caches sets. In the way-based partitioning (also named vertical slicing), the partitions are constructed by aggregating one or more cache ways. We selected index-based partitioning that requires the modification of the virtual memory management within the hypervisor.

8.2.4.1 ARM Cache Architecture

This subsection describes the shared cache architecture used in ARM processors, recalling the general concepts and the terminology. In particular:

- A line is the smallest loadable unit of a cache, a block of contiguous words from main memory;
- An index is the part of a memory address that determines in which line(s) of the cache the address can be found;
- A way is a subdivision of a cache, each way being of equal size and indexed in the same fashion. The lines associated with a given index value from each way are grouped to form a set;
- The tag is the part of a memory address stored in the cache that identifies the main memory address associated with a line of data.

The main caches of ARM cores are always implemented as set associative. The cache is divided into equally-sized ways. A memory location is mapped to a line. The index field of the address is used to select a particular line and points to an individual line in each way. In our work, we used a quad-core ARM Cortex-A7 processor, with two cache levels. Each core has a 32 KB L1 2-way set-associative instruction cache, with 32-bytes line length, and a 32 KB L1 4-way set-associative data cache, with 64-bytes line length. All the cores share a second level of cache. The L2 shared cache has the following feature:

- 512 KB cache size;
- fixed line length of 64 bytes;
- physically indexed and tagged cache;
- 8-way set-associative cache structure;
- pseudo-random cache replacement policy.

8.2.4.2 Cache Coloring

To achieve index-base partitioning we implemented page coloring: a software technique to control the mapping by which index links the physical memory addresses to a cache set; this is done in hardware so that each address is mapped to one set of the cache. The definition of color indexes makes use of the bits that overlap between the physical page number and the index of the set. In this way, the hypervisor can assign different colors to different guests. The number and the size of colors are hardware-dependent because they are linked to the cache address format.

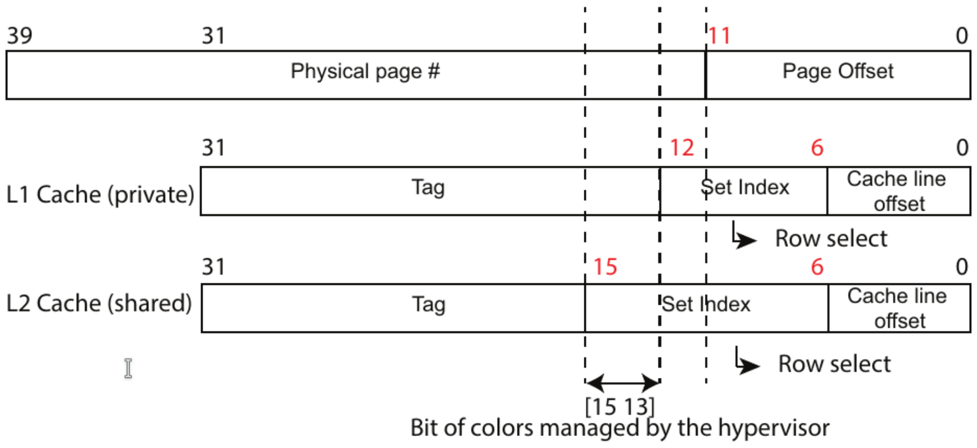


Figure 8.14: Cortex A7 with 512 KB of L2 shared cache, address bits.

Figure 8.14 shows the structure of the addresses for the cache hierarchy of the Cortex A7 with 512KB of L2 cache. The physical page number overlapped with the L1 index set for one bit, which means that it can be divided into two colors at most. For the L2 cache, the overlapping bits are four, so it is possible to have up to 16 colors. However, this would require partitioning the cache L1, which is not interesting because it is already private to each core. Hence, the useful bits for partitioning the L2 cache are only three and allow to obtain up to 8 colors each with a size of 8KB, as can be easily derived from the position of the bits within the address. To assign a shared cache partition to different guests using colors the hypervisor must be modified to allocate memory space using addresses that belong to the colors chosen for the specific guest. Each guest is given access to a virtual space of contiguous memory that is allocated in physical memory in a discontinuous manner, according to the above rules. To implement this mechanism, the double level of addresses translation is required, especially by changing the translation tables of the stage 2. Figure 8.15 shows a high-level architecture of the system just described.

Each guest OS will have a virtual address space and will have the task of managing and set page tables for the stage 1 translation. The guest OS will use the generated addresses as physical addresses because it is unaware of running in a VM, but the generated addresses are IPA and will not be used to address the host memory. Due to this solution, everything that happens in guest memory, including the first level of translation, is unknown to the hypervisor that does not need to know specific details, but only to which guest each IPA belongs. Using this information and the guest colors the hypervisor can set the stage 2 translation tables in such a way the coloring rules are respected.

8.2.5 Implementing coloring on ARM with Xvisor

In Xvisor, the devices visible to each guest are specified through a DTS file. Inside the tree, there must be an aspace node that represents the sub-tree where all the devices with their characteristics are listed. Each node within the space node is mapped to the system as a region. The regions that are mapped into memory must have the following parameters:

- **manyfest_type** = ”real”, mean that the region is real;

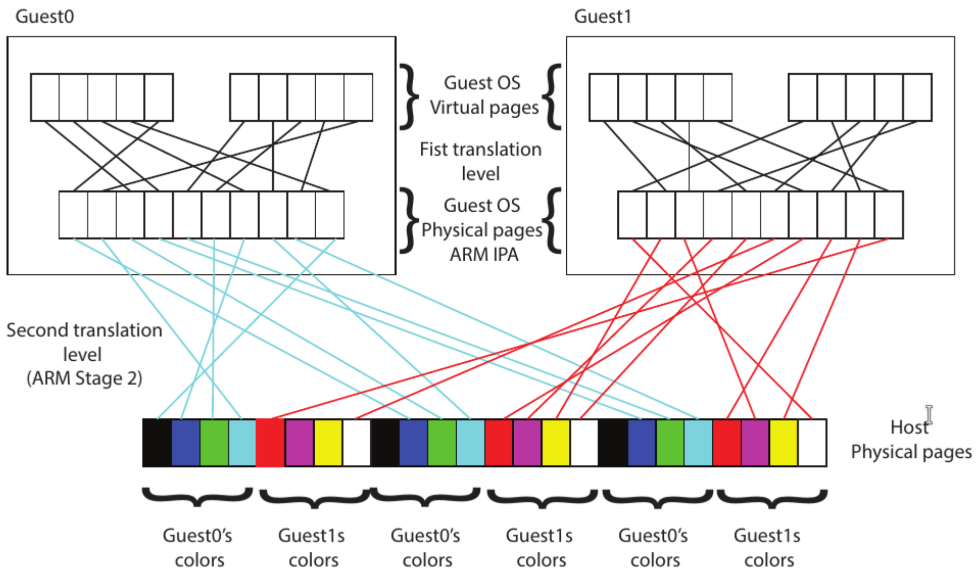


Figure 8.15: Hypervisors cache coloring architecture.

- **address_type** = ”memory”, mean that the region will be mapped in DRAM memory;
- **device_type** = ”alloced_ram” or ”alloced_rom”, mean that the region represents a memory RAM or ROM for the guest.

The regions also have two parameters indicating the physical address that the guest will use to address that particular region and the corresponding size. When a guest is created, Xvisor generates a data structure for each region found in the DTS file in which are saved all the information about the region, this structure will then be stored in memory as a node in a red-black tree so that it can be efficiently searched. When it comes to a region that must be allocated in memory, Xvisor searches a continuous piece of DRAM memory of the same size as those specified in the DTS file that is not yet assigned to any guest. Once the piece of memory is found, it is marked as allocated and will no longer be available to other guests and the initial data address of the allocated memory area (host physical address) is saved in the data structure.

In the data structure, Xvisor will have a guest physical address (GPA), a host physical address (HPA) and a size. When the guest attempts to access the GPA for which the Stage 2 page table is not yet present an abort (instruction or data) is generated, and Xvisor searches for the region data structure, extract the host physical address, and fill in the page table of Stage 2 appropriately. This approach allows the guest to resume and use the DRAM region that was assigned to him.

The problem with this approach is that each guest is assigned a portion of contiguous physical memory, which means that each guest uses memory parts that will be mapped across the entire shared cache. The consequence is that two hosts running simultaneously on two different CPUs are generating inter-core interference with each other due to the continuous replacement of data at the shared cache level.

8.2.5.1 Changes to Xvisor

The first thing to be changed is the structure of the regions due to the need to allocate more pieces of host memory to the same region; hence a single HPA will not be enough. The new regions require additional fields to support coloring:

Each region is divided into several pieces that are called region maps. Each of these maps addresses a portion of host memory. Using the *map_order* parameter is possible to change the size of the maps, that is necessary because to each guest it is possible to assign more contiguous colors to use a larger map and also because the size of the color is architecture dependent. The *map_count* variable is needed to save the number of maps composing the region, and it is calculated using the size of the region and the size of the map. The maps pointer variable points to a vector, of size *map_count*, in which are saved all the maps that compose the host memory region. The *colors* variable allows saving which colors have the memory pieces of the region. It is a mask where each bit represents if the corresponding color belongs to a guest (value 1) or not (value 0). In the reference architecture, it uses the first eight bit.

The initialization flow of the guest address space partially changed: instead of searching a single block of host memory for each memory region, Xvisor searches for each map a block of host memory having the map size and compatible with the colors saved in the region colors variable.

All the parameters needed to the guest coloring can be passed to Xvisor through the DTS file. Thus the parser has been extended with the new parameters *colors* and *map_order*. In this way, no code modification is needed to change the colors assignation to the Guests.

The *colors* parameter must be a mask of eight bits in which are set the bits of the colors that are assigned to the guest region. The lower bit means the first color; the higher bit means the eighth color. If the *colors* parameter is set to zero or isnt present the guest region will not be colored.

The *map_order* parameter is needed to indicate the order of the maps; it is used to calculate the size of the maps with the following formula:

$$map\_size = 2^{map\_order} \tag{8.1}$$

If the *map_order* is not specified, it is calculated taking into account the size of a color and the colors assigned. In the case of a not colored region the following relation is guarantee:

$$VMM\_PAGE\_SIZE \leq map\_order \leq align\_order. \tag{8.2}$$

8.2.5.2 Experimental Results on Cache Coloring

A set of tests has been conducted to validate the usefulness of the implemented spatial isolation technique. The tests were carried out using a Raspberry Pi 2 platform equipped with an ARM Cortex-A7 processor with a 512KB shared cache. All tests are designed to compare the predictability and amount of interference by running two guests managed by Xvisor with and without isolation.

Several tests have been conducted to test the benefits of using cache coloring. System configuration and experiments have the following features:

- Two core dedicated to Xvisor;
- One core and half of shared cache (4 contiguous colors 256KB) dedicated to Guest0;
- One core and half of shared cache (4 contiguous colors 256KB) dedicated to Guest1;
- Guest1 is used as interfering guest constantly accessing a large part of memory (10240 KB), producing cache eviction in the shared cache;
- Guest0 performs the time measurements to understand the amount of the interference generated by Guest1 in the case of no-coloring Xvisor and the case of coloring Xvisor;

The test executed by Guest0 accesses for M iterations a portion of the memory of size N KB while measuring the access time. Also, to observe the execution time trend as a function of the amount of memory accessed, the test is repeated X times by varying the parameter N in a range between 8KB and 512KB. These results allow comparing the behavior of the two Xvisors when the memory is partitioned (i.e., each guest has a limited portion) with the case where memory is fully shared among guests. For each value of N is calculated the mean and the maximum execution time to access N KB of memory. The tests are performed under the following conditions:

Figure 8.16 shows the comparison between the curves of AVG and MAX by varying the memory size accessed in each of the four cases. The case of Xvisor without coloring and a single running

	Xvisor No Colored	Xvisor Colored
Only Guest0	solo No Coloring	solo Coloring
Guest0 and Guest1	corun No Coloring	corun Coloring

Table 8.5: Test case conditions.

guest has the best AVG and MAX execution times for all memory sizes; obviously, in this case, the unique guest has a 512KB L2 cache all by itself. Conversely, the case of Xvisor without coloring running two guests has the worst performance regarding AVG and MAX execution time for all memory sizes; moreover, as the amount of memory used increases, the execution times increase considerably. We can see, however, that the non-colored Xvisor case has excellent performance from 0 to 256KB (i.e., the partition size assigned to the single guest) and is always better in the case of two guests. When using Xvisor with coloring, the difference between the performance running one guest and quite similar those executing two guests, showing the improvements concerning predictability even in the presence of multiple concurrent guests instances.

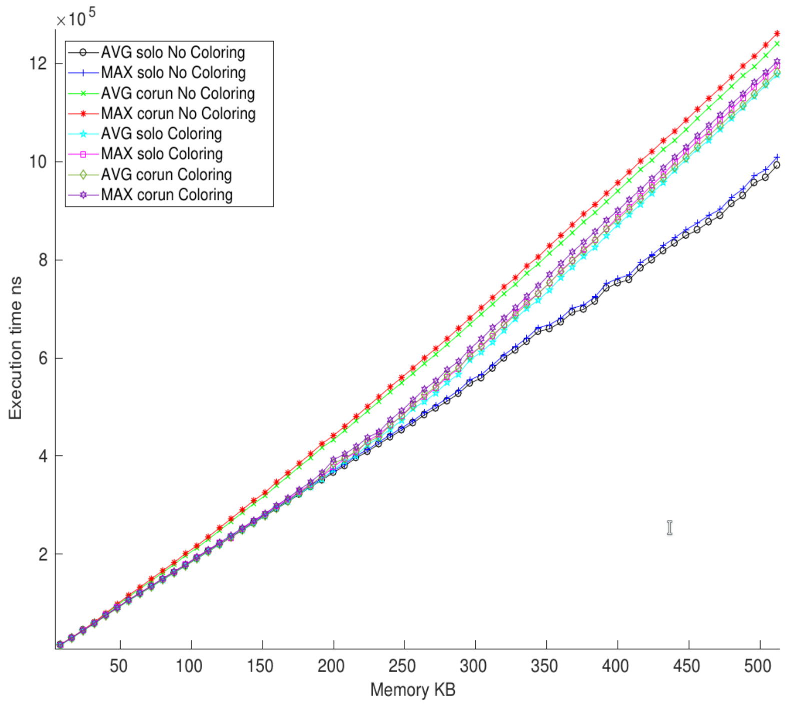


Figure 8.16: Overall comparing on AVG and MAX.

Figure 8.17 shows execution times accessing 120 KB memory area, so inside the partition assigned to the guest. In this case, Xvisor without coloring suffers a high interference generated by another guest (about 30 μs). Instead, both cases of Xvisor with coloring present comparable execution times without any interference.

8.2.6 Memory throttling

The throttling term indicates techniques for managing and allocating shared resources of a host to various guests. Throttling is needed to monitoring the workload and managing the requests of the various guests so that they do not exceed an assigned threshold. When embedded multicore platforms share the same hardware managed by a hypervisor throttling techniques cqn be applied to manage an essential shared device: the DRAM memory.

Figure 8.18 shows an example where a four core platform is allocated by the hypervisor as follows:

- Core 0 and Core 1 execute the hypervisors background code;

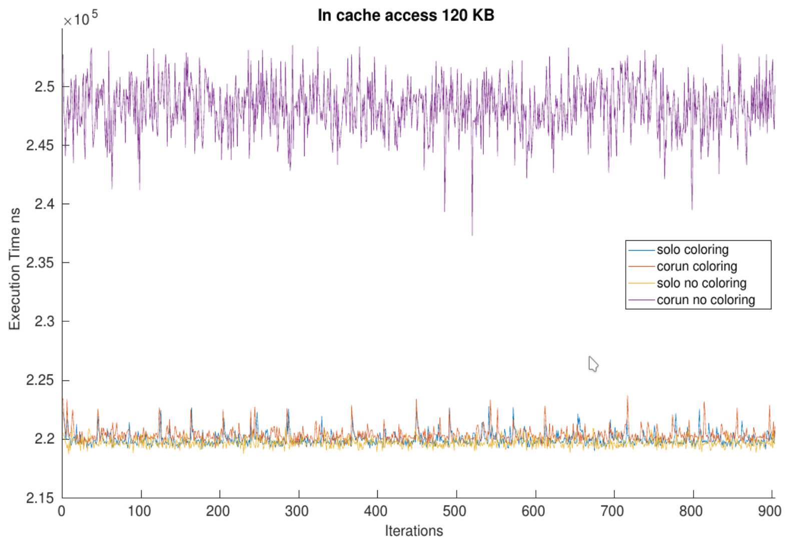


Figure 8.17: Execution times accessing 120 KB of memory.

- Core 2 is dedicated at the *Guest 0*;
- Core 3 is dedicated at the *Guest 1*;
- The DRAM is partitioned between the hypervisor and the guests.

In the scheduling example, *Guest 1* (seen as a black box) starts a long phase of access to DRAM memory, thus producing a considerable interference for *Guest 0*. From the point of view of real-time systems, the biggest problem is the lack of information on the amount of interference that a guest can potentially create. To handle this problem and get predictable guest behavior, a bandwidth reservation technique can be implemented, assigning to each guest a maximum number of memory accesses in every period. Such an approach allows giving guarantees concerning predictability even if the behavior of the guests remains unknown.

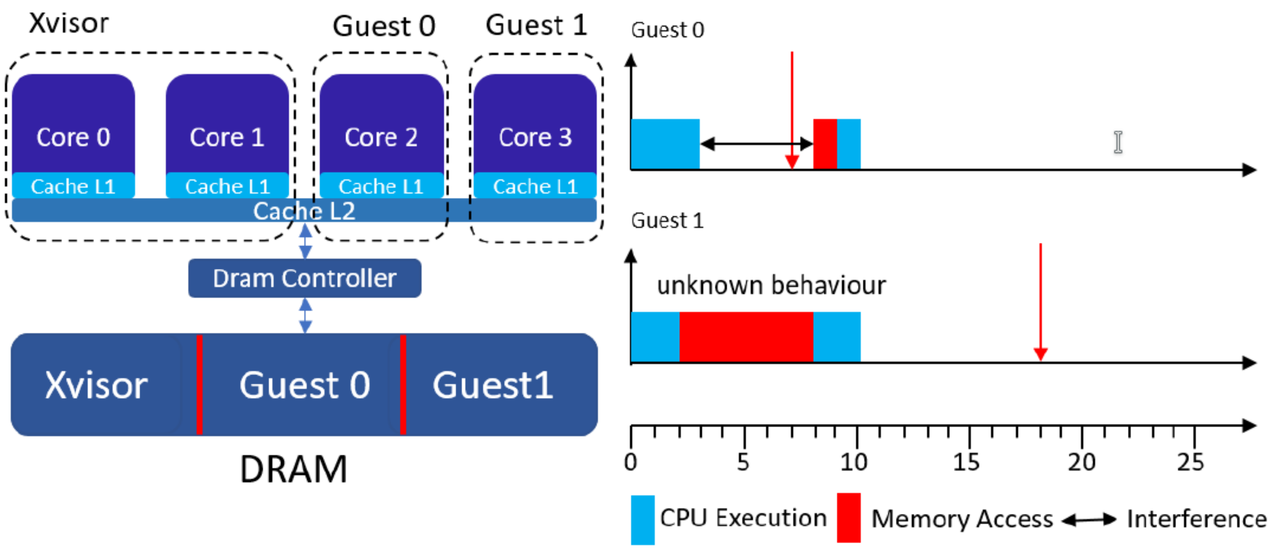


Figure 8.18: Memory access interference example.

Before-mentioned techniques have been explored for real-time operating systems, but often require more information on tasks. The first and perhaps the most important work on the memory bandwidth reservation that has been considered is MemGuard[37].

8.2.6.1 Memory Reservation Architecture Design

The bandwidth reservation system monitors all guests memory accesses and modifies their status. By making a high-level analysis, the components required for memory reservation are:

- A period and a budget for each guest, these will determine the band allocated to the guest;
- A counter to keep the remaining budget for each guest;
- A new state ”recharge” where the guest can be in;
- Edit and add actions during state transitions;
- Add new status transactions.

A crucial step concerns modifying the state machine that describes all the possible states of the guest Figure 8.19 shows a minimal state machine that takes into account all the actions needed to manage the memory reservation system.

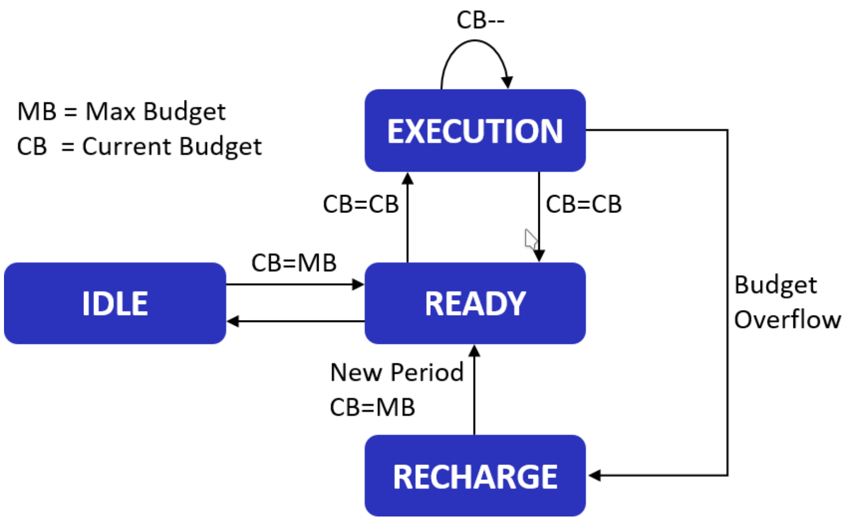


Figure 8.19: Guests State Machine for Memory Reservation

When a guest switches from *idle* state to *ready* state, it is necessary to initialize the current budget counter with the assigned budget value. In the transition from *ready* to *execution* and vice versa, the current budget counter is saved, its value is updated as the guest is in the *execution* state. The update is done by counting the memory accesses (in the reference architecture is used the ARM PMU feature). If the guest ends its current budget during the execution, it must be suspended until the beginning of the new period; in this case the guest switches from the *execution* state to the new *recharge* state. At the beginning of the new period, the guest current budget will be recharged to the maximum, and it will pass to the *ready* state if currently in *recharge* state. Saving the current value in the transition from *execution* to *ready* is very important because it is needed to supports the preemption of the guests; more precisely, when a guest is preempted the value of its current budget must be saved and then restored when it returns in the *execution* state. Figure 8.20 shows an example of execution in which a guest is preempted, and subsequently rescheduled within the same period, keeping the value of the budget always updated; in the last part of the scheduling, the guest is stopped until next period because it ends his current budget.

8.2.6.2 Implementation on Xvisor

The scheduler within Xvisor is a module that has the responsibility of choosing which VCPUs must run on the physical CPUs. The selection of VCPUs is done by extraction from ordered priority queues,

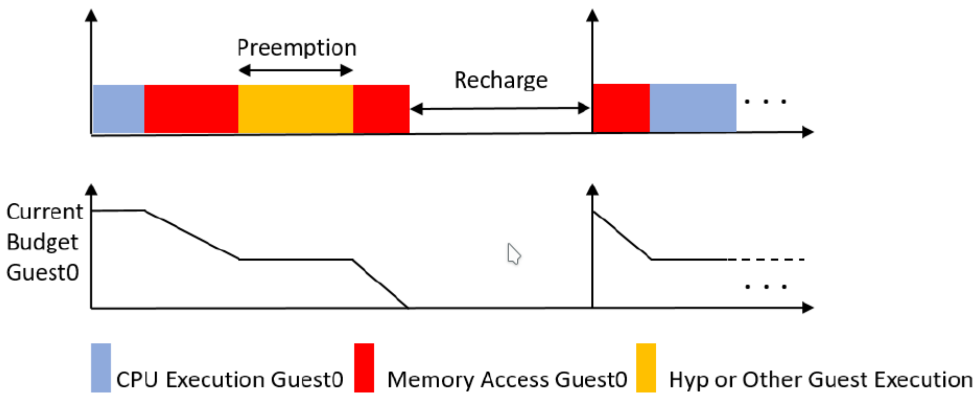


Figure 8.20: Memory Reservation Scheduling Example.

which are sorted through scheduling policies managed independently of the scheduler module, thus making the scheduler and the scheduling algorithms independent. One of the main tasks of the scheduler module is to ensure that each VCPU evolves its state by respecting the state machine shown in Figure 8.19. Notice that in Xvisor the entity that is scheduled and runs on a physical CPU is the VCPU and not Guest. I Xvisor there are two types of VCPUs, and those related to the guests are the Normal VCPUs. The status of each guest is then linked to the status of its VCPUs. A VCPU also has an affinity parameter by which can be indicated in which physical CPUs it is allowed to run regardless of the scheduling algorithm. In the proposed approach, to each guest is assigned only one VCPU that can run only on a dedicated physical CPU; in this way, there is no need to consider the scheduling algorithm because each guest has a dedicated physical CPU. However, the designed memory reservation system is entirely independent of the scheduling algorithm.

VCPUs are represented through a struct that store the current state of the VCPU, a copy of the physical CPU state, and more information about statistics and scheduling. The scheduler module has two main components represented by the `vmm_scheduler_state_change` and `vmm_scheduler_switch` functions. The first function has the task of handling all VCPUs' state transitions, except those between Idle and Running states. VCPU state changes are required by the manager module that is in charge of managing the system. When a VCPU state change is required, this function must ensure that the transaction is one of those mapped in the state machine and perform all necessary steps during the state transition.

The second function is called periodically through a timer event and handles the state transition from ready to running and vice versa. In this case, it will check which VCPU is first in the queue to be scheduled and, if it is different from the current one, switches between the current and the first one in the queue. During the state transition, the function will have to save the status and statistics of the descheduled VCPU and reload all the data of the new VCPU running on the physical CPU.

8.2.6.2.1 ARM PMU Support

ARM Cortex-A processors include dedicated logic to count different operations executed by the processor at runtime. This type of hardware allows us to perform accurate statistics and is often used to perform debug checks or performance measures. This hardware is called Performance Monitoring Unit (PMU). The Cortex A7 PMU provides four programmable counters for counting any of the events enabled in that particular processor, and a counter dedicated to clock cycles.

In multicore architecture, each core has a dedicated PMU system. The procedure for programming and starting PMU counters is shown on the right side of Figure 8.21, while the left side shows how to read the counters.

The PMU can also be programmed to launch an interrupt when one of the counters overflows; this feature is essential because it allows performing actions when the interrupt arrives and can be exploited for the memory reservation system. Xvisor did not have support for PMUs, so anew sub-

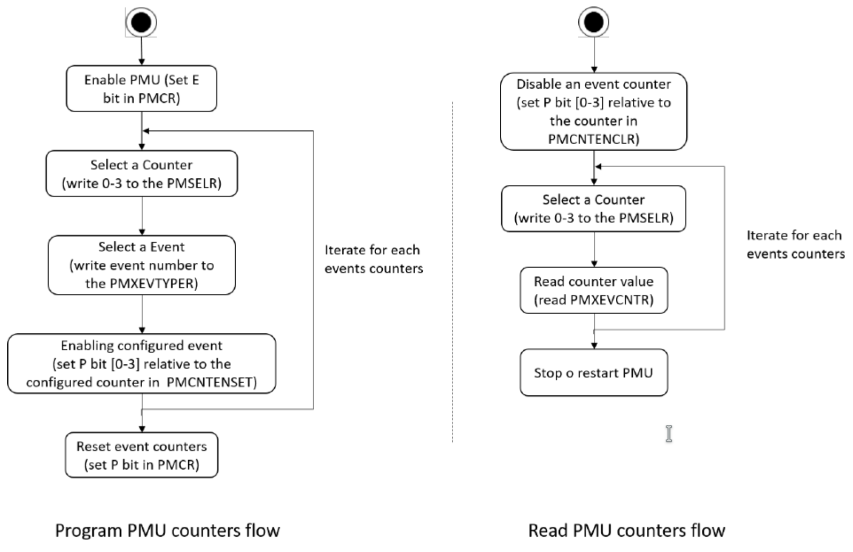

Enable PMU (Set E bit in PMCR)
Select a Counter (write 0-3 to the PMSELR)
Select a Event (write event number to the PMXEVTYPER)
Enabling configured event (set P bit [0-3] relative to the configured counter in PMCNTENSET)
Reset event counters (set P bit in PMCR)
Iterate for each events counters
Program PMU counters flow
Disable an event counter (set P bit [0-3] relative to the counter in PMCNTENCLR)
Select a Counter (write 0-3 to the PMSELR)
Read counter value (read PMXEVCNTR)
Stop o restart PMU
Iterate for each events counters
Read PMU counters flow

Figure 8.21: PMU setting/reading flow.

system to manage the PMU has been designed and implemented (counters configuration, counters reading and interrupt handling). First of all, at the boot, Xvisor must execute the initialization that links the provided handler to the interruptions generated by the PMU. This initialization must be done by each core because the PMUs interrupts are private, for each core, and therefore need a dedicated handler.

Once the handler for PMU interrupt has been initialized, a set of functions has been provided to automate the steps for managing the PMU, in particular:

- `u32 _read_cpu_counter(int r)` reads and returns the value of the counter identified by parameter `r`;
- `void _setup_cpu_counter(u32 r, u32 event, u32 cnt)` allows to set the counter identified by parameter `r` to count the events identified by the `event` code, the counter will be initialized with the value of the *cnt* parameter and the count will start from this value;
- `bool start_pmu_for_vcpu(struct vmm_vcpu *vcpu)` allows to execute all steps to initialize the PMU and start the count of memory access of the `vcpu`;
- `bool stop_pmu_for_vcpu(struct vmm_vcpu *vcpu)` stops the PMU by reading counter values and updating the `vcpu` accordingly.

With the above functions is possible to count the memory accesses of each VCPU, saving the values when needed and obtaining an interrupt when the VCPU exceeds a certain threshold.

8.2.6.2.2 Memory Reservation System implementation on Xvisor

To implement the memory reservation system within Xvisor, parameters have been added to the VCPU structure. They are initialized using DTS within the manager module, and a set of scheduler extensions have been done to obtain a new state machine that implements the behavior shown in Figure 8.19. The new parameters in the VCPU structure are as follows:

- `u32 mbudget` This variable stores the budget assigned to the VCPU that is the number of allowed memory accesses in a period;
- `u32 mactual_budget` This variable holds the current value of memory accesses made by the VCPU, and is necessary when a VCPU is preempted, and residual value extracted from the PMU count has to be saved;

- `u64 mperiod` This variable stores the period allocated to each VCPU;
- `bool mrecharge` This variable indicates when the VCPU is in the recharge state;
- `struct vmm_timer_event mrecharge_evt` This timer is needed to start a recharging event each period, and requires a dedicated handler.

The initialization code must take the `mperiod` and `mbudget` values from the DTS file and initialize the relative parameter using those values. The `actual_budget` parameter must be set to the *mbudget* value (max level). The `mrecharge` parameter must be set to false because the VCPU is not in *recharge* state. In this way, we can change the bandwidth assigned to each guest without having to make changes to the Xvisor code. The initialization just described is performed for each normal VCPU in the system.

Figure 8.22 shows the new Xvisors state machine with all the transition and the states needed to support the memory bandwidth reservation system, the initialization described above is done in the transaction highlighted with the number (1).

In the transition from the *RESET* state to the *READY* state (Figure 8.22(3)), the timer event parameter added to the VCPU is initialized. The timer is configured to raise an interrupt periodically, and the value of the period is stored in the `mperiod` parameter. Moreover, the timer is linked to a specific handler that will be called when the interrupt is raised. The initialization of the timer must specify the core in which the handler will be executed, which is the physical core assigned to the VCPU in the current implementation. The `mactual_budget` parameter is recharged to the maximum value, and the timer is started. Viceversa, the transaction from the *READY* state to the *RESET* state (Figure 8.22(2)) is more simple and only stops the timer event of the VCPU.

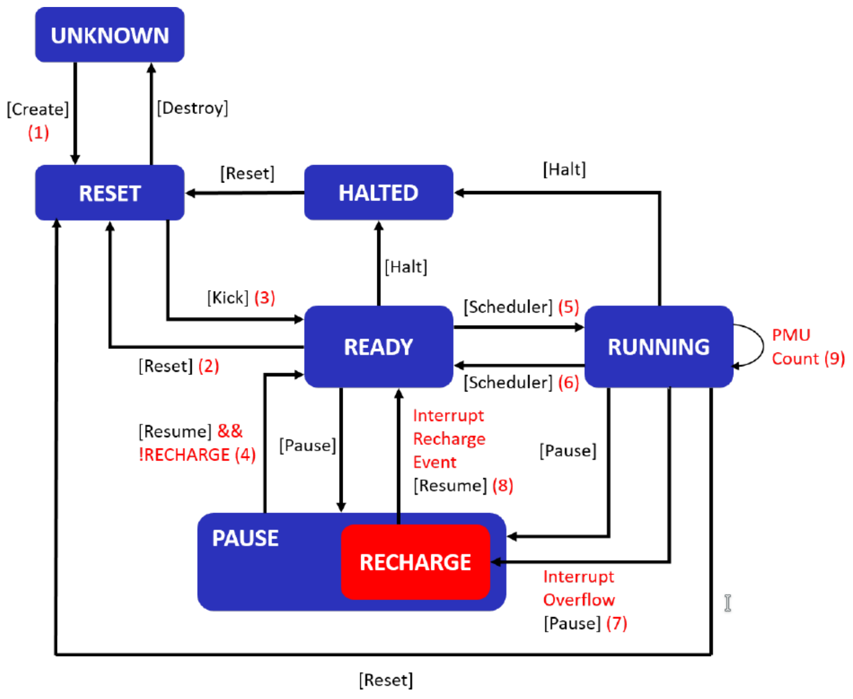


Figure 8.22: New Xvisor State Machine.

The interrupt handler for the recharge event generated by the timer have to do several tasks to recharge the VCPU budget and make a state change if necessary; the VCPU to be handled will be passed to the handler via a timer structure parameter. Specifically, the operations performed by the handler are the following:

- Extract the VCPU structure from the handler parameter;

- Acquire the lock on VCPU memory reservation parameters;
- Reload the current budget value to the maximum value;
- Set to false the variable `mrecharge`, this is needed because the VCPU must be put in the *READY* state if is currently in the *RECHARGE* state (see Figure 8.22(8)));
- Enable the flag that allows the VCPU to go in the WFI mode (Wait for Interrupt);
- Release the lock on VCPU memory reservation parameters;
- After finishing the operations on the parameters, restart the timer to count a new period, which is the only way to have periodic events within Xvisor;
- At this point, a *Resume* operation is performed on the VCPU, so it would be put to the *READY* state if it was suspended in the *RECHARGE* state awaiting a budget recharge (Figure 8.22(8)). In both cases, this operation will restart the PMU count, so that the new budget value is taken into account;

The *RECHARGE* state has been implemented as a sub-state of the *PAUSE* state and is mapped via the *mrecharge* variable added to the VCPU structure, so when a VCPU is in the *RECHARGE* state will also be in the *PAUSE* state. This design choice allowed to minimize the changes of the Xvisor scheduling structure. As a drawback, a check must be performed each time a *RESUME* operation is invoked for a VCPU that it is not in *RECHARGE* state (see Figure 8.22(4)); otherwise, the status change will not be allowed. The only way to exit the *RECHARGE* state is when the recharge event handler described above is executed. When a VCPU passes from the *READY* state to the *RUNNING* state; the PMU is enabled to count the memory accesses made by the VCPU during its execution. Also, the PMU needs to be configured to launch an interrupt to handle the suspension of the execution when the current budget is consumed. As described in subsection 8.2.6.2.1, the PMU can only interrupt when one of the counters goes into overflow; for this reason, the counter must be started from a calculated value so that the counter goes into overflow when the current budget is finished. All the startup and programming operations of the PMU are made by the `start_pmu_for_vcpu` function, that does:

- Writes all the PMU registers to clear and reset all the counters;
- Sets the first counter to count data memory accesses (code DMACC 0x13), and sets the counter to a value computed whit this formula: `MAX_COUNTER_VALUE - vcpu->mactual_budget`;
- Writes the PMU register to clear overflow;
- Writes the PMU register to enable overflow;
- Writes the PMU register to enable PMU and Counters;

Viceversa, in the transition from the *RUNNING* state to the *READY* state (see Figure 8.22(6)), the PMU counter has to be stopped and the current budget must be updated with the value read from the counter, that is the residual memory accesses. The `stop_pmu_for_vcpu` function does all the necessary operations that are:

- Writes the PMU registers to stop all counters;
- Reads the counter value from the register and updates the current budget with the following formula: `MAX_PMU_COUNTER_VALUE - counters_value`;
- Writes the PMU registers to disable the entire PMU system.

When the VCPU is in the *RUNNING* state the transaction on Figure 8.22(9) indicates that the PMU is running and counting the memory accesses. The count does only in a hardware way, and all the updates of the parameters are stored only at changing state. The last transaction that must be described is the PMU interrupt overflow (Figure 8.22(7)), that means that the VCPU ended her budget so must be put in *RECHARGE* state. All the operations needed to change the state from *RUNNING* to *RECHARGE* are done in the handler linked to the PMU overflow interrupt. The handler must do the following operations:

- Reads the PMU register that indicates which of the counters suffered an overflow;
- Checks that the overflow is from the counter that counts the memory accesses; otherwise, the interrupt will not be handled. This check is necessary if the counters are used for other purposes.
- Writes the PMU register to clear the overflow;
- Disables the VCPU to go to WFI;
- Sets the `mrechage` variable to TRUE to indicate that the VCPU will be in the *RECHARGE* state up to the next period and will not be able to go in the *RUNNING* state;
- Sets to zero the current budget variable;
- Calls the pause operation for the VCPU that will be put in the PAUSE state with the `mrecharge` flag set, in this way the VCPU will be in the *RECHARGE* sub-state.

Figure 8.23 shows the flow diagram of the PMU interrupt handler.

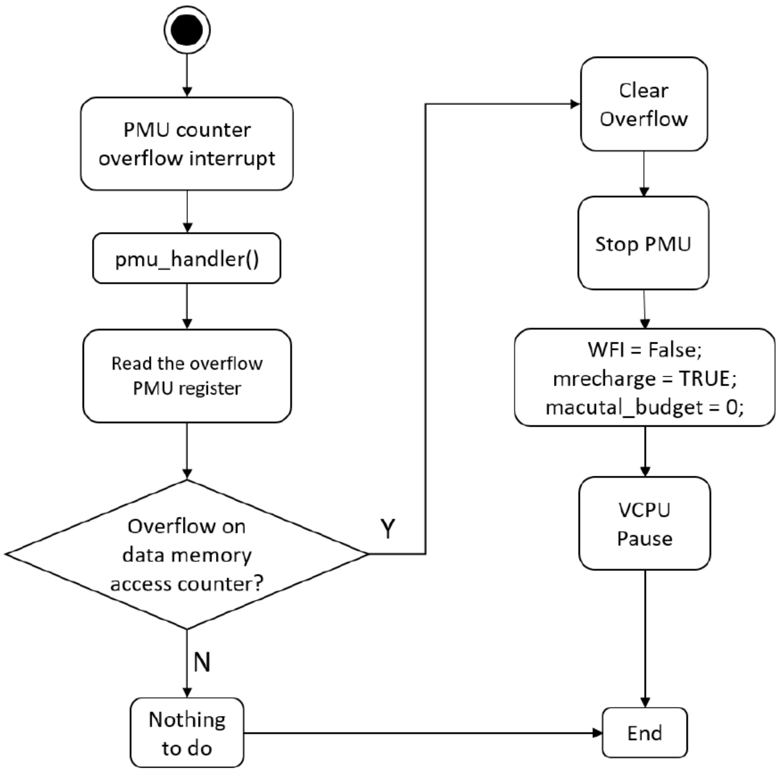


Figure 8.23: Flow diagram of the PMU interrupt handler.

8.2.6.3 Experimental Results on Memory Bandwidth Reservation

To test the results obtained by using the memory reservation several experiments have been conducted with the same conditions already listed in subsection 8.2.5.2. In this case, the test code accesses (performing a write operation) a variable memory area in the range from 512 KB to 10240 KB, thus being sure to use DRAM memory. In Figure 8.24 and Figure 8.25 are shown the AVG and MAX execution times to access a variable amount of memory.

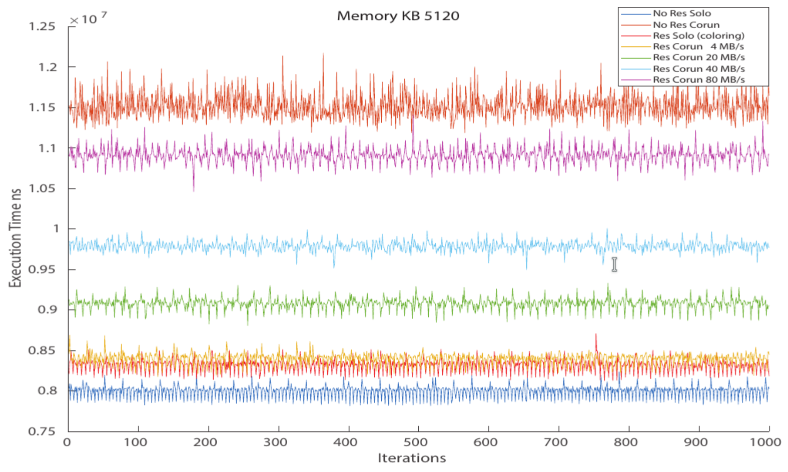


Figure 8.24: AVG execution times comparison varying the bandwidth.

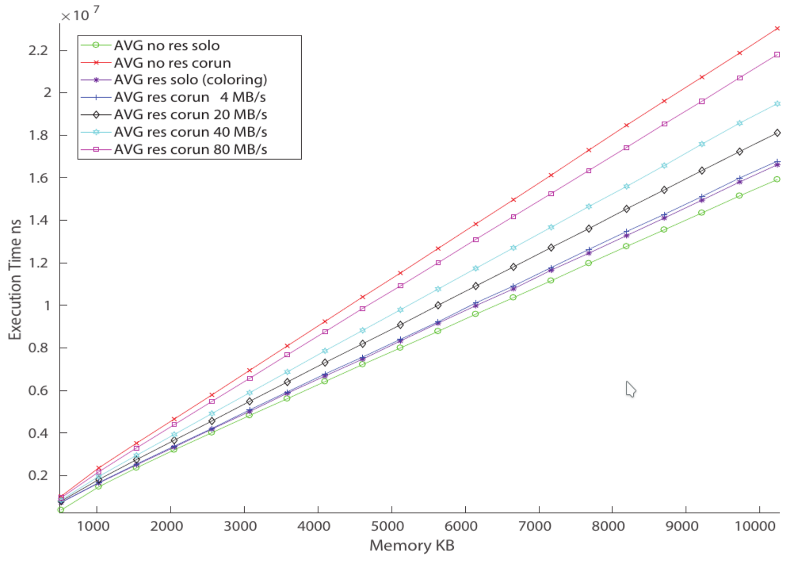


Figure 8.25: MAX execution times comparison varying the bandwidth.

The best case is where there is only one guest using Xvisor without coloring; this is quite obvious because the guest has a larger amount of cache available and no interference. As in the experiments on cache coloring, the worst case is still Xvisor without coloring running two guests due to the effects of suffered interference (up to +48%).

Using Xvisor with coloring and memory reservation, and limiting the maximum bandwidth of the interfering guest, the interference can be limited according to the assigned bandwidth. In this way, it is possible to estimate how much the other guests could interfere and it will be possible to tune such an interference according to the performance and temporal constraints of the real-time guest.

Finally, Figure 8.26 shows the execution times of various iterations accessing a 5120 KB memory area. It presents how the bandwidth assigned to the interfering guest affects the suffered interference.

The obtained results confirm that integrating memory reservation and cache coloring techniques into a hypervisor allows providing time guarantees to the guests.

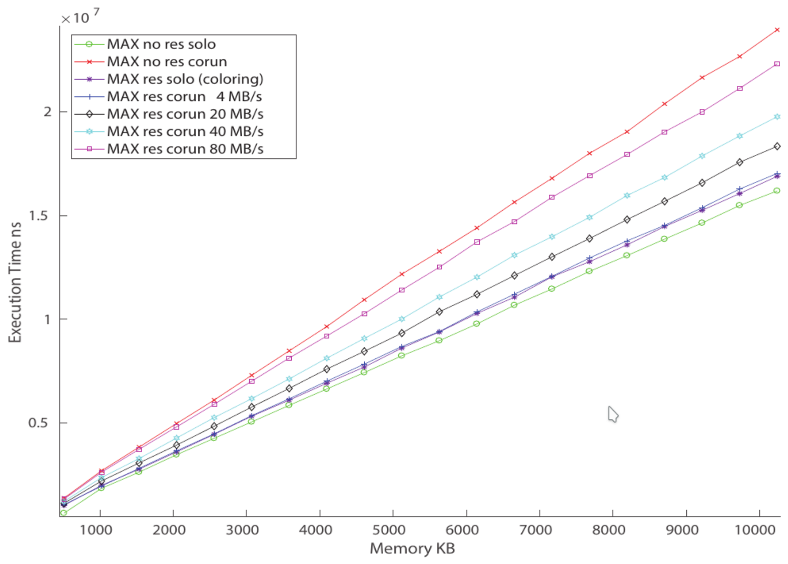


Figure 8.26: Comparison accessing 5 MB of memory varying the bandwidth.

Chapter 9 Summary and Conclusion

This deliverable has presented the final implementation work in WP4. The work was not a straight path and the project had to take detours and conduct additional research. In the end, the consortium has been able to cover a very wide set of topics ranging from security, safety, spacial and temporal and temperature/energy separation, to RTEs to configure and enforce mixed-critical policies. The result has shown that achieving a fully integrated mixed-critical approach (from a CPU, to SoC, hypervisor, OS/RTOS, extensions and RTEs, unto a user-land applications) is still a challenge. Despite that experience the project has clearly pin-pointed obstacles and made a big step forward providing the basic blocks and integration of those basic blocks into a functional mixed-critical system.

Glossary

BB-RTE	Budget-Based RunTime Engine	III, IV, VI, 23–25, 27, 30, 31, 34, 35, 40
BiQuad	bi-quadratic distributed control system application	VI, 43, 44
eDRON	embedded Directed Rotodrone Operated Network	VI, VIII, 44, 45
FMS	Flight Management System	VI, VIII, 40–44
METrICS	Measure Environment for TIme-Critical Systems	VI, 29, 30
PMC	Performance Monitor Counters	25, 27, 28

Bibliography

[1] Open Virtualization's SierraVisor and SierraTEE. `www.openvirtualization.org/open-source-arm-trustzone.htmltrustzone.html`.

[2] Luca Abeni and Giorgio Buttazzo. Integrating multimedia applications in hard real-time systems. In *Real-Time Systems Symposium, 1998. Proceedings. The 19th IEEE*, pages 4–13. IEEE, 1998.

[3] RISC Advance. Machines ltd. *An introduction to Thumb*, 1995.

[4] SYSGO AG. PikeOS 4.2: RTOS with hypervisor-functionality, March 2017.

[5] Gene M. Amdahl. Validity of the single processor approach to achieving large scale computing capabilities. In *Proceedings of the Spring Joint Computer Conference*, pages 483–485, Atlantic City, April 1967. ACM.

[6] ARM. Arm information center.

[7] ARM Limited. Fixed Virtual Platforms: FVP Reference Guide, Nov 2015.

[8] ARM Limited. Versatile Express: 64 Bit Juno r2 ARM Development Platform, Nov 2015.

[9] Thomas G. Baker. Lessons learned integrating COTS into systems. In *Proceedings of the First International Conference on COTS-Based Software Systems*, ICCBSS '02, pages 21–30, 2002.

[10] Jingyi Bin, Sylvain Girbal, Daniel Gracia Perez, Arnaud Grasset, and Alain Merigot. Studying co-running avionic real-time applications on multi-core cots architectures. *Embedded Real Time Software and Systems conference*, Feb 2014.

[11] Gabriel Fernandez, Javier Jalle, Jaume Abella, Eduardo Qui nones, Tullio Vardanega, and Francisco J. Cazorla. Resource usage templates and signatures for cots multicore processors. In *52nd Design Automation Conference*, DAC, 2015.

[12] Stuart Fisher. Certifying Applications in a Multi-Core Environment: The World's First Multi-Core Certification to SIL 4, 2013.

[13] Sylvain Girbal, Xavier Jean, Jimmy Le Rhun, Daniel Gracia Pérez, and Marc Gatti. Deterministic Platform Software for hard real-time systems using multi-core COTS. In *Proceedings of the 34th Digital Avionics Systems Conference*, DASC'2015, 2015.

[14] Sylvain Girbal, Daniel Gracia Pérez, Jimmy Le Rhun, Madeleine Faugère, Claire Pagetti, and Guy Durrieu. A complete toolchain for an interference-free deployment of avionic applications on multi-core systems. In *Proceedings of the 34th Digital Avionics Systems Conference*, DASC'2015, 2015.

[15] Sylvain Girbal, Jimmy Le Rhun, and Hadi Saoud. METrICS: a measurement environment for multi-core time critical systems. In *Embedded Real Time Software and Systems (under review)*, ERTS '18, 2018.

[16] GlobalPlatform. Globalplatform made simple guide: Trusted execution environment (tee) guide, 2016.

[17] John L. Gustafson. Reevaluating amdahl's law. *Commun. ACM*, 31(5):532–533, May 1988.

[18] Infineon, Ltd. *AURIX TC27x B-Step 32-Bit Single-Chip Microcontroller - User Manual*.

[19] Infineon, Ltd. *TriCore 32-bit TriCoreV1.6 Core Architecture 32-bit Unified Processor Core - User Manual v1.0*.

[20] International Electrotechnical Commission. IEC 61508: Functional safety of electrical, electronic, or programmable electronic safety-related systems, 2011.

[21] International Organization for Standardization (ISO). ISO 26262: Road Vehicles Functional Safety, 2011.

[22] Xavier Jean, David Faura, Marc Gatti, Laurent Pautet, and Thomas Robert. Ensuring robust partitioning in multicore platforms for ima systems. In *Digital Avionics Systems Conference (DASC), 2012 IEEE/AIAA 31st*, pages 7A4–1. IEEE, 2012.

[23] Raimund Kirner and Peter Puschner. Obstacles in worst-case execution time analysis. In *Proceedings of the 11th IEEE Symposium on Object Oriented Real-Time Distributed Computing*, pages 333–339, 2008.

[24] Kevin D Kissell. Mips16: High-density mips for the embedded market. In *Salon des solutions informatiques temps réel*, pages 559–571, 1997.

[25] Angeliki Kritikakou, Claire Pagetti, Christine Rochange, Matthieu Roy, Madeleine Faugère, Sylvain Girbal, and Daniel Gracia Pérez. Distributed run-time WCET controller for concurrent critical tasks in mixed-critical systems. In *Proceedings of the 22th International Conference on Real-Time and Network Systems (RTNS'14)*, pages 139–148, 2014.

[26] John P Lehoczky. Enhanced aperiodic responsiveness in hard real-time environment. In *Proceedings of IEEE Real-Time Systems Symposium'87*, pages 261–270, 1987.

[27] E. Mezzetti and T. Vardanega. On the industrial fitness of wcet analysis. In *Proceedings of the 11th International Workshop on Worst Case Execution Time Analysis (WCET2011)*. 2011.

[28] Jan Nowotsch and Michael Paulitsch. Leveraging multi-core computing architectures in avionics. *European Dependable Computing Conference*, pages 42–52, 2012.

[29] Anup Patel, Mai Daftedar, Mohmad Shalan, and M. Watheq El-Kharashi. Embedded hypervisor xvisor: A comparative analysis. 2015.

[30] Radio Technical Commission for Aeronautics (RTCA) and EURopean Organisation for Civil Aviation Equipment (EUROCAE). DO-297: Software, electronic, integrated modular avionics (IMA) development guidance and certification considerations.

[31] R. Russell. Virtio pci card specification v0.9.5 draft. 2012.

[32] Brinkley Sprunt. The basics of performance-monitoring hardware. *Micro, IEEE*, 22(4):64–71, 2002.

[33] Brinkley Sprunt, Lui Sha, and John Lehoczky. Aperiodic task scheduling for hard-real-time systems. *Real-Time Systems*, 1(1):27–60, 1989.

[34] Jay K. Strosnider, John P. Lehoczky, and Lui Sha. The deferrable server algorithm for enhanced aperiodic responsiveness in hard real-time environments. *IEEE Transactions on Computers*, 44(1):73–91, 1995.

[35] Wikipedia. Earliest deadline first scheduling. https://en.wikipedia.org/wiki/Earliest_deadline_first_scheduling.

[36] Wikipedia. Fixed Virtual Platforms FVP Reference Guide. https://en.wikipedia.org/wiki/Virtualization.

[37] Heechul Yun, Gang Yao, Rodolfo Pellizzoni, Marco Caccamo, and Lui Sha. Memguard: Memory bandwidth reservation system for efficient performance isolation in multi-core platforms. In *Real-Time and Embedded Technology and Applications Symposium (RTAS), 2013 IEEE 19th*, pages 55–64. IEEE, 2013.