D5.4 Evaluation of the operating systems and hypervisors

Version 0.3

Documentation Information

Contract Number	871669
Project Webpage	https://www.ampere-euproject.eu/
Contractual Deadline	30.06.2023
Dissemination Level	Public (PU)
Nature	DEM
Authors	Jan Rollo (SYS) Ida Maria Savino (EVI)
Contributors	Claudio Scordino (EVI) Luca Cuomo (EVI) Tommaso Cucinotta (SSSA) Gabriele Ara (SSSA) Alessandro Ottaviano (ETHZ) Nils Wistoff (ETHZ) Darshak Sheladiya (SYS) Andrej Kruták (SYS)
Reviewer	Luis Miguel Pinho (ISEP)
Keywords	Multi-OS, RTOS, Linux, hypervisor, ROS, RISC-V

AMPERE project has received funding from the European Union's Horizon 2020 research and innovation programme under the agreement No 871669.

Change Log

Version	Description Change
V0.1	Preliminary version
V0.2	Minor changes
V0.3	Internal Review

Table of Contents

Executive Summary

This deliverable aims at evaluating the performance of the software architecture developed in WP5 of the AMPERE project. In particular, the evaluation is provided for the hypervisor, operating systems and communication middleware developed in this WP.

1 Introduction

This deliverable aims at providing experimental results to evaluate the overall performance of the software architecture developed in WP5 (i.e. hypervisor, operating systems and communication middleware). The results are given for the different configuration options and technological improvements developed in the Work Package.

For running the experiments, we have used the AMPERE reference platform consisting of a Xilinx ZCU102 evaluation board [1]. The platform is based on a Xilinx UltraScale+™ MPSoC device [2] which includes a quad-core ARM® Cortex-A53, a dual-core Cortex-R5 real-time processors, a Mali-400 GPU and a programmable FPGA. On the FPGA, we have synthesized a 64-bit CVA6 RISC-V soft-core [3] (also known as "Ariane"). Although the AMPERE project has selected also another platform (i.e. NVIDIA Jetson Xavier AGX), the performance measurements have been performed only on the Xilinx platform. The reason is twofold: the Xilinx platform can be considered a more complete platform (because it also contains a programmable FPGA and Cortex-R processors); moreover, NVIDIA never provided the requested details for running PikeOS on its own platform.

The software architecture developed in AMPERE includes the PikeOS hypervisor and ElinOS, a Linux distribution executed as a guest VM of the PikeOS hypervisor [4, 5]. Both components are commercial products by partner SYSGO. Furthermore, the sofware architecture includes Erika RTOS [6], a real-time operating system designed according to the OSEK/VDX and AUTOSAR Classic standards [7]. In the context of AMPERE project, Erika RTOS has been ported to run as a PikeOS guest on a Cortex-A processor and to run on a RISC-V processor synthesized on the FPGA, integrating also the Micro-ROS protocol [8, 9] to communicate with the Linux OS counterpart.

This deliverable provides an empirical evaluation of the various components under different configurations. For example:

- the performance of the Erika RTOS has been measured when executed on different processors (namely, Cortex-A, Cortex-R and RISC-V);
- the performance of the middleware for Inter-OS communication has been measured for communication towards both Cortex-A and RISC-V;
- the scheduling latency of PikeOS Linux VM has been measured with and without the usage of the PREEMPT_RT real-time patch [10].

The document is organized as follows:

- Section 2 contains an empirical evaluation of PikeOS hypervisor. It contains an evaluation of the communication latency between applications running on different PikeOS guests. It also contains a summary of the assurance on PikeOS that it enforces separation.
- Section 3 shows the performance evaluation of the Erika RTOS running on different processors and with different levels of hardware/software optimization. Furthermore, it contains the performance evaluation for Inter-OS communication, where OSes run on different processors (i.e., Linux on Cortex-A and Erika RTOS on RISC-V CPU).
- Section 4 contains an empirical evaluation of the Linux OS. More in detail, it contains the analysis of scheduling latency in Linux-based PikeOS guest. Furthermore it contains the evaluation of the overhead introduced by the virtualization on real-time applications, organized in Directed Acyclic Graphs (DAGs), running on a Linux-based guest.

2 Hypervisor evaluation

This section provides an empirical evaluation of the various components under the PikeOS hypervisor.

Section 2.1 contains the evaluation of the communication latency between applications running on different PikeOS guests. Communication is based on the inter-guest communication mechanisms provided by PikeOS and the DDS-XRCE [9, 11] protocol, specifically designed for resource-constrained systems. Section 2.2 details the capability of PikeOS hypervisor to provide separation of partitions.

2.1 Inter-OS communication evaluation

This section provides experimental results on the communication latency between applications running on different PikeOS guests. With reference to the AMPERE scenario in Figure 1, both Linux and Erika RTOS run as virtualized guests of the PikeOS hypervisor.

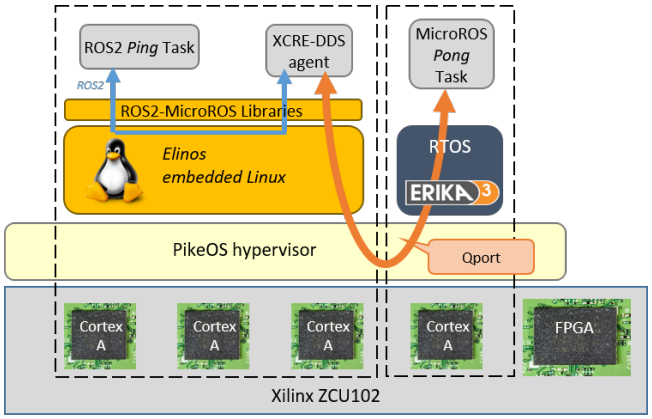

Figure 1: Inter-OS communication in PikeOS-based virtualized environment.

The inter-OS communication between an application running on Linux and one running on Erika RTOS is based on DDS-XRCE [8, 9], a DDS protocol specifically designed by OMG for resource-constrained systems. In particular, in the context of the AMPERE project we have integrated eProsima's Micro XRCE-DDS stack on Erika RTOS . In this client-server protocol, the devices (clients) communicate with an XRCE-DDS Agent (server) which provides the intermediate bridging service towards the DDS Data Global Space [11].

Data has been exchanged through a inter-guest communication provided by PikeOS (e.g., QPort). In order to minimize the communication latency between the OSes, both the Micro-ROS stack implemented in Erika RTOS and the XRCE Agent running on Linux exchange data only through QPort-based mechanisms. More in detail, a QPort was used to send data from the Linux agent to Erika RTOS application and another Qport was used to send data in the opposite direction.

The communication latency has been evaluated through a "ping-pong" application that measured the round-trip time from Linux to ERIKA and back to Linux. More in detail, the "ping" application, running on the Linux guest, publishes a message on the Ping topic. The XRCE Agent forward the messages to the topic subscribers. When the "pong" application, running on ERIKA guest, receives the Ping messages, it publishes a new message on the Pong topic. Such message is forwarded by the XRCE Agent to all the subscribers of such topic. As shown in Figure 2, the communication latency ΔT corresponds to the span of time between the time a Ping message is transmitted and the time a Pong message is received in the "ping" application. It includes the delay due to the periodic engine of the Micro-ROS framework.

The experimental results, contained in Table 1, show the minimum, average and maximum communication latencies, that include the delay due to the Micro-ROS framework and PikeOS Inter-OS communication mech-

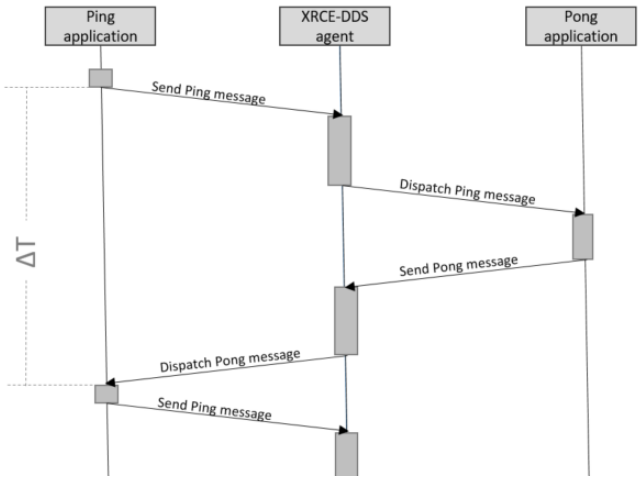

Figure 2: Communication latency in the "ping-pong" application.

anism (i.e., QPort). Although the minimum value is slighlty higher than the delay of an inter-node communication using ROS2 on Linux, the other values are comparable (or even lower, in the case of the maximum value) than the time needed on Linux for such kind of communication.

We can therefore conclude that the movement of part of the computation to ERIKA has not introduced a significant penalty in terms of communication. On the other end, it has allowed to isolate and run such computation on a properly safety-critical RTOS.

ΔT	Experimental results
Min	1.015 msec
Average	1.185 msec
Max	2.079 msec

Table 1: Ping-Pong time communication.

2.2 Separation property

The PikeOS hypervisor provides the capability to provide separation of partitions. For instance, it is publicly stated in the PikeOS Common Criteria for IT Security Evaluation (CC) scope document (called "Security Target" according to the CC [12]) that *"The TOE (target of evaluation, that is the evaluated PikeOS system) is a Separation Kernel, which allows to effectively separate multiple applications running on the same platform from each other. Such applications can range from small bare-metal programs up to entire Operating Systems. Non-privileged applications may be malicious, and even in that case the TOE ensures that malicious applications are neither capable of harming other applications nor the TOE itself.*

Separation Kernels aim to establish a degree of isolation between the applications on a single system (e.g., a hardware platform) which, in terms of security, is comparable to running the application executables on physically separate platforms [13]. However, Separation Kernels also provide communication facilities that allow the applications to interact with each other, if configured by the Integrator.

The TOE includes a wide range of additional features and functionalities such as direct memory access, process control, memory management, different communication services and more. Together with the real-time capability of the TOE, this allows to build and operate embedded systems in areas with a high demand towards security and safety such as automotive, avionics, medical devices, industrial and railway applications.

(...)

SYSGO defines separation as follows: The TOE separates partitions by managing their accesses to and usage of resources, such as memory, devices, processors, and communication channels, as defined by the configuration. Isolation of a partition is the absence of communication with other partitions, except partitions hosting the components implementing the system API, when no communication channels or shared resources between the partition and other partitions are configured. Isolation is a special case of separation.

Additionally, the TOE has the characteristics of an embedded real time operating system. Thus, the partitioning is configured statically and the TOE does not include typical desktop operating system services (e.g. user login, printer drivers). The TOE will typically be installed and operated on a hardware platform suitable for embedded systems.

The major security services provided by the TOE are:

- *Separation in space of applications hosted in different partitions from each other and from the PikeOS Operating System according to the configuration data,*
- *Separation in time of applications hosted in different partitions from each other and from the PikeOS Operating System according to the configuration data,*
- *Control of information flows between applications hosted in different partitions via assigning to the partitions communication objects and access rights to those.*
- *Management of the TOE (e.g. system partition API) and the TOE data (e.g. threads, tasks).*

”

Analyses: In order to show the separation ability of PikeOS across partitions, that is e.g., the performance of a software running within a partition when on the other partitions nothing is running, vs when something is being actively computing on the other partition, different analytical automatic/semiautomatic/manual analyses are done: During the CC evaluation we had to give a structured argument, why these separation properties hold for security. Similarly for safety, there is a PANA (Partition Analysis). It shows that partitions are separated. In addition each PikeOS service is analyzed regarding its WCEP (Worst Case Execution Path) including each delay which need to be considered. The analysis is available as part of safety certification artifacts.

Tests: On the test side, there are numerous functional tests of separation mechanisms and there is the PikeOS High Level Longrun testsuite, which fulfills the test objective in question to ensure separation. Furthermore there is a test set, which contains stress test partitioning on multi processor platforms. SYSGO also runs fuzz testing. Tests are part of the evidence presented for the DO-178C, ISO 26262, IEC 61508, Common Criteria certifications, as e.g. documented in [14]. These tests are run on target architectures (e.g. X86, ARM, PowerPC) in regular intervals.

In AMPERE, the partitions e.g. on the Erika setup (see AMPERE scenario in Figure 1) are (1) ELinOS and (2) Erika/ROS as guests and for this particular case the general separation property also holds. Note that in the AMPERE scenario, intentionally, there is a configured QPort-based communication channel between the partitions, this is a good example of having an intentional controlled information flow (the control of the of the QPort is done by the applications), while maintaining separation otherwise.

3 Erika RTOS evaluation

This section details the performance evaluation of the Erika RTOS running on the different processors. For running the experiments, we have used the processors available on the Xilinx ZCU102 evaluation board [1]: the ARM Cortex-A53, the ARM Cortex-R5 and a CVA6 "Ariane" RISC-V soft-core synthesized on the FPGA [3]. The activity on RISC-V was included in the DoW even if not requested for executing the two specific use-case of AMPERE. The momentum that RISC-V technology is experiencing (also and especially in Europe) has pushed the consortium to look ahead and get prepared for the next technology transition which is expected to impact various application domains, including automotive.

Section 3.1 shows the performance evaluation of the Erika RTOS running on different processors and with different levels of hardware/software optimization.

Section 3.2 contains the performance evaluation for Inter-OS communication, where OSes run on different processors. In particular, we consider the scenario where Linux OS runs on ARM Cortex-A whereas Erika RTOS runs on the RISC-V CPU.

3.1 Erika RTOS performance evaluation

Section 3.1.1 describes experimental results on different processors. Section 3.1.2 details the RTOS optimizations to obtain better performance in all the tested processors, whereas Section 3.1.3 describes the improvement of the hardware design for the synthetized CPU on the FPGA.

3.1.1 Erika RTOS on RISC-V

3.1.1.1 Baseline RISC-V description

On the FPGA, we have synthesized a CVA6 "Ariane" RISC-V CPU [3], an open-source design provided by partner ETHZ. The processor is a 6-stage, single-issue, in-order CPU implementing the I, M, A and C extensions of the 64-bit RISC-V instruction set (RV64IMAC). It implements a Translation Lookaside Buffer (TLB) to accelerate address translations from the virtual to the physical domain, and branch-prediction through a branch target buffer (BTB) and branch history table (BHT).

The soft-core was synthesized on the target FPGA at 50 MHz with 32 kiB and 16 kiB data and instruction, respectively. CVA6 hosts three level sensitive interrupt signals as from the RISC-V Privileged Specifications [15]:

- machine-mode timer interrupt,
- machine-mode software interrupt (inter-processor interrupt)
- machine-mode/supervisor-mode external interrupts

The machine timer and machine software interrupt pending registers (`mtip`) and `msip` respectively) are provided by a Core Local Interruptor (CLINT) hardware Intellectual Property (IP), which generates one interrupt for each hardware thread (hart, a RISC-V execution context). While `mtip` generates timer interrupts with a specific frequency, `msip` handles communications among processors by making interrupt request through writing/reading in dedicated memory-mapped registers. The first 12 interrupts identifiers are reserved for timer, software and external interrupts in machine (M), supervisor (S) and user (U) privilege modes. Other interrupt entries up to XLEN (for a RV64 processor such as CVA6, XLEN=64) [15] are platform specific and referred to as *local* interrupts. Finally, the machine external and supervisor external interrupt pending registers `meip/seip` bring the information from *external* devices to the hart. The Platform Local Interrupt Controller (PLIC) [16] provides centralized interrupt prioritization and routes shared platform-level interrupts among multiple harts via the `meip/seip` interrupt signals.

3.1.1.2 What we measured

Since Erika RTOS is a real-time operating system designed according to the OSEK/VDX and AUTOSAR Classic standards, the programming paradigm is "run-to-completion" and the configuration (e.g. number of tasks) is statically defined at compile time. In this type of operating systems, the Interrupt Service Routines (ISR) are divided in two categories:

- ISR1: High-priority low-overhead routines, that cannot call syscalls;
- ISR2: Priority-based routines which could imply a rescheduling once finished.

The real-time performance of the Erika RTOS has been measured through an existing benchmark [17] that measures the time needed by the RTOS for performing a set of critical scheduling activities (e.g., task activation time, task exit time, ISR call time, etc.). The test suite also allows to benchmark the latency of the two types of interrupt service routines. The tested functions are, namely:

- `act`: activates a higher priority task and measures how long it takes to start its execution.
- `actl`: activates a low priority task and measures how long it takes to return to the caller.
- `intdisable`: measures the time needed for disabling all interrupts.
- `intenable`: measures the time needed for enabling all interrupts.
- `isrentry`: measures the time elapsed between the occurrence of an interrupt and the execution of the related ISR1 handler.
- `isr2entry`: measures the time elapsed between the occurrence of an interrupt and the execution of the related ISR2 handler.
- `isrexit` : measures the time elapsed between the end of an interrupt handler and when the task previously running resumes execution.
- `istentry`: measures the time elapsed between the end of an interrupt handler and the execution of the task activated by such interrupt handler.
- `istexit`: measures the time elapsed between the end of a task handling an interrupt and when the task previously running resumes execution.
- `terml`: measures the time needed for terminating a task and switching to a lower priority one.

3.1.1.3 Erika RTOS baseline performance

When porting the Erika RTOS on RISC-V we have taken inspiration from the previous FreeRTOS optimization [18]. However, CVA6 is a 64-bit CPU, and therefore it has not been possible to use the Embedded ABI [19] to shorten interrupt latency by reducing the number of caller-saved registers. Similarly, it has not either been possible to replace the standard integer instruction set with a reduced-length instruction set explicitly designed for embedded systems (e.g. RV32E). We have therefore only optimized interrupt handling, by emulating the local Interrupt Priority Levels (IPL) through an array statically generated by the OS tools.

Times have been measured in processor cycles, measured through the mcycle `CSR` register.

To evaluate the performance, the same benchmark has been executed also on both the Cortex-R5 and the Cortex-A53 cores available on the ZCU102 board. For the Cortex-R5, the number of cycles have been measured through the `PMCCNTR` register. For the Cortex-A53, instead, cycles have been measured through the cycle counter register `PMCCNTR-EL0`:

```
__asm__ __volatile__ ("MRC p15, 0, %0, c9, c13, 0 \n" :  "=r" (cycles));
```

It is important to point out that, in case of Cortex-A, the RTOS has been run on top of a hypervisor according to the typical configuration used when running RTOSs on Cortex-A processors. This also allowed to have a direct comparison against the previous work [20]. The presence of the underlying hypervisor, however, implied some non-negligible latency to trap and re-inject interrupts to the guest RTOS. The possible interference from

Linux on shared hardware resources has been removed by putting the Linux kernel in panic mode through the following command:

```
$ echo c > /proc/sysrq-trigger
```

Table 2 reports the best-case and the worst-case number of the cycles measured over 100 consecutive runs.

Test name	CLINT RISC-V		Cortex-R5		Cortex-A53	
	Min	Max	Min	Max	Min	Max
act	355	1378	736	2554	442	2449
actl	306	549	657	1123	429	631
intdisable	53	147	116	522	85	753
intenable	56	149	132	196	92	92
isr2entry	369	1478	855	2333	3406	7185
isrentry	240	686	366	528	3141	3561
isrexit	157	479	127	174	303	334
istentry	460	609	1020	1247	635	862
istexit	534	701	759	771	674	710
terml	521	597	862	1012	539	698

Table 2: RTOS performance (processor cycles).

3.1.2 Erika RTOS optimization

The next step consisted in optimizing the code of the RTOS to obtain better performance in all the tested processors. The first optimization consisted in modifying the ISR2 handling by avoiding to activate the ISR as a Task and directly calling the handler (i.e. not calling `osEE_activate_isr2()`). Moreover, similarly to [18], we have used the `-O3` optimization level of the GCC compiler.

Table 3 reports the best-case and the worst-case number of the cycles measured over 100 consecutive runs.

Test name	CLINT RISC-V		Cortex-R5		Cortex-A53	
	Min	Max	Min	Max	Min	Max
act	347	1204	247	1197	129	984
actl	267	551	172	280	88	90
intdisable	36	156	20	116	16	152
intenable	37	82	25	43	17	35
isr2entry	357	1484	256	1044	2988	4945
isrentry	229	692	170	286	2959	3060
isrexit	143	342	127	587	241	247
istentry	412	587	282	434	137	300
istexit	460	606	351	848	328	339
terml	405	484	295	372	172	196

Table 3: RTOS optimized performance (processor cycles).

If we restrict the analysis to the ARM Cortex-R5 (i.e. the direct competitor to RISC-V) and to the worst-case values, we obtain the values summarized in Table 4. From the presented values, we can see that the selected RISC-V processor still shows lower performance than the competing ARM Cortex-R5 architecture.

We identify the bottleneck of the design in CVA6's interrupt handling support. This, in fact, is not tuned for targeting fast-interrupt management and low interrupt latency, typically enabled by the following HW/SW features:

1. Hardware support for fine-grained and configurable interrupt priorities
2. Late-arriving interrupt behaviour (preemption and nesting) [21]
3. Banked stack pointer [21] (i.e. different stack pointers for different privilege levels)
4. Hardware support for automatic saving of registers during the context switch
5. Context save/restore optimization with back-to-back interrupts (tail chaining).

In the next section, we address the first two of the above mentioned design items. In particular, we extend the current CLINT interrupt controller with a Core Local Interrupt Controller (CLIC) [22], and provide an evaluation on the performance of the Erika RTOS .

The remaining three design items involve the implementation of more advanced hardware features in the processor itself, and will be investigated in a future work.

Test	Cortex-R5	CLINT RISC-V	Overhead
act	1197	1204	+1%
actl	280	551	+97%
intdisable	116	156	+34%
intenable	43	82	+91%
isr2entry	1044	1484	+42%
isrentry	286	692	+142%
isrexit	587	342	-42%
istentry	434	587	+35%
istexit	848	606	-29%
terml	372	484	+30%

Table 4: Worst-case overhead of CLINT RISC-V with respect to Cortex-R5.

3.1.3 Hardware design improvement

CVA6 lacks support for *vectored interrupts*, which store the interrupt service routine of each interrupt at a separate address. Albeit increasing the code size as the vector table grows larger, this mechanism helps reducing the overall interrupt response time. Furthermore, the PLIC does not support interrupt preemption (nesting), nor runtime-configurable interrupt priorities and interrupt threshold control, which have to be simulated entirely in software.

As a matter of fact, we first modified the CVA6 interrupt interface by replacing level sensitive interrupts with an handshake mechanism carrying the interrupt identifier and the request information to the processor, that acknowledges the handshake. We then added support for vectored interrupts by implementing interrupt identifier decoding logic to compute the jumping address of the vector table.

In a second step, we extended the CLINT with the Core Local Interrupt Controller (CLIC). We employed an open-source implementation of the CLIC[1] that reflects the latest status of the RISC-V CLIC draft specifications. The integration process included the addition of specific CSRs registers in the processor's micro-architecture as from specifications [22]. The CLIC introduces several improvements to the standard CLINT with the aim of achieving faster interrupt handling. Among those are dedicated memory-mapped registers for software configurable interrupt priority and levels at the granularity of each interrupt line, runtime-configurable interrupt

[1] https://github.com/pulp-platform/clic

mode and trigger type, support for interrupt preemption in the same privilege level (nesting of horizontal interrupts). Lastly, selective hardware vectoring enables the programmer to optimize each incoming interrupt for either faster response (vectored mode) or smaller code size (non-vectored mode, when each interrupt traps to the same exception handler address).

CVA6 interrupt handling was modified as in Fig. 3. In the improved design, the PLIC still arbitrates external system level interrupts and the legacy CLINT generates the timer interrupt. These interrupts are routed through the centralized CLIC interrupt source. In the same fashion, inter-processor interrupts are fired by writing to the corresponding CLIC memory-mapped registers. Lastly, using CLIC, local interrupts can be extended to up to 4096 lines instead of being limited to the processor's XLEN. In this work, we implemented 256 input interrupt lines arbitrated by the CLIC.

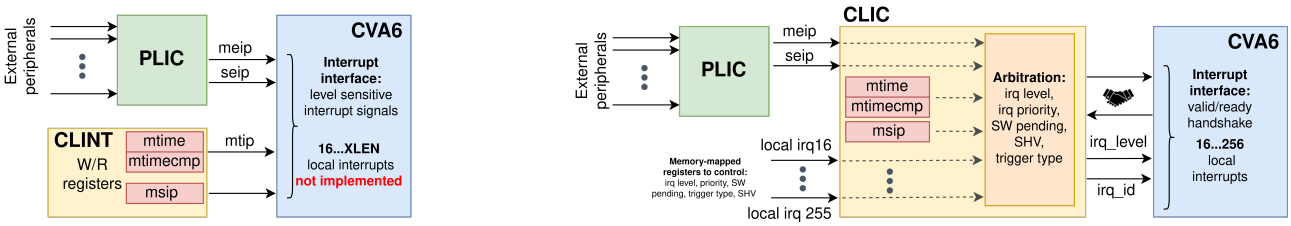

Figure 3: **(a)** Original CLINT + PLIC interrupt interface. **(b)** Improved CLIC + PLIC interrupt interface.

Table 5 contains the experimental results on this new hardware architecture using the `-O3` optimization level of the GCC compiler. As it can be seen, the worst-case overhead on RISC-V has become closer to the one measured on the competitor chip (i.e. ARM Cortex-R5). In particular, for 4 metrics (i.e. act, isrentry, istentry and istexit) the number of cycles needed by the RTOS are equal or even lower than the ones needed on Cortex-R5. These experimental results confirm that RISC-V is a proven technology for running AUTOSAR Classic stacks of next-generation automotive MCUs.

Test	Cortex-R5	CLIC RISC-V	Overhead
act	1197	1151	-4%
actl	280	464	+66%
intdisable	116	128	+10%
intenable	43	81	+88%
isr2entry	1044	1226	+17%
isrentry	286	539	+88%
isrexit	587	481	-18%
istentry	434	435	+0%
istexit	848	774	-9%
terml	372	408	+10%

Table 5: Worst-case overhead of CLIC RISC-V with respect to Cortex-R5.

3.2 RISC-V communication evaluation

This section provides experimental results on the communication latency between applications running on Linux and Erika RTOS respectively. As shown in Figure 4, the OSes run on different processors (i.e., Linux runs on ARM Cortex-A and Erika RTOS runs on the RISC-V CPU). The design in this case is of course different: the hardware architecture is AMP instead of SMP. Therefore, isolation and resource partitioning are physical Characteristics rather than being provided by a hypervisor.

The inter-OS communication between an application running on Linux and one running on Erika RTOS is based

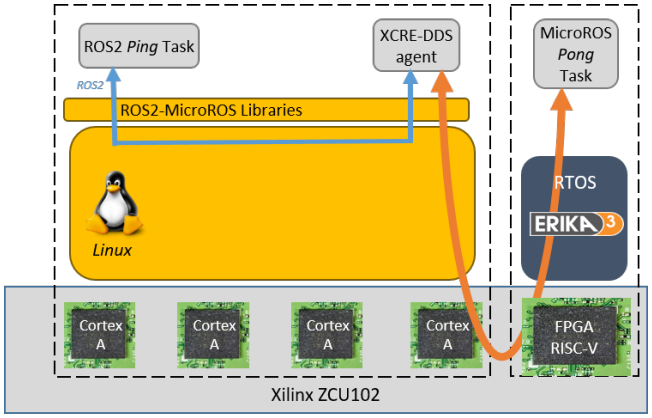

Figure 4: Inter-OS communication scenario.

on DDS-XRCE [8, 9], a DDS protocol specifically designed by OMG for resource-constrained systems. In particular, we have integrated eProsima's Micro XRCE-DDS stack on Erika RTOS . The clients communicate with an XRCE-DDS Agent running on Linux which provides the intermediate bridging service towards the DDS Data Global Space [11].

Data has been exchanged through a non-cached shared memory area, while the interrupt source has been based on the one for the UART device, since it was the only interrupt source visible by both operating systems. The involved processes on Linux (i.e. DDS Agent and ROS2 application) have been scheduled using a real-time priority (i.e. SCHED_RR with priority 99).

The communication latency has been evaluated through a ROS2 “ping” application running on Linux and a Micro-ROS "pong" application running on Erika RTOS on RISC-V. As shown in Figure 2, the communication latency ΔT corresponds to the span of time between the time a Ping message is transmitted and the time a Pong message is received in the “ping” application. It includes the delay due to the periodic engine of the Micro-ROS framework.

As shown in Table 6 the experimental results showed a minimum, average and maximum communication time of **2.0**, **2.2** and **3.7 msec**, respectively.

ΔT	Experimental results
Min	2.002 msec
Average	2.202 msec
Max	3.668 msec

Table 6: Ping-Pong time communication.

It is important to highlight that the Micro-ROS framework has a periodic engine which added some delay to the communication. In particular, the `rclc_executor_spin_some` function had a period of 1 msec, while all the other interactions were event-driven. In the future, the design of the selected RISC-V processor could be improved by adding user-defined interrupt sources to be used in place of the UART.

4 Linux evaluation

This section contains an empirical evaluation of the Linux OS.

Section 4.1 contains the analysis of scheduling latency in Linux-based PikeOS guest. More in detail, the scheduling latency on ElinOS guest has been measured with and without the optimizations provided by the PREEMPT_RT real-time patch [10].

Section 4.2 contains the evaluation of the overhead introduced by the virtualization on real-time Linux applications. More in detail, it shows the impact of the PikeOS hypervisor on the execution time of real-time applications, when using FPGA acceleration through the FRED framework for Linux. A similar evaluation on the impact of PikeOS on more complex applications organized in Directed Acyclic Graphs (DAGs), is presented in Deliverable D4.4 [23] instead.

Note that the Linux kernel in the AMPERE run-time stack has been modified to include the APEDF variant of the SCHED_DEADLINE real-time CPU scheduler, the APEDF-aware modifications to the schedutil component handling DVFS on Linux, and the runmeter energy monitor within the kernel, and its configuration has been customized in ELinOS to support the deployment under PikeOS. These are all described and evaluated in Deliverable D4.4 [23], given the tight relationship of these features with the predictability, energy estimation and efficiency capabilities of the Linux kernel in the AMPERE run-time subsystem, all aspects dealt with in the context of WP4, where they are discussed also in relationship to the multi-criteria optimization framework realized in WP3 (and described in Deliverable D3.4 [24]).

4.1 Scheduling latency evaluation in ElinOS guest

This section evaluates the real-time performance by measuring the scheduling latency of the OS, with and without the optimizations provided by the PREEMPT_RT patch. In every OS, in fact, there is always a delay between an activation event and the instant when the unblocked task starts execution, even for the highest-priority tasks. Typical sources of such delay include time for operating context switch and change of privilege level, timer granularity, preemptions from tasks at higher priority, contention on shared resources, critical sections, etc. Such delay, known as "*scheduling latency*", affects the response time of the task and imposes a lower bound on the deadlines that can be supported by the system. Therefore, it is essential to take into account scheduling latencies at system design to understand whether the system can meet the needed timing requirements.

We have compared the scheduling latency measured using two typical preemption models of Linux: the preemptible kernel (`PREEMPT_LL`, available on all Linux kernels) and the fully-preemptible kernel (`PREEMPT_RT`, provided by the PREEMPT_RT patch). In a Linux-based preemptible kernel (`PREEMPT_LL`) all kernel code, that is not executing in a critical section, is preemptible. This allows reaction to interactive events by permitting a low-priority process to be preempted involuntarily even if it is in kernel mode executing a system call and would otherwise not be about to reach a natural preemption point. Thus, applications run more 'smoothly' even when the system is under load, at the cost of slightly lower throughput and a slight runtime overhead to kernel code. On the other hand, a Linux-based fully-preemptible kernel (`PREEMPT_RT`) specifically aims to improve scheduling latency by reducing the number and the length of critical sections in the kernel that mask interrupts or disable preemptions [10].

Scheduling latency under Linux is typically measured using `cyclictest`, a tracing tool that treats the kernel as a black box and reports the scheduling latency experienced by user-level tasks. In our experiments, we stressed the system by creating interference through both the `find` command (generating I/O traffic by scanning the filesystem on the SD memory and printing on console) and the `stress` tool, generating CPU, memory and I/O interference:

```
$ ./rt-test/stress -c 8 -i 8 -m 8 -vm-bytes 8000000
```

The worst-case latency has been measured through `cyclictest` using the following options:

```
$ ./cyclictest -mlockall -smp -priority=80 -interval=200 -distance=0
-duration=5m
```

As shown in Table 7, the worst-case latency is 13.4 ms without PREEMPT_RT and **159 μs** with PREEMPT_RT. It means that the latency has been reduced of about **99%** by simply applying the patch and recompiling the Linux kernel. The measured value is suitable for the execution of the real-time control tasks targeted by the AMPERE project.

Preemption model : `PREEMPT_LL`											
T: 0 (251)	P:80	I:200	C:1499959	Min:	8	Act:	13	Avg:	12	Max:	82
T: 1 (252)	P:80	I:200	C:1499547	Min:	6	Act:	11	Avg:	11	Max:	13404
T: 2 (253)	P:80	I:200	C:1499266	Min:	6	Act:	11	Avg:	11	Max:	44
T: 3 (254)	P:80	I:200	C:1499027	Min:	8	Act:	12	Avg:	11	Max:	67
Preemption model : `PREEMPT_RT`											
T: 0 (261)	P:80	I:200	C:1499980	Min:	8	Act:	13	Avg:	12	Max:	55
T: 1 (262)	P:80	I:200	C:1499389	Min:	9	Act:	14	Avg:	12	Max:	159
T: 2 (263)	P:80	I:200	C:1498969	Min:	9	Act:	12	Avg:	11	Max:	44
T: 3 (264)	P:80	I:200	C:1498628	Min:	9	Act:	14	Avg:	12	Max:	53

Table 7: Measured scheduling latency using different preemption models.

4.2 Impact of virtualization on real-time tasks

To evaluate the impact of running real-time Linux applications organized in DAGs under PikeOS, we set up a series of benchmarks on the Xilinx UltraScale+ ZCU102 target platform.

In particular, the purpose of these benchmarks is to evaluate the overhead introduced by the hypervisor on guest applications. In general, we distinguish between two kinds of overhead:

- One affecting all applications executing on the CPU; in this case, the overhead is due to the interference of the hypervisor when performing system calls or other routine activities, even without interacting with external devices.
- The second affects only FPGA-accelerated tasks; for these tasks, the overhead is due to the mediation of the hypervisor in the interactions between the operating system and the external FPGA device.

Once properly measured experimentally on the target platform, these two overhead contributions can be taken into account when performing the offline optimization steps described in Deliverable D3.4 [24], by inflating each expected task execution time accordingly. This is coherent with the way we consider other kinds of overheads or high-priority interference from other tasks executing on the same machine as the target applications (e.g., the FRED server application), as detailed in [25].

The benchmark applications that we prepared focus on evaluating these two kinds of overhead. First, each overhead is evaluated in isolation from one another. Then, we measure the overall impact of the hypervisor on more complex application scenarios that emulate the behavior of the industrial use-case applications. This last part is described in Deliverable D4.4 [23].

4.2.1 Experimental test bench

The test bench used to evaluate the impact of running software and hardware real-time tasks under PikeOS consists of a set of micro-benchmark applications that perform a pre-fixed amount of work, either in software, or as hardware-accelerated runnables through the FRED framework.

For running the experiments, we have used the AMPERE reference platform consisting of a Xilinx ZCU102 evaluation board[1]. All experiments execute with a Linux kernel version 5.10, compiled with PREEMPT_RT support and with the CPU frequency fixed to the maximum available one on the target platform (1.2 GHz). Experiments without the hypervisor use a PetaLinux distribution[2], while those with PikeOS[3] (version 5.1) use ElinOS[4] (version 7.1). Both kernel versions are compiled to provide support for the FRED framework to access the FPGA device and execute real-time hardware-accelerated calls.

4.2.2 Overhead on tasks executing on the CPU

For this evaluation, our benchmark application performs a pre-determined amount of work consisting of a mix of both CPU and memory-bound operations. We then compare the time required to execute the same workload with and without the hypervisor (leaving any other configuration of the platform unchanged, e.g., DVFS settings, scheduling priorities, etc.) to obtain a rough estimate of this overhead.

To eliminate most other forms of interference other than the hypervisor, most services executing on the operating system have been disabled at boot, or shut down manually before the runs.

From our experimentation on the target Xilinx Ultrascale+ ZCU102 platform, we estimated approximately a 1.18% overhead when executing an application on the CPU compared to the same scenarios without the hypervisor in place.

4.2.3 Overhead on tasks executing on the FPGA

The overhead experienced by hardware-accelerated tasks can be split into two different values:

- Reconfiguration overhead, related to only the time required to reconfigure a slot of the external FPGA device when replacing a hardware accelerator with another one (while time-scheduling accelerators on a slot of the FPGA fabric, as allowed by FRED);
- execution overhead, related to only the duration of an invocation to a hardware function that has already been configured on one slot of the FPGA fabric, including the time needed to actually perform the computations of the hardware function, together with the time needed for the data exchanges between the CPU and the FPGA device.

To evaluate these two kinds of overhead, we used some simple representative FRED-compatible hardware tasks. These tasks perform some simple mathematical operations between vectors and matrices, and they have been obtained by using Xilinx High-Level Synthesis tool, in combination with DART (the latter is needed to compile hardware designs so that they can be deployed on FPGA slots using the FRED framework).

Table 8: Comparison between reconfiguration time of FPGA tasks without and with PikeOS, together with the estimated overhead in percentage.

Hardware Task	Without PikeOS [ms]	With PikeOS [ms]	Overhead [%]
`mul128`	22.856	23.821	4.223
`mul64`	22.789	23.251	2.029
`sum1024`	22.808	23.872	4.664
`xor1024`	22.809	23.596	3.450

[1] See: https://www.xilinx.com/products/boards-and-kits/ek-u1-zcu102-g.html.
[2] See: https://www.xilinx.com/products/design-tools/embedded-software/petalinux-sdk.html.
[3] See: https://www.sysgo.com/pikeos.
[4] See: https://www.sysgo.com/elinos.

Table 8 shows the difference between the time required to reconfigure the external FPGA device without the hypervisor and under PikeOS. From our experimentation, we observed that this overhead is overall proportional to the amount of time spent in reconfiguration without the hypervisor involved. Overall, we measured an overhead up to 5% compared to the scenarios without PikeOS involved. Notice however that in the overall AMPERE workflow this overhead will be paid if two or more tasks are selected for execution on the FPGA. If during the offline optimization phase only one task is selected to run on the FPGA, this cost is never experienced by the target application.

On the other hand, the hypervisor introduces overhead also on the execution of a hardware task. Table 9 compares the execution time of the various hardware tasks that we selected with and without the hypervisor. In this case, the overhead that we measured does not seem to be related to the original execution time of the hardware task. Rather, the hypervisor introduces a fixed amount of overhead, which of course impacts more significantly shorter hardware tasks. This is aligned with the fact that, if no FPGA slot reconfiguration is needed, the hypervisor interference happens only during the interactions between the application main thread, and the FRED server, as their socket-based interactions through a socket on Linux involves system calls that are mediated by the hypervisor. From our experimentation, the overhead introduced is no more than 0.057ms per hardware task execution.

Table 9: Comparison between execution time of FPGA tasks without and with PikeOS, together with the estimated overhead in percentage.

Hardware Task	Without PikeOS [ms]	With PikeOS [ms]	Overhead [%]
`mul128`	4.332	4.382	1.150
`mul64`	1.224	1.277	4.265
`sum1024`	0.843	0.898	6.422
`xor1024`	0.843	0.899	6.738

5 Acronyms and Abbreviations

CLINT	Core Local Interruptor
CPU	Central Processing Unit
DAG	Direct Acyclic Graph
DDS	Data Distribution Service
FPGA	Field-Programmable Gate Array
GPU	Graphics Processing Unit
IP	Intellectual Property
MPSoC	Multi-Processor System on a Chip
OS	Operating System
PLIC	Platform Local Interrupt Controller
ROS	Robot Operating System
RTOS	Real-time operating system
SoC	System On Chip
UART	Universal Asynchronous Receiver Transmitter
WP	Work Package
XRCE	eXtremely Resource Constrained Environments

6 References

[1] Xilinx, “Zcu102,” https://www.xilinx.com/products/boards-and-kits/ek-u1-zcu102-g.html.

[2] ——, “Ultrascale+ mpsoc,” https://www.xilinx.com/products/silicon-devices/soc/zynq-ultrascale-mpsoc.html#productTable.

[3] O. group, “CVA6 RISC-V CPU,” https://github.com/openhwgroup/cva6.

[4] SYSGO, “Pikeos,” https://www.sysgo.com/pikeos.

[5] ——, “Elinos,” https://www.sysgo.com/elinos.

[6] Evidence Srl, “Erika enterprise rtos,” http://www.erika-enterprise.com.

[7] AUTOSAR, “Classic platform,” https://www.autosar.org/standards/classic-platform/.

[8] “ROS2 - robotic operating system 2,” https://index.ros.org/doc/ros2/.

[9] “microros,” https://micro.ros.org/.

[10] The Linux Foundation, “The real-time linux collaborative project,” https://wiki.linuxfoundation.org/realtime/.

[11] eProsima, “microros agent,” https://github.com/eProsima/Micro-XRCE-DDS-Agent#readme.

[12] G. Fumaroli, “Security Target PikeOS Separation Kernel v5.1.3,” 2022. [Online]. Available: https://www.bsi.bund.de/SharedDocs/Downloads/DE/BSI/Zertifizierung/Reporte/Reporte1100/1146b_pdf.pdf?__blob=publicationFile&v=3

[13] J. Rushby, “Design and verification of secure systems,” in *Eighth ACM symposium on operating system principles*, 1981, pp. 12–21, tex.lookup: Rushby1981design-and-verification-of-secure-systems. [Online]. Available: http://www.sdl.sri.com/papers/sosp81/sosp81.pdf

[14] Bundesamt für Sicherheit in der Informationstechnik (BSI), “Certification Report BSI-DSZ-CC-1146-2022,” Tech. Rep., 2022. [Online]. Available: https://www.bsi.bund.de/SharedDocs/Downloads/DE/BSI/Zertifizierung/Reporte/Reporte1100/1146a_pdf.pdf?__blob=publicationFile&v=1

[15] A. Waterman, Y. Lee, R. Avizienis, D. A. Patterson, and K. Asanović, “The risc-v instruction set manual volume ii: Privileged architecture version 1.9,” EECS Department, University of California, Berkeley, Tech. Rep. UCB/EECS-2016-129, Jul 2016. [Online]. Available: http://www2.eecs.berkeley.edu/Pubs/TechRpts/2016/EECS-2016-129.html

[16] RISC-V Community, “RISC-V Platform-Level Interrupt Controller Specification,” https://github.com/riscv/riscv-plic-spec/blob/master/riscv-plic.adoc, 2022.

[17] C. Scordino, I. M. Savino, L. Cuomo, L. Miccio, A. Tagliavini, M. Bertogna, and M. Solieri, “Real-time virtualization for industrial automation,” in *2020 25th IEEE International Conference on Emerging Technologies and Factory Automation (ETFA)*, vol. 1. IEEE, 2020, pp. 353–360.

[18] R. Balas and L. Benini, “Risc-v for real-time mcus-software optimization and microarchitectural gap analysis,” in *2021 Design, Automation & Test in Europe Conference & Exhibition (DATE)*. IEEE, 2021, pp. 874–877.

[19] R.-V. N.-I. Specifications, “Proposal for new Embedded ABI (EABI),” https://github.com/riscv/riscv-eabi-spec, Tech. Rep., 2019.

[20] P. Burgio, M. Bertogna, N. Capodieci, R. Cavicchioli, M. Sojka, P. Houdek, A. Marongiu, P. Gai, C. Scordino, and B. Morelli, “A software stack for next-generation automotive systems on many-core heterogeneous platforms,” *Microprocessors and Microsystems*, vol. 52, pp. 299–311, 2017.

[21] ARM, “Cortex®-M3 Technical reference manual,” https://developer.arm.com/documentation/ddi0337/latest/, Tech. Rep., 2008.

[22] RISC-V Community, “"Smclic" Core-Local Interrupt Controller (CLIC) RISC-V Privileged Architecture Extension,” https://github.com/riscv/riscv-fast-interrupt/blob/master/clic.adoc, 2022.

[23] AMPERE, “Deliverable D4.4 – Evaluation of run-times,” June 2023.

[24] ——, “Deliverable D3.4 – Evaluation of multi-criteria optimizations,” June 2023.

[25] T. Cucinotta, A. Amory, G. Ara, F. Paladino, and M. D. Natale, “Multi-criteria optimization of real-time DAGs on heterogeneous platforms under p-EDF,” *ACM Transactions on Embedded Computing Systems*, Apr. 2023. [Online]. Available: https://doi.org/10.1145%2F3592609