A Model-driven development framework for highly Parallel and EneRgy-Efficient computation supporting multi-criteria optimisation

D4.4 Evaluation of run-times

Version 1.0

Documentation Information

Contract Number	871669
Project Webpage	https://www.ampere-euproject.eu/
Contractual Deadline	30.06.2023
Dissemination Level	Public (PU)
Nature	Report
Authors	Tommaso Cucinotta, Gabriele Ara (SSSA)
Contributors	Francesco Paladino and Alexandre Amory (SSSA) Sara Royuela and Eduardo Quiñones (BSC) Sergio Mazzola (ETHZ) Tiago Carvalho, Mohammad Samadi and Luis Miguel Pinho (ISEP) Enkhtuvshin Janchivnyambuu (SYS)
Reviewer	Darshak Sheladiya (SYS)
Keywords	run-time energy management, FPGA, dynamic partial reconfiguration, performance counters

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	Initial version with skeleton and section assignments.
V0.2	First round of contributions from partners.
V0.3	Refined contributions from partners.
V0.4	All contributions included. Ready for review.
V1.0	Final version with reviewers' comments included.

Table of Contents

1 Executive Summary

This document constitutes Deliverable *D4.4. Evaluation of run-times*, built upon the work carried out in WP4, with contributions from several other WPs. The document is provided in the format of a report, and it was due on project month 36 (December 2022), extended to month 42 (June 2023) after the 6-months extension of the project duration agreed with the EC.

D4.4 presents the last updates on the components included in the run-time architecture of AMPERE in the last months of the project, summarizing the activities carried out by AMPERE partners in Task 4.5 “Run-time mechanisms for safety/security” and Task 4.6 “Run-time validation”. It shows results from a set of experiments providing an experimental evaluation of the mechanisms realized in WP4, including energy management, scheduling algorithms for predictable computation and communication, segregation mechanisms for safety/security support, and mechanisms for resilience support.

2 Updated AMPERE run-time architecture

2.1 AMPERE run-time architecture overview

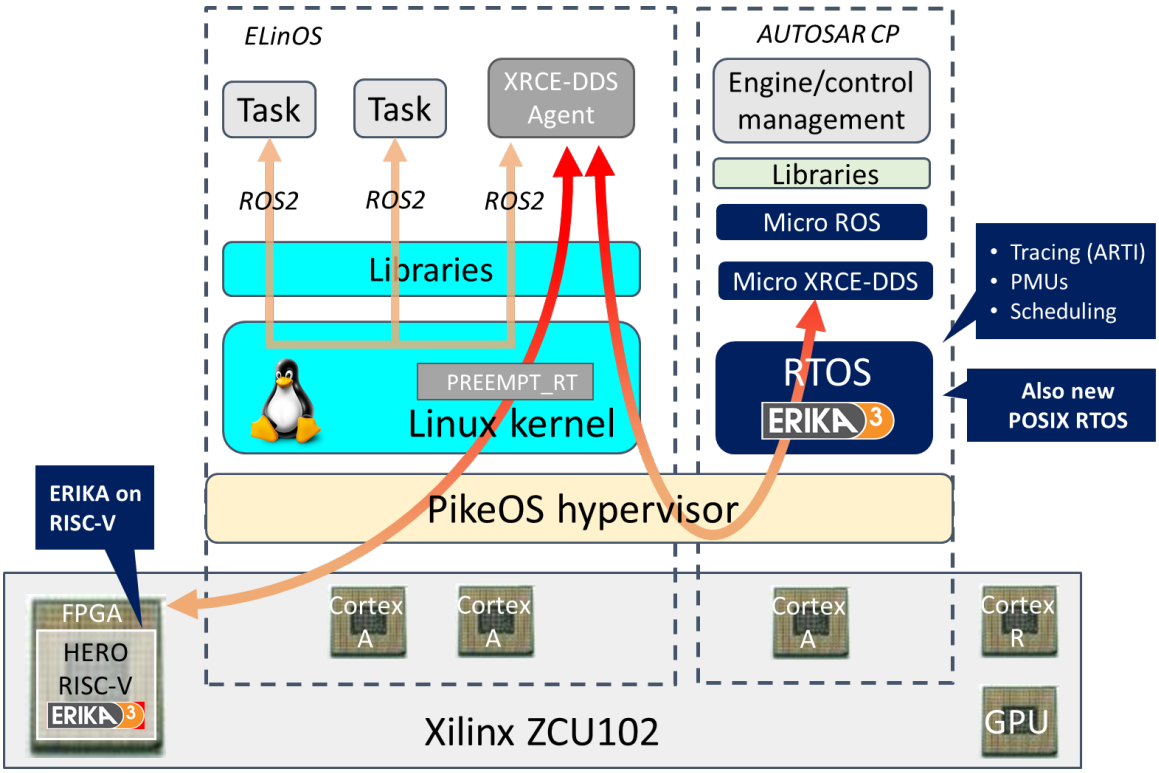


Figure 1: The AMPERE run-time architecture.

The AMPERE general run-time architecture is depicted in Figure 1, where the major run-time components and mechanisms realized throughout the whole software stack are highlighted. The final runtime architecture needs some customization for the two reference boards considered in the project, the Xilinx UltraScale+ and the NVIDIA Jetson AGX, as not all components can be deployed on both boards. Figure 1 refers purposely to the Xilinx platform with FPGA acceleration, where the architecture is enriched with the PikeOS real-time hypervisor supporting separation among the Linux domain and the hard real-time Erika RTOS domain. However, the reference architecture for the Jetson Xavier AGX board with GPU acceleration, is obtained simply as the Linux partition in the same figure, deployed on bare-metal, with the additional detail that the Linux distribution is a Ubuntu-based distribution enriched by NVIDIA with a set of development tools for CUDA. The CUDA run-time components can be leveraged by the `libomptarget` components of the OpenMP run-time, for GPU offloading.

More precisely, Table 1 summarizes the software components and their version(s) that have been included in the final AMPERE run-times for the two boards. A number of components are actually modifications made by AMPERE partners to standard components:

- the Linux kernel 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;
- the OpenMP run-time is a modified version of the version coming with `clang` and `llvm`, version 17.0.0, supporting replication of runnables, variants constructs for hardware offloading, static mapping, integration with the Extrae monitoring framework, and the ability to deploy worker threads under

Table 1: Software components and their versions in the AMPERE runtimes

Component	Version(s) on the US+	Version(s) on the AGX	Notes
PikeOS hypervisor	5.0	–	
ELinOS	7.0	–	Linux distribution for Xilinx UltraScale+ under PikeOS
Erika RTOS	Internal	–	See D5.4 [1]
Peta Linux OS	2020.2	–	Linux distribution for Xilinx UltraScale+ bare-metal
Ubuntu Linux OS	–	20.04.6 LTS	Includes Jetson Linux 35.2.1
Linux kernel	5.10.104-tegra	5.10.107-ELinOS-5439-rt64	Modified by SSSA
Runmeter	1.0	1.0	
OpenMP runtime	LLVM 17.0.0	LLVM 17.0.0	Modified by BSC
Micro-ROS for Erika	Internal to Erika	–	See D5.4 [1]
Micro-XRCE DDS	eProsima XRCE-DDS	–	See D5.4 [1]
ROS2	Galactic	Galactic	
XRCE DDS Agent	–	–	
FRED	Internal, open-source	–	See D4.3 [2]
CUDA	–	11.4	

SCHED_DEADLINE.

Further details about the various components are provided in the sections that follow.

2.2 FRED porting and integration with PikeOS/ElinOS

The execution of FPGA-accelerated tasks on the Xilinx UltraScale+ target platform of the AMPERE project is performed through the FRED framework [3], which allows the predictable execution of hardware-accelerated tasks on FPGA-based system-on-chips platforms. The project is developed at the Real-Time Systems Laboratory (RETIS Lab) of SSSA AMPERE partner.

The FRED architecture has been described in great detail in D4.3 [2]. Here, we list the changes that have been committed to the software. Originally, the FRED in-kernel components have been developed as a collection of Yocto layers, which made it compatible with Petalinux v2020.2-compatible kernel versions (see Section 3.2.3 of D4.3 [2]). This was useful to engineer a bare-metal set-up on the Xilinx UltraScale+ platform. Later, to run FRED-based applications on top of the hypervisor PikeOS (necessary for use-cases targeting the Xilinx UltraScale+ platform), these components had to be ported to be compatible with the latest version of ElinOS (7.0), its companion Linux distribution. To do so, the Scuola and SysGo collaborated to incorporate all the necessary components into a single project, containing:

- a PikeOS specification targeting the Xilinx UltraScale+ platform;
- an ElinOS implementation integrated with FRED support;
- FRED user-space components, like its daemon and client library.

The project is hosted at `https://github.com/fred-framework/fred-elinos`, where step-by-step instructions can be found to compile and flash the framework onto a memory card, ready to be executed on the target platform. The compiled Linux system includes also PREEMPT_RT support and other necessary patches like the one described in Section 2.3.1, necessary to properly execute FRED applications in presence of real-time threads under the SCHED_DEADLINE scheduler.

2.3 Linux kernel changes for energy-aware real-time scheduling

In this section we describe the changes applied to the Linux kernel to properly support energy-aware real-time scheduling on Linux developed as part of the AMPERE project. While some of these changes are fundamental to ensure the predictability of applications like those targeted by the AMPERE project, others are more general and target different classes of systems that are related by not exactly part of AMPERE.

2.3.1 Swapping the RT and SCHED_DEADLINE scheduling classes

Many accelerator frameworks, including FRED [3], rely on a client-server architecture in which a single process[1] (either running in userspace or as a kernel worker thread) is in charge of managing acceleration requests coming from multiple client processes executing on the same host. In this context, we must take into account that real-time applications sending acceleration requests to the server process will block waiting for the execution of their request. The server process however might not be able to execute if its priority is not a real-time priority, always preempted by other real-time tasks executing on the same system. To avoid cases like this, in which a high-priority process (the client) is blocked and is subject to some form of priority inversion (other processes execute preventing it to wake up potentially indefinitely), we must execute the server at a sufficiently high priority. In particular, to expedite the execution of acceleration requests as much as possible, the server process should be assigned the highest priority available in the system. This way, there can be potentially no gaps between the instantiation of a new acceleration request and the effective start of the execution of the accelerated task on the FPGA/GPU. In other words, the server process should behave more like an interrupt, which is executed as soon as possible, rather than as a regular real-time process.

In regular Linux implementations, several scheduling classes are available, two of which are real-time: RT (corresponding to the POSIX SCHED_FIFO and SCHED_RR scheduling policies) and SCHED_DEADLINE. The latter has the highest priority of the two, meaning that a task that is assigned to the SCHED_DEADLINE scheduling class will always preempt a task belonging to the RT class. In EDF-based scheduling, however, priorities are dynamic (depending on the absolute deadlines of each task), and it is not so straightforward to assign the server process a set of SCHED_DEADLINE parameters (runtime budget, relative deadline, and period) so that it will always be the highest priority process in the system.

To overcome this limitation, a patch has been developed for the Linux scheduler that swaps the priorities of the two scheduling classes, SCHED_DEADLINE and RT, so that RT is the one with the highest priority. While normally this behavior is undesirable (tasks belonging to the RT class cannot be easily throttled like those executing under SCHED_DEADLINE, for example, which have a specific budget), it is also the most straightforward solution to the aforementioned problem. When running on a system compiled with this patch, the accelerator server process can now be assigned the highest RT priority (99), and it will always be the highest priority process in the system[2], always preempting real-time tasks belonging to the SCHED_DEADLINE class on the same CPU.

With this change in place, offline optimization tools, like the one described in D3.4 [4], can treat the execution of the server process as a high-level interference, just like other forms of interference (e.g., interrupt handlers).

The above described patch for the Linux kernel turns out to be useful also to deal with kernel threads that are involved in the interaction of Linux applications with the GPU device, for example on the NVIDIA Jetson Xavier AGX board. Indeed, with reference to this platform, we can observe that, by tracing an application performing a number of calls to a GPU-offloaded function, for example using a `pragma omp target device` OpenMP directive, we can see that a number of kernel threads are activated several times during the interaction with the accelerator. This will be shown in the experiment in Section 3.1.3.

[1] Without loss of generality, we consider only one single process as the one in charge of executing acceleration requests. In reality, multiple processes might be involved in a single acceleration request. As long as the same precautions are taken for all the processes involved, the reasoning described here applies also for the multi-process case, even if in this section we always refer to a single "server" process.

[2] As long as no other RT process is assigned the same priority.

2.3.2 Fixing energy-aware real-time scheduling on Linux

Dynamic energy-aware real-time scheduling is out of the main scope of the AMPERE project, in which the purpose of the multi-criteria optimization phase [4] is that of ensuring that real-time tasks executing on a properly statically configured system. It is nonetheless interesting to explore dynamic mechanisms that attempt to dynamically change the system configuration (e.g., DVFS settings) and/or task placement and scheduling conditions. While this is unsuitable for hard real-time scheduling of critical systems, like the industrial use cases proposed for the AMPERE project, it might lead to better solutions for soft real-time systems, which have more relaxed constraints.

The Linux kernel provides dynamic energy-aware real-time scheduling by implementing the GRUB-PA algorithm [5], which regulates the interaction between the SCHED_DEADLINE scheduling class and the `schedutil` CPUFreq governor (when selected).

`schedutil` attempts to impose some restrictions on the frequency selection depending on the information provided by SCHED_DEADLINE. In particular, to avoid breaking the guarantees provided by SCHED_DEADLINE to its tasks, `schedutil` tries to select the next frequency for a CPU such that the CPU capacity does not drop below the "running bandwidth" [5] advertised by SCHED_DEADLINE for each CPU. In other words, `schedutil` selects the minimum CPU frequency capable of scheduling the set of SCHED_DEADLINE tasks on each CPU.

While these mechanisms seem relatively safe, they can be broken almost trivially by an unsuspecting user. Global EDF (G-EDF), as implemented in SCHED_DEADLINE, cannot provide any bounded tardiness guarantee to userspace on multi-core systems where CPU frequencies are free to change over time [6, 7] (either due to DVFS or to some other mechanism like thermal throttling): tasks scheduled under G-EDF can potentially migrate at any activation, which leads to the running bandwidth of each CPU fluctuating a lot over time; GRUB-PA will attempt to select safe frequencies to run at, but since this value is tied to the running bandwidth of the CPU, it will be subject to fluctuations as well. Generally, a global utilization admission test (such as the one currently implemented by SCHED_DEADLINE) does not work when each CPU capacity can change over time (due to the changing frequency) and tasks are scheduled using G-EDF.

Finally, the maximum capacity of a CPU in Linux is defined as the capacity of the CPU when running at the maximum frequency. Unfortunately, on many platforms, the frequencies advertised as the maximum typically lead to thermal issues. The issue is prominently present on embedded and mobile devices, which often cannot afford active cooling. The unsustainability of these frequencies for relatively long periods is a significant issue for the admission of tasks to SCHED_DEADLINE, because tasks may be admitted even though it is virtually impossible to schedule them or even provide other guarantees, like "bounded tardiness" [8], due to thermal throttling. This behavior can be easily reproduced by attempting to execute any task with utilization close to the maximum capacity of a CPU, if the execution time of the task is carefully calibrated.

To address these issues, SSSA has developed a set of patches to the Linux kernel concerning the SCHED_DEADLINE scheduling class and the `schedutil` frequency governor.

2.3.2.1 Thermal-safe scheduling of real-time tasks

This patch set addresses the issue of scheduling real-time tasks under potential thermal throttling conditions. It assumes that the user of the system has some knowledge (either by design or by empirical evidence) about which DVFS settings available on the target system are "thermal-safe" and which are not.

The changes applied by this patch involve both SCHED_DEADLINE and the `schedutil` frequency governor. In the modified Linux implementation, the capacity of each CPU core (accounted by both SCHED_DEADLINE and `schedutil`) is tied with the maximum scaling frequency selected by the user for each CPU. If the user selects a "thermal-safe" frequency as the maximum one, then real-time tasks cannot be affected by thermal throttling-related issues anymore. Implementing this mechanism requires to change the frequency selection mechanism in `schedutil` (so that the correct capacity is accounted for each CPU) and in SCHED_DEADLINE (to adjust the consumed runtime of each task accordingly).

2.3.2.2 Adaptively Parititioned EDF scheduler for SCHED_DEADLINE

This patch set changes the scheduling policy implemented by SCHED_DEADLINE from G-EDF to another policy, called Adaptively Partitioned EDF (APEDF) [9, 10], with no change on the user-space API. This strategy has been devised as a middle ground between G-EDF and Partitioned EDF (P-EDF), in an attempt of retaining the positive aspects of both G- and P-EDF without their respective negative aspects.

The core idea behind APEDF is that if a taskset is partitionable, the scheduler will automatically partition its tasks to the available cores, realizing automatically what is effectively a P-EDF policy, for which not only a tardiness bound, but also precise deadline guarantees, can be provided. For non-partitionable task sets, a fall-back mechanism to G-EDF is provided. Leveraging this strategy, the number of task migrations is reduced drastically (if possible) compared to G-EDF, significantly improving the DVFS effectiveness of the scheduler when used in combination with the `schedutil` frequency governor (see Section 3.2).

To implement APEDF, SCHED_DEADLINE must be modified to push away tasks only if they do not fit on the core in which they wake up and to disable all pull mechanisms. When pushing a task away, it will be moved to a different CPU where it fits; otherwise, we will fall back to the regular G-EDF push mechanism if no such CPU can be found. APEDF can support different partitioning strategies, similar to P-EDF. Examples include First-Fit (FF) or Worst-Fit (WF). If FF is used, there is a sufficient global utilization bound that can establish whether a taskset is partitionable and, conversely, schedulable. This bound can be used for hard real-time tasks during admission control to provide the guarantee that no deadline will be missed.

2.4 Run-time mechanisms for resilience

AMPERE defines two software mechanisms to achieve resilience: (1) task-level parallel replication through OpenMP (based on the definition of SIL/ASIL levels in the models), and (2) dynamic monitoring through fine-grained proactive orchestration. These mechanisms are summarized in deliverable D4.3 [2] and further detailed in D3.3 [11]. Furthermore, all resilience features are included in demonstrator D2.3 [12].

A preliminary evaluation of the two resiliency mechanisms, in terms of performance, accuracy and programmability, was presented in D3.3. In this case, the techniques are evaluated on top of an implementation provided by the University of Siena (UNISI) of the tracking sub-system of the ODAS use case. This evaluation concluded that:

1. Regarding *performance*, the observer mechanism introduces minimal overhead, hence it is the suitable solution when performance is the main goal. Yet, replication shows good scalability when the system offers available resources, reducing the effects of the overhead.
2. Regarding *accuracy*, the combination of the two mechanisms offers the best results, reaching more than 90% in many scenarios. In isolation, dynamic monitoring is better for detecting silent faults, while replication shows better results for detecting erroneous results.
3. Regarding *programmability*, OpenMP is clearly the simplest solution, requiring very little effort and knowledge from the programmer. Dynamic monitoring shows acceptable levels of programmability by decoupling the implementation of the application from the observation.

The final evaluation of the resilience approach in AMPERE is presented in deliverable D3.4 [4]. In this case, only the replication mechanism is tested because the code generated from the models is synthetic, i.e., functions contain only artificial code that resembles the load produced by the real implementation. Consequently, there are no real variables to track, and this has two main consequences while evaluating resilience: (1) the dynamic monitoring mechanism cannot be applied, and (2) accuracy cannot be evaluated. Hence, D3.4 [4] presents an evaluation of resilience on top of the PCC and ODAS use cases in terms of performance and scalability.

2.5 Run-time energy monitoring

This chapter describes the updates to the run-time energy monitoring support implemented as part of the AMPERE ecosystem, along with its evaluation carried out for milestone MS4. The deliverable D3.3 [11] describes the power and energy models adopted for the multi-criteria offline optimization strategies. Deliverable D3.4 [4] complements it with the related model updates for milestone MS4.

In the following, we report the related updates to the Runmeter online energy monitoring framework proposed in deliverable D4.3 [2]. Starting from the platform's power models [11, 4], such a framework provides accurate and responsive energy estimates at different level of granularity during the application runtime. The monitoring framework has a negligible impact on system's performance, and the estimates can be used for monitoring purposes, as well as actuation drivers (e.g., for DVFS, or energy-aware task scheduling). We also summarize the evaluation of the monitoring framework, which was already carried out for the NVIDIA Jetson AGX Xavier target platform [13, 14] as part of deliverable D4.3 [2].

2.5.1 Updates to the energy model

At the core of Runmeter there are the power models devised as part of deliverable D3.3 [11]. During the multi-criteria integration phase, minor incompatibilities appeared between the performance monitoring counters (PMCs) required for such models and some profiling tools in the AMPERE ecosystem. In particular, this issue affected the power model for the NVIDIA Jetson AGX Xavier's CPU. To solve it, we updated the CPU power model as reported in D3.4 [4] and repeated the model training step.

For consistency between the offline multi-criteria optimization, performed as part of WP3, and the power models driving WP4's energy estimates, we also updates the power models in Runmeter. As in D3.4 [4], such an update does not affect in any visible way the accuracy of CPU energy consumption estimation.

2.5.2 Evaluation of the energy monitoring framework

The evaluation of the energy monitoring framework is part of MS4. However, as the framework was already available, we already performed such analysis as part of deliverable D4.3 (Section 4.2) [2]. In the following, we summarize the main results for energy monitoring accuracy and framework overhead for the Xavier board.

2.5.2.1 Energy estimation accuracy

Our energy monitoring framework, Runmeter, runs as part of the Xavier's Linux kernel. Floating-point operation are not allowed in the kernel, and would anyway impact the system's performance in a non-negligible way. For this reason, we converted our power models [11] to a 64-bit fixed-point data type with 29 bits for the fractional part.

As reported in D4.3 [2], our evaluation shows that the conversion to a fixed-point model causes a maximum absolute error of about 17 mW for the approximation, corresponding to a relative error of belo 0.8%.

As far as power estimation accuracy is concerned, with the fixed-point-based power model, we report a maximum absolute percentage error (APE) of 29% over all collected samples, with an average APE of around 9%, when compared to the analog power sensor of the board.

2.5.2.2 Monitoring overhead

Runmeter impacts the system's activity with overhead due to PMCs collection and manipulation for statistics and model estimation. As reported in D4.3 [2], the overhead for such activities in worst-case condition of extreme context switching accounts for a maximum of 0.7%, being generally much lower for typical scenarios.

For example, the overall runtime of Runmeter for idle contitions at 2.266 GHz is less than 0.4 ms per unit of time, accounting for 0.04 % of overhead for each CPU. The cost for the framework invocation is hence negligible in most of the cases, and complies with the KPI3.2, requiring minimal overhead for run-time monitoring (<1 %).

2.5.3 Support for other platforms

2.5.3.1 GPU and FPGA support

While demonstrated for the CPU power model, our methodology can be extended to GPU and FPGA support to fully address the heterogeneity of the target platform. However, the integration of such additional models in the Linux kernel introduces additional challenges due to their counters not being directly accessible from within the CPU domain. As a practical example, the Xavier's GPU PMCs are only accessible through NVIDIA's proprietary CUPTI API [15], which is not available at the Linux kernel level. Indirect approaches would then be required, which would introduce a performance overhead higher to the 1% threshold required by KPI3.2. For this reasons, analyzed more in dept in deliverable D4.3 [2], we propose a model which takes into account a simplified model for any additional accelerator, trading model accuracy for lower complexity and increased performance.

$$P = L + \sum_{i=1}^{\#cores} \left(g_i \cdot G_i + \sum_{j=i}^{\#PMCs} x_{ij} \cdot A_{ij} \right) + x_{GPU} \cdot A_{GPU} + x_{FPGA} \cdot A_{FPGA}$$

The independent parameters $x_accelerator$ represent the activity of a given accelerator with a simplified linear model. Such parameter can be a PMC directly selected from the counters locally available to the CPU that expose some degree of correlation with $accelerator$'s activity. As in the case of the FPGA, $x_accelerator$ can also be a metric summarizing the accelerator's activity directly computed by its internals and exposed to the CPU.

3 Evaluation of predictability of real-time tasks

This section describes the various experiments that were carried out to validate the AMPERE runtime effectiveness in ensuring predictability of the hosted applications. The effectiveness of the multi-criteria optimization strategy for the platforms developed in WP3, is convalidated applying the methodology to randomly generated synthetic workload scenarios, deployed on the FPGA-accelerated Xilinx UltraScale+ board. Finally, Section 3.3.1.1 shows how the secondary optimization goal of slack maximization can be conveniently leveraged to obtain more robust configurations.

3.1 Implementing Multi-DAG Real-Time Application Scenarios on Linux

For the purpose of evaluating the results produced by the optimization tool described in D3.4 [4], SSSA developed an application that emulates the behavior of a real application comprising one or multiple real-time DAGs running on Linux. This application, which we will refer to as RTDAG [16], reads the system configuration provided by the MIQCP solver developed by SSSA and starts several parallel threads of execution, each corresponding to one of the tasks in the original DAG specification.

In RTDAG, the source task of each DAG is activated periodically while the others execute according to the same three steps described in the previous section. In this case, the implementation of each task performs a series of operations on a set of matrices, a representative workload of a typical real-time image processing pipeline. The operations performed by each task are carefully calibrated offline to result in the expected execution times when running on the most powerful (big) core at the highest available frequency. For more information on the calibration of task execution times, see Section 3.1.1. Thanks to the accurate time scaling model described in D3.4 [4], if the tasks execute for the correct amount in that system configuration we expect them to behave according to that model when run at lower frequencies or on less powerful cores.

Each task executed by RTDAG limits operations on shared memory only at the beginning/end of its activation and performs mostly CPU-intensive operations in between. Each task that is not the DAG originator takes as input a set of matrices and produces another set of matrices as output for its successors. The synchronization operations between predecessors/successors in each DAG are implemented using condition variables. Only when all the predecessors of a task have completed their execution, the task is notified, thus it is unblocked and can proceed forward for its own computations.

The condition variables described above are defined per-task, in the sense that there is one for each task in each DAG. When one of the tasks is about to terminate, it iterates through its successors to check whether it is the last of that task predecessors to complete execution, and if so it signals the task using its associated condition variable. This implementation is necessary to avoid spurious wakeups of tasks in DAGs, meaning wakeups that may lead a task to block again after checking that not all its predecessors have completed execution. These spurious wakeups are to be avoided at all costs when implementing real-time tasks under the SCHED_DEADLINE scheduler. If a task like this were to wake up before it can actually start execution, the scheduler would assign it the wrong absolute deadline for sure, or at least one that does not match the expectation of our solvers. Thanks to this single-wakeup approach, we eliminate that possibility, resulting in SCHED_DEADLINE always assigning the correct absolute deadlines to DAG tasks.

3.1.1 Calibrating real-time tasks execution time

As mentioned in the previous section, the implementation of a real-time task provided by RTDAG is synthetic and as such it does not perform any meaningful manipulation of the data received in input by each task. In

reality, each task executing on the CPU[1] performs a static set of algebraic operations involving floating-point matrices, which we consider a "*tick*". This *tick* is considered the minimum amount of execution that a task can run for. For longer execution times, we can repeat the execution of a *tick* multiple times, approximating any execution time with an integer number of *ticks*. Selecting a very limited number of matrix operations as part of a *tick* is fundamental, so that the granularity of the execution times emulated by the synthetic tasks does not pose a concern.

Given a task set specification, RTDAG assigns to each synthetic task a number of *ticks* to run depending on its declared WCET. Since the WCET declared in each DAG specification is specified as the expected execution time of the task when executing on the most powerful (big) core at the highest frequency, we measured the average execution time t_{tick} needed by a *tick* in those conditions, then calculated the number of *ticks* a task τ_i has to run for, in order to exhibit a target execution time of C_i, as equal to

$$\left\lfloor \frac{C_i}{t_{tick}} \right\rfloor$$

This considers that $C_i^{ns} = 0$ for our synthetic workload, as in the basic *tick* computations we used matrices of limited size. Since the number of *ticks* assigned to each task is computed statically, independently of the frequency or CPU type on which the task will be deployed, the amount of work that a task will do at each activation is always the same, emulating the behavior of a real non-synthetic application.

RTDAG provides the option to precisely calibrate the value of t_{tick} when deployed on a new platform, by repeatedly executing the workload of a single *tick* and outputting a number of statistics on the measured execution times.

However, any estimate on the execution time of a real-time task (synthetic or not) running on Linux under SCHED_DEADLINE can be influenced by several factors, including: interfering activities generally known as "OS noise" [17], e.g., due to the execution of IRQs or non-preemptibility sections of the kernel [18]; other tasks that may be scheduled by a scheduler that has a higher priority than SCHED_DEADLINE[2]; cache-level interference due to other tasks (even lower-priority ones), running on the same or different cores causing an unforeseen volume of evicted L2/LLC CPU cache lines[3]. All of these factors may impact the execution times of tasks, that might sporadically happen to be longer than experienced in previous profiling campaigns.

In general, high-level models for task WCET should consider all of these factors when specifying a value, so that the indicated WCET is effectively what its name suggests, the worst-case, and no task may ever exceed it, no matter the situation. In our case, the WCET is specified by the DAG generation tool, so we have to reason in the opposite direction when selecting the number of *ticks* of execution for each task. If we used the simple formula indicated above using the average measured time t_{tick}, then a synthetic task within RTDAG would result in having its *average* execution time approximately equal to the specified C_i value, i.e., roughly half of the times we would expect the task to have a longer duration. Thus, to let C_i represent an upper-bound to a task execution time, including the possible sources of interference mentioned above, RTDAG applies a correction factor R in the formula for computing the target execution time:

$$R \cdot \left\lfloor \frac{C_i}{t_{tick}} \right\rfloor$$

where we verified experimentally that setting $R = 0.95$ was sufficient to avoid actual executions longer than the intended C_i values, at any frequency or core type for the desired platform.

[1] For tasks executing on an accelerator, no calibration was necessary, as they perform a fixed amount of work dictated by the implementation of the accelerated task. See the section Section 3.1.2 for more details.

[2] Typically SCHED_DEADLINE is the scheduler with the highest priority among the Linux schedulers, which means that no task scheduler by a different scheduler than SCHED_DEADLINE can ever preempt a task running under SCHED_DEADLINE. However, this behavior can be easily changed via a simple patch to the Linux kernel, as shown in Section 2.3.1.

[3] Cache-colouring techniques may be conveniently leveraged to mitigate these issues.

3.1.2 Accelerated tasks implementation

RTDAG provides also support for hardware-accelerated tasks. These tasks are implemented similarly to their pure-software counterparts, in that they each wait for the termination of all their predecessors before waking up, starting a computation intensive section, and signaling all their successors for completion. There are two key differences in how they are implemented with respect to the other tasks: first, they implement the computation intensive part by performing an accelerated call via an accelerator framework; second, they run at a priority higher than the one of other SCHED_DEADLINE tasks. As mentioned in Section 2.3.1, this change is necessary, regardless of the kind of task accelerator framework used, to ensure the validity of the configurations provided by the optimizer. Since many of these accelerator frameworks employ a client-server architecture, in which a server process manages a queue of acceleration requests from several other tasks, it is imperative that these processes execute at a higher priority than any other task started by RTDAG, in order for them to be treated as interference in the optimization tool. We used this approach also for the software part of the hardware-accelerated tasks part of RTDAG; this way, whenever a pure software task wakes up the thread that's supposed to be hardware-accelerated, there is the smallest delay possible between the request of starting a hardware-accelerated task and the actual request to the acceleration daemon process to run the requested task on the FPGA/GPU.

Since tasks scheduled using SCHED_DEADLINE may be preempted whenever the acceleration daemon is selected for execution, effectively the daemon can be considered an interrupt which may fire and interfere with SCHED_DEADLINE tasks. For this reason, it can be addressed in the same way we address IRQ interference in the optimizer. In general, for each platform this interference can be measured using some targeted experiments. It should be noted that any long software tasks may be subject to this kind of interference multiple times, if multiple hardware tasks are activated during its execution and the daemon executes on the same core. This kind of interference can of course be eliminated if the daemon is statically bound to a core that is never used by any software task for execution. If deemed necessary, the optimization framework may be configured to reserve a core for this and/or other activities on the system, albeit this would adversely impact the optimality of the found solutions.

RTDAG provides support for different kinds of accelerators implementing accelerated tasks using the following frameworks:

- tasks that should be executed on FPGA are implemented leveraging the FRED FPGA acceleration framework [3], described in D4.3 [2]. Leveraging this framework, one or multiple tasks in each scenario can be indicated to be always accelerated on FPGA, dynamically reconfiguring the hardware platform if necessary.
- Tasks that can be executed on the GPU leverage OpenMP to compile a set of operations that should be offloaded as a kernel on the GPU.

In general, RTDAG tasks execute repeated matrix multiplication operations to emulate the behavior of a typical real-time application. The size of the matrices to multiply can be chosen arbitrarily by the user, as well as the number of operations each task will perform on its private data. As mentioned in Section 3.1, RTDAG Regarding OpenMP-accelerable tasks, RTDAG users can choose whether a task will be executed on the CPU (sequentially) or the GPU, which is parallelized through OpenMP. An evaluation of the benefits of offloading this kind of operation to the GPU through OpenMP is included in D2.5 [19].

3.1.3 Jetson Xavier AGX

In this subsection, we perform an experiment highlighting the usefulness of the above mentioned kernel patch swapping the RT and SCHED_DEADLINE scheduling classes for our use-cases. Indeed, when performing offloading to a GPU device on the NVIDIA Jetson Xavier AGX board, our reference GPU-accelerated platform, we can see that a number of additional device driver threads are involved in the interaction with the GPU device. This can be shown with an experiment in which we trace using `perf` a simple applica-

tion performing a number of calls to a GPU-offloaded function. Precisely, tracing a simple offloaded matrix multiplication, we can find: `nvmap-bz`, `irq/203-gk20a_s`, `irq/56-host_syn`, `cuda-EvtHandlr`, `nvgpu_clk_arb_p`, `nvgpu_pg_init_g`, `nvgpu_channel_p`, `nvgpu_nvs_gv11b`.

Repeating the experiment with a variable number of calls to the offloaded function, we can also isolate which ones, among these, are responsible for performing an offloaded call, once the GPU device is already initialized and the GPU kernel to be executed does not need to be changed. As Figure 2 shows, it is clear that the `irq/56-host_syn` is involved at every offloaded function invocation.

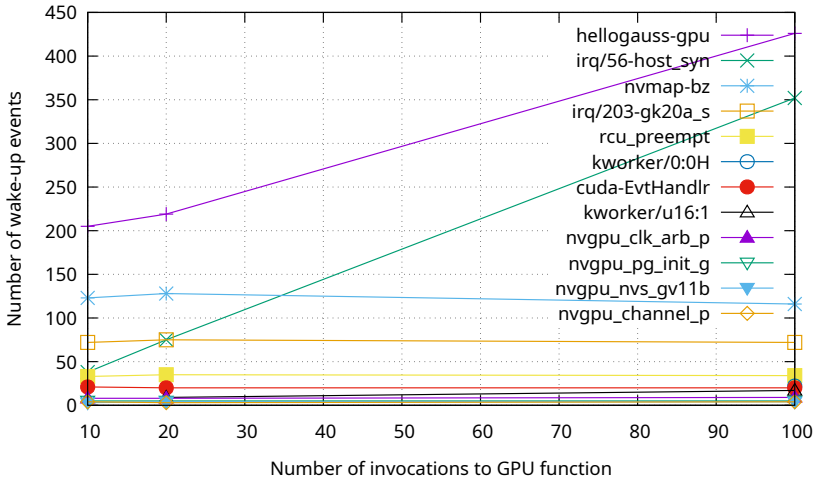


Figure 2: Number of occurrences of wake-up events (Y axis) for a number of kernel threads and processes (different curves), at varying numbers of invocations (X axis) of a GPU-offloaded matrix multiplication.

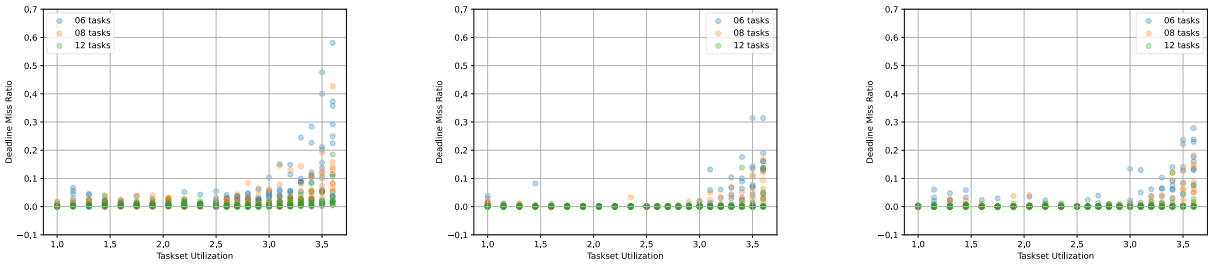
These kernel threads are activated several times but for a very short time, and their execution runs concurrently with our main application threads (which are executing regular CPU workloads, as the thread calling the GPU-accelerated function is temporarily suspended waiting for the result), thus they can interfere with any of the threads deployed on the platform. Therefore, these kernel threads interfere with our main workload similarly to how a device driver interferes with user-space applications.

In our methodology, we decided to consider the interference due to these kernel threads by inflating slightly the WCETs of regular CPU runnables, so that we increase slightly the robustness of the platform configuration to possibly unmodelled effects. However, to let these threads execute correctly, we need to give them a real-time priority, and configure the real-time scheduling class above the SCHED_DEADLINE one, so that our main application workload, deployed using SCHED_DEADLINE, is preempted by these kernel threads, and GPU acceleration can be performed on-time, as expected. This is achieved using the kernel patch described above.

The just described scenario is similar to how we dealt with the interferences due to the FRED server and FPGA reconfiguration kernel thread on the Xilinx board, where a similar configuration allowed us to consider these very small interruptions by just a small inflation of WCETs.

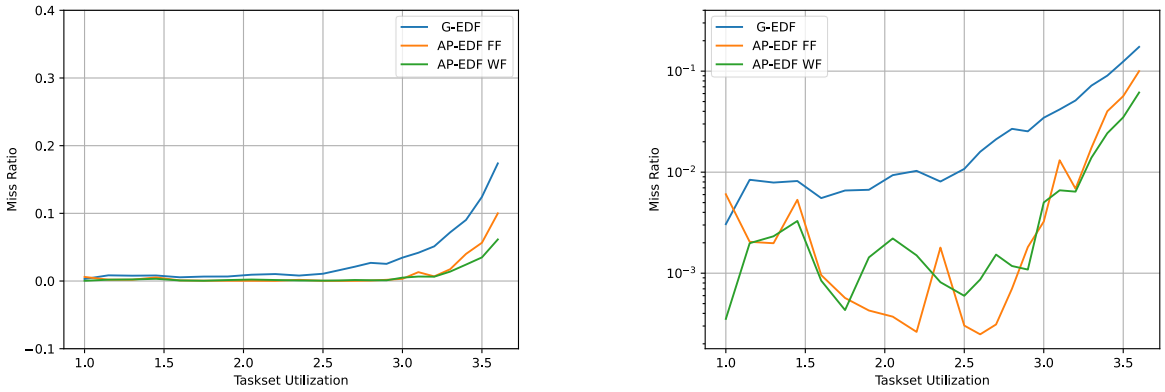
3.2 Evaluating APEDF performance

To evaluate the effectiveness of the patches described in Section 2.3.2, we performed several experiments on an ODROID-XU4 platform. On this platform, each CPU island is characterized by 4 CPUs sharing a single DVFS setting (meaning all CPUs on the same island must execute at the same frequency). For these experimentations, only the “big” cores have been used, turning off the “LITTLE” island; so for our purposes, the target platform is effectively a 4-core platform characterized by a single shared frequency among them. For these experiments, the frequency of 1.3 GHz has been selected as maximum frequency, to avoid thermal-throttling



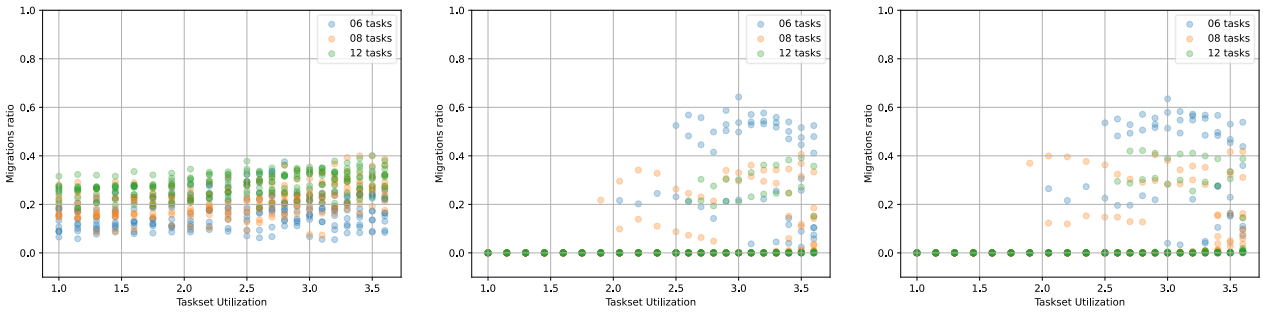
(a) G-EDF (b) APEDF FF (c) APEDF WF

Figure 3: Comparison between different scheduler implementations for SCHED_DEADLINE in terms of deadline miss ratios.



(a) (b)

Figure 4: Average deadline miss ratio for each SCHED_DEADLINE implementation. Plots refer to the same data, the only change being the Y axis (linear on the *left* plot and logarithmic on the *right* one).

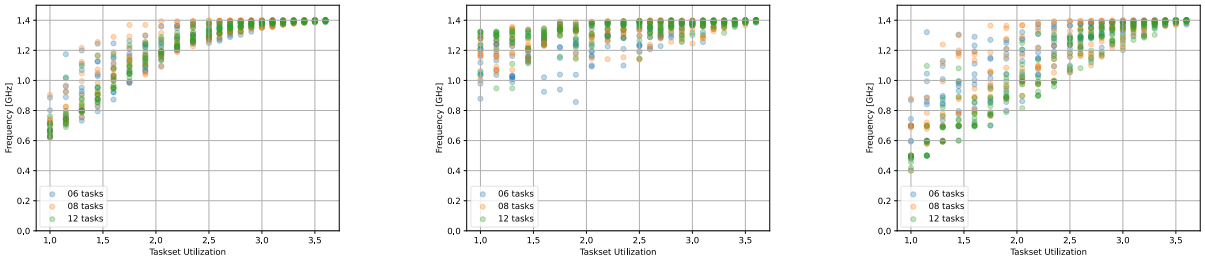


(a) G-EDF (b) APEDF FF (c) APEDF WF

Figure 5: Comparison between different scheduler implementations for SCHED_DEADLINE in terms of task migrations.

issues as described in Section 2.3.2.1.

Figure 3 compares the performance of an unpatched SCHED_DEADLINE implementation (i.e., using G-EDF as scheduling policy) against the APEDF implementation described in Section 2.3.2.2, either using First-Fit (FF) or Worst-Fit (WF) as task partitioning strategies. With each scheduler implementation, we executed several tasksets with increasing total utilization, from 1.0 up to 3.6, since the platform has 4 cores. In all experiments, the selected frequency governor is `schedutil`, with its default rate limit. In general, APEDF consistently out-



(a) G-EDF (b) APEDF FF (c) APEDF WF

Figure 6: Comparison between different scheduler implementations for SCHED_DEADLINE in terms of average frequency selection.

performs G-EDF, using either partitioning strategy, both regarding the number of deadline misses (see also Figure 4) and the number of task migrations (see Figure 5). In particular, with FF, we can see virtually no missed deadline up to the global utilization bound we expect from theory (~2.5); for the same tasksets, G-EDF tends to show misses even for very low system utilization.

Regarding DVFS performance, APEDF using FF, on average, selects higher frequencies (using `schedutil`), as shown in Figure 6 since it tends to pack all the tasks on the first core, and the tested platform has only one shared frequency island. For this kind of platform, WF selects, on average, lower frequencies than the other two strategies while retaining fewer deadline misses compared to G-EDF. This result is mainly due to the lower number of task migrations that characterizes APEDF, regardless of the partitioning strategy (see Figure 5).

3.3 Experimental Evaluation on Linux

All our experimental evaluations using RTDAG follow the same methodology, regardless of the target platform used for the evaluation. Starting from a representative set of task sets, composed of one or multiple concurrent DAGs, we run them through the optimization tool, which finds suitable runtime conditions to execute the task set (CPU/accelerators frequencies, task mapping, individual task deadlines, etc.).

For each task set, we execute on RTDAG on the target platform. If necessary, RTDAG will set on startup the frequency of each processing unit to reflect the one selected by the optimization tool. Once the platform is configured accordingly, it will start all the real-time tasks corresponding to each task in the task set. Each task is assigned the correct SCHED_DEADLINE parameters, as indicated by the optimizer. For each task set, several iterations of each DAG are executed one after the other, either for a set amount of time or until the hyper-period of all the DAGs part of the task set is reached. After RTDAG terminates, statistics related to end-to-end response times of each DAG are collected.

These operations have been performed on multiple target platforms, some which are the same target platforms of the project and some others which are used for further validation.

3.3.1 Evaluation on a ODROID-XU4 Platform

The ODROID-XU4 is a platform comprising on two DVFS-caplable ARMv7 CPU islands in big.LITTLE configuration. On this platform, we selected a set of about 200 task sets comprising one or multiple DAGs. Using the optimization tool described in D3.4 [4], we provided RTDAG with the correct configuration to execute each task set on the ODROID-XU4 board under SCHED_DEADLINE.

In order to validate also the expected average power consumption as indicated by the optimization tool, before starting each task set a monitoring application is executed on a secondary machine. This application monitors

the power consumed by the ODROID-XU4 board using an external power monitor connected to its power supply.

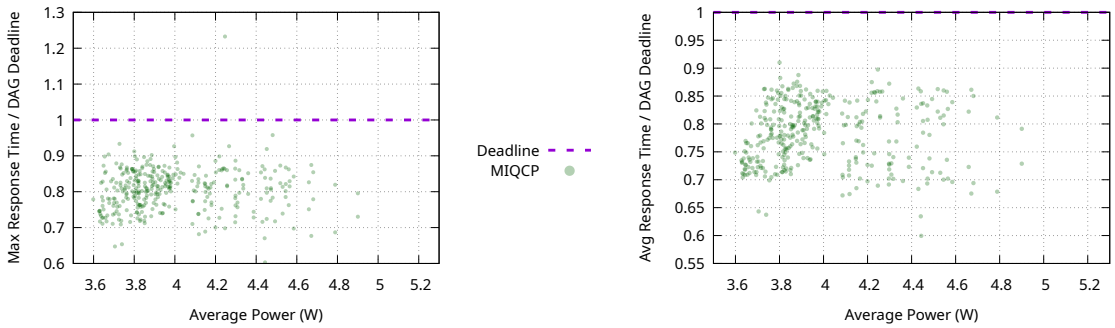


Figure 7: Maximum (left) and average(right) relative response times (Y axis) vs average power consumption (X axis).

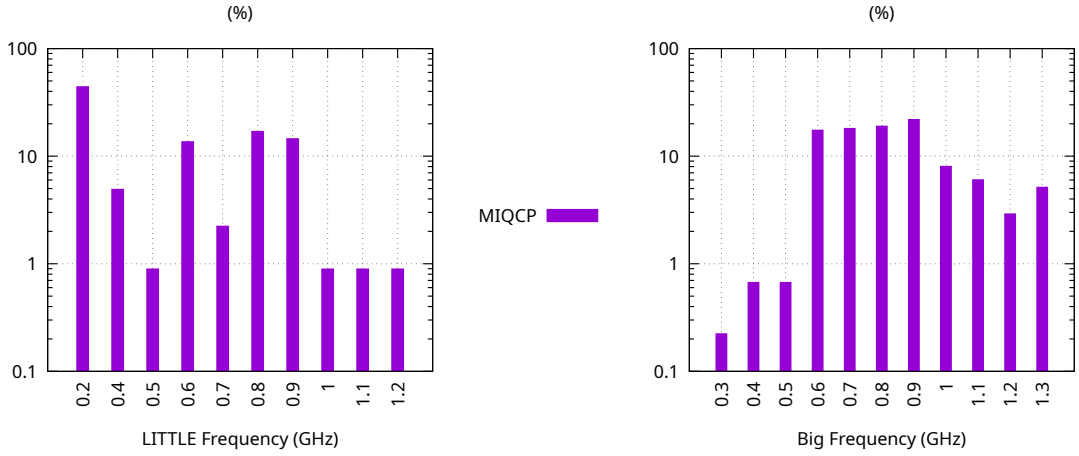


Figure 8: Chosen frequencies for the two islands

In Figure 7, we report the relative DAG response times (end-to-end response-time of each DAG divided by its deadline, on the Y axis) versus the measured power consumption (on the X axis), for the scenarios we ran. The left and right plots report the maximum and average relative response times, respectively, on the Y axis. Since the optimization objective in this case is that of minimizing the average power consumption of each task set, each CPU island is kept at a relatively low-power frequency setting. A visualization of the frequency selection performed by the optimizer is provided in Figure 8, which represents the number of times a certain frequency is selected by the optimizer using histograms for each island. From the plot it is clear that the optimizer makes very little use of the higher frequencies available on each CPU island.

This preference for low operating frequencies unavoidably results in each DAG having relatively long response times, close to the DAG end-to-end deadlines (the dashed line at 1.0 on the Y axis in Figure 7). In just 3 of the 200 scenarios we ran we observed deadline misses, which were not expected according to the optimization tool. The optimizer however has a certain tolerance that cannot be taken into account when moving on to the target platform. To address this issue, the secondary optimization goal of minimum slack maximization can be conveniently leveraged to obtain more robust configurations.

3.3.1.1 Maximizing robustness under power budget constraints

To evaluate the robustness improvement provided by the secondary optimization goal, we re-optimized 19 out of the 200 scenarios considered in our evaluation above. The selected scenarios are the ones characterized by the smaller slack found by the first optimization goal, including the 3 scenarios that exhibited sporadic deadline misses when executed on the ODROID-XU4 board. These scenarios were re-optimized, this time indicating to the optimizer to maximize each DAG slack. To avoid losing the benefits of the first optimization, the minimum average power consumption found for each task set has been provided to the optimizer as a power budget constraint, which cannot be exceeded. For each of these task sets, the optimizer found solutions with (typically) better response times, without exceeding the provided power budget. Figure 9 reports the minimum and maximum values (vertical segments) of the maximum relative slack obtained in repeated runs of each scenario on the ODROID-XU4 platform. In the plot, the red segments indicate scenarios optimized only for the minimum power, while the green ones are optimized both for power and slack. Visually, it is evident that the scenarios optimized also for slack resulted in a general lower (or sometimes equal) statistics of the obtained relative slack. Additionally, the few scenarios that originally missed their deadlines under power-only optimization, turned out to stay safely below the deadline limit, in the re-optimized configuration maximizing slack (at an equal average power consumption).

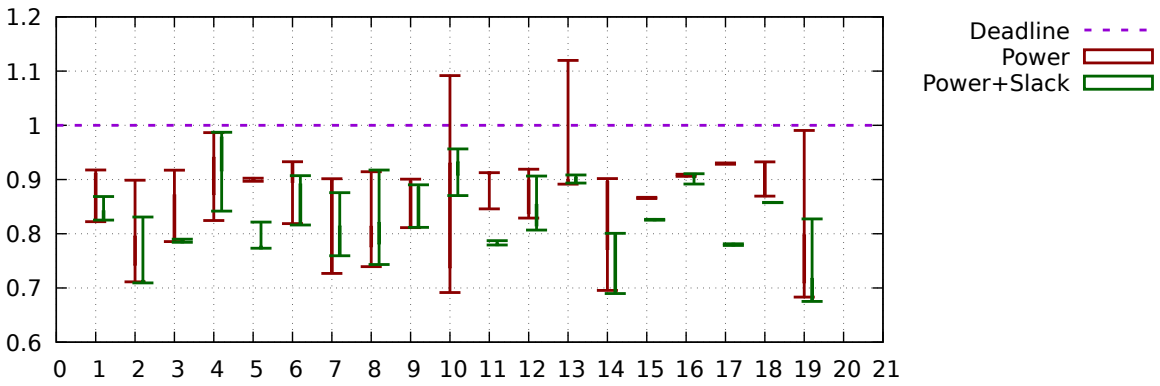


Figure 9: Maximum response times relative to DAG periods (Y axis) for various DAGs (X axis), with and without slack optimization (green and red lines, respectively), in addition to power optimization.

3.3.2 Evaluation on the Xilinx UltraScale+ Platform

The Xilinx UltraScale+ ZCU102 board, one of the target platforms of the AMPERE project, provides access to its programmable logic subsystem to execute FPGA-based tasks. Access to this functionality is obtained by leveraging the FRED framework. In addition, for the purposes of the AMPERE project this platform is to be used by executing a Linux guest on top of PikeOS hypervisor.

To validate the expected behavior of real-time tasks executing on Linux on this platform, we performed experiments running several optimized task sets using RTDAG. To evaluate the impact that different components may have on the predictability of real-time workloads executed with RTDAG, our experimentation can be divided into 4 different sets of experiments:

1. scenarios with no FPGA-accelerated tasks executing on Linux “bare-metal” (i.e., with no PikeOS);
2. scenarios **with** FPGA-accelerated tasks executing on Linux “bare-metal”;
3. scenarios with no FPGA-accelerated tasks executing on Linux on top of PikeOS;
4. scenarios **with** FPGA-accelerated tasks executing on Linux on top of PikeOS.

An evaluation of the impact of PikeOS on individual real-time tasks executing either on the CPU or FPGA-accelerated using FRED can be found in D5.4 [1]. The purpose of these evaluations is to assess the impact of virtualization on more complex scenarios in which several real-time tasks are involved.

For all these tests, the calibration of the execution time of software tasks has been carried out once without PikeOS, so that the same amount of work is performed in corresponding scenarios. For simplicity we tested these scenarios always selecting the maximum available frequency for both the CPU and the FPGA.

3.3.2.1 DAGs with no FPGA-accelerated tasks

In these first scenarios, we generated several software-only tasksets and optimized them for execution on the Xilinx UltraScale+ board at the maximum frequency. We then executed them using RTDAG multiple times to test how much PikeOS impacts their performance. Overall, we measured an increase of the end-to-end DAG response times of about 2 %, which is slightly higher than the overhead that we evaluated for single-task applications in D5.4 [1].

3.3.2.2 DAGs with FPGA-accelerated tasks

For tests involving at least one FPGA-accelerated task, we generated some more task sets and optimized them for the minimum average power consumption once executed on the UltraScale+ platform. We then executed them, just like the previous sets, both with and without PikeOS mediation.

In this case, the FRED daemon application is started and executed at a higher priority than RTDAG software tasks, as indicated in Section 3.1.2. The hardware IPs used as hardware tasks are the same described in D5.4 [1], for which we already know the overhead introduced by PikeOS, both in terms of reconfiguration cost and in terms of regular execution time. In all the scenarios that we optimized, the solver chose to avoid leveraging the reconfiguration capabilities of the FRED framework and accelerate only one among the tasks with a possible hardware implementation.

Table 2: Behavior of real-time tasks **with** one FPGA-accelerated task executing on the Xilinx UltraScale+ platform when executing with or without the mediation of the PikeOS hypervisor.

Taskset	DAG Period [ms]	Average Response Times [ms]	
		Without PikeOS	With PikeOS
1	120	103.158	MISS
2	100	76.101	77.017
3	120	103.125	104.785
4	90	76.015	77.623
5	120	103.045	106.759

Table 2 shows a subset of the scenarios that we tested. For each scenario in the table, we show measured average DAG end-to-end response times, comparing values measured when executing the same scenario either with or without PikeOS. In this case, the measured overhead is a mix of the contributions of the overheads incurred when executing FPGA-accelerated calls in D5.4 [1] and the overhead measured above for software-only multi-task scenarios. In this case, applications can be optimized for robustness, as described in Section 3.3.1.1, to tolerate as much overhead as possible. However, that might not be sufficient. For example, see scenario 1 in Table 2, where the resulting DAG systematically misses its end-to-end deadline (equal to the DAG period), even after re-optimizing the scenario for robustness sake.

To address this issue, the overhead introduced by PikeOS on individual tasks, either software or hardware-accelerated (as described in D5.4 [1]) should be taken into account by the optimization tool. This can be done

by “inflating” expected tasks WCETs before supplying them to the optimizer. We repeated this step, optimizing scenario 1 both for power and for robustness using the “inflated” execution times and with the optimizer signaled that indeed the original solution that it provided was not feasible under the original conditions. It instead found a different solution, which we verified this time to be feasible by repeating the experiment on the target platform.

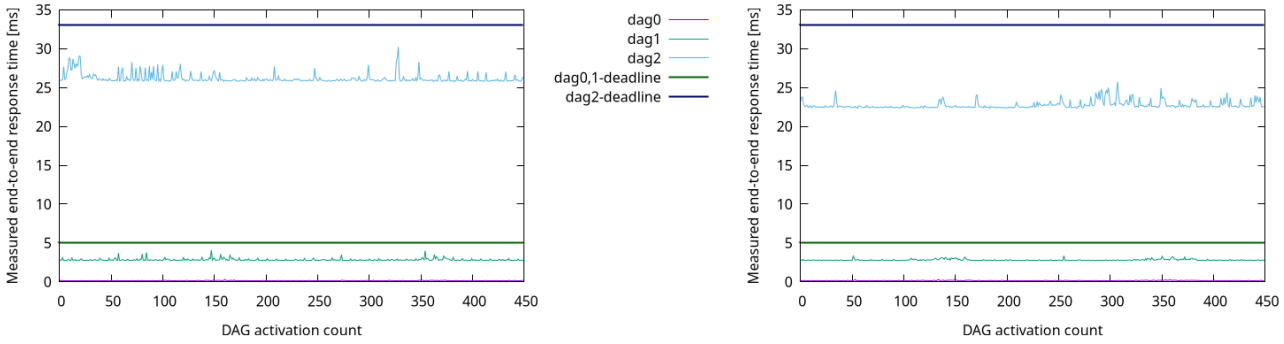
3.3.3 Evaluation of the PCC Use-Case

This section evaluates the performance of the PCC use-case when deployed on the two target platforms of the AMPERE project. The tasks to execute are emulated via RTDAG (see Section 3.1) and depending on the selected platform they can be offloaded to the GPU via OpenMP or to the FPGA accelerator via FRED.

3.3.3.1 Evaluation on the Xavier AGX Platform

Optimization Objective	DAG	Response Time [ms]			Deadline [ms]
		Minimum	Average	Maximum	
Minimum Power	0	0.138	0.156	0.791	5
Minimum Power	1	2.725	2.856	4.863	5
Minimum Power	2	25.770	26.161	30.213	33
Maximum Robustness	0	0.140	0.162	0.460	5
Maximum Robustness	1	2.719	2.816	4.885	5
Maximum Robustness	2	22.315	22.752	25.723	33

Table 3: Response times measured on the NVIDIA AGX Xavier platform for the PCC use case, depending on the optimization objective.



(a) Minimum average power

(b) Maximum robustness

Figure 10: Execution of the PCC use case with RTDAG on the NVIDIA AGX Xavier platform, depending on the optimization objective.

The output configuration provided for the PCC Use Case in D3.4 [4] has been executed on the Xavier AGX target platform, emulating each task using RTDAG and deploying accelerated tasks on the GPU using OpenMP pragmas. Extensive experimentation has been performed using the optimization output for the minimum average power consumption and the maximum robustness (Sections 3.3.1 and 3.3.2, respectively, of D3.4).

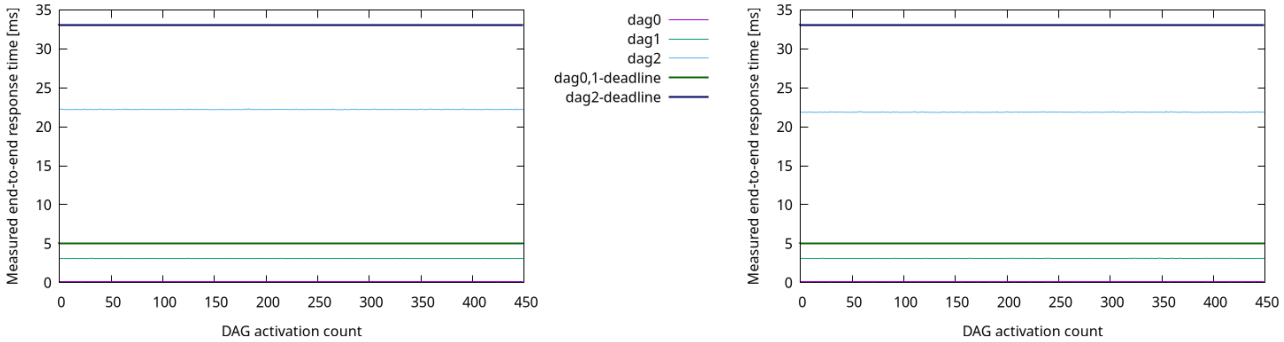
Table 3 shows the collected data for each of the DAG sets that characterize the PCC use-case. As you can see, all the solutions found by the optimizer do respect the respective end-to-end DAG deadlines once deployed on the target platform. When optimizing for the minum average power consumption, the solver did not place

any task on the GPU, discouraged by the high power consumption associated with that platform component. On the other hand, when we optimized the use-case for maximum robustness the solver did offload one of the runnables (of DAG n. 2) to the GPU. This indeed results in a faster end-to-end response time, both on the average case and with respect to the maximum registered value by RTDAG during the execution of several repeated DAG set activations, at the cost of slightly increased power consumption. A plot representing the response times measured per DAG activation is shown in Figure 10.

3.3.3.2 Evaluation on the Xilinx UltraScale+ Platform

Optimization Objective	DAG	Response Time [ms]			Deadline [ms]
		Minimum	Average	Maximum	
Minimum Power	0	0.143	0.150	0.172	5
Minimum Power	1	3.087	3.099	3.138	5
Minimum Power	2	22.150	22.216	23.086	33
Maximum Robustness	0	0.142	0.149	0.157	5
Maximum Robustness	1	3.096	3.110	3.154	5
Maximum Robustness	2	21.823	21.866	22.828	33

Table 4: Response times measured on the Xilinx UltraScale+ ZCU102 platform for the PCC use case, depending on the optimization objective.



(a) Minimum average power

(b) Maximum robustness

Figure 11: Execution of the PCC use case with RTDAG on the Xilinx UltraScale+ ZCU102 platform, depending on the optimization objective.

The same methodology described in the previous section has been used to evaluate the performance of an emulated DAG set deployed according to specification provided by the optimizer on the other target platform of the project, the Xilinx UltraScale+ ZCU102. Again, two different deployment configurations have been selected by the optimizer, depending on the optimization objective: one that minimizes the power consumption and one that maximizes the robustness of the system with respect to real-time constraints.

For FPGA-accelerated tasks, we used the same IPs described in Section 3.3.2. Tasks that in the original specification provide both a software-based and an accelerated implementation, the same function is implemented both as a sequential software function and as an hardware IP (e.g., matrix multiplication). The right implementation is then selected at runtime according to the deployment configuration selected by the optimizer.

Table 4 shows the collected data for each of the DAG sets executed on the Xilinx UltraScale+ platform. Also in this case, we confirmed that once deployed on the target platforms the applications behave as expected and respect their end-to-end deadlines. Again, we see that the configuration found for the minimum average power consumption objective does not take advantage of the FPGA accelerator present on the platform, while the

one that maximizes the robustness does select one runnable (of DAG n. 2) to execute on the FPGA. Similarly to the other platform, the solution that maximizes the system robustness does present faster average and maximum end-to-end response times, at the cost of slightly increased power consumption. Figure 11 shows the measured response times per DAG activation depending on the selected configuration.

3.4 Experimental evaluation of dynamic mapping algorithms

This section analyzes the performance of two mapping algorithms, based on an actual implementation in the LLVM-based runtime of AMPERE. This implementation considers the heuristics from D3.4 [4] which have performed better in the simulations. This implementation considers separate per-thread queues, similar to LLVM basic (dynamic) algorithm, providing therefore heuristics for both the allocation phase (selecting which thread to map) and dispatching phase (in each thread selecting the next task part to execute). For the allocation phase, two best fit scheduling heuristics are considered, which take into account the minimum total execution time (MTET) of the queues, and the minimum number of task parts (MNTP). For the dispatching phase, a minimum execution time (MET) algorithm is implemented.

The heuristics are compared to LLVM's default scheduler which implements a version of the work-stealing scheduler, which is known to be highly performant [20]. The implementation uses the AMPERE adapted version of LLVM , which contains additional specifications for the clang front-end and the OpenMP runtime to support the creation of the TDGs. This has no impact in the scheduler and the evaluation in this section. Note that the work-stealing feature of the LLVM OpenMP runtime is disabled for the execution of the mapping algorithms, to reduce the variability in the runtime executions.

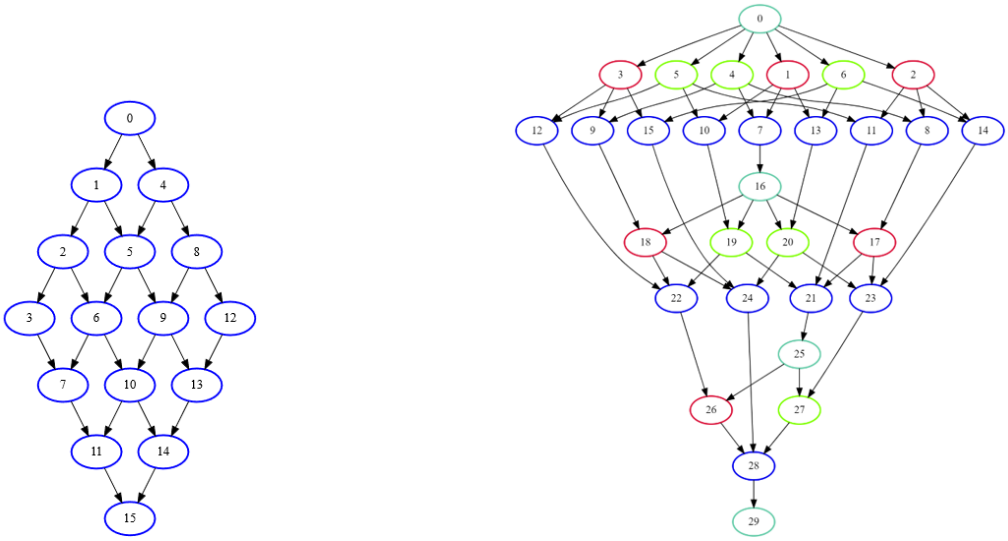
This evaluation considers as benchmarks, the Heat and SparseLU kernels from real-world applications, provided in the AMPERE taskgraph benchmarks [4]. The Heat application includes a Heat diffusion simulator implemented with Gauss-Seidel method, which shows a Stencil computation. The SparseLU application contains a SparseLU matrix decomposition, which shows an irregular form of parallel tree, retracting to one final task. Heat is an application with a regular TDG structure (Figure 12a, in the evaluated case the number of tasks in the TDG is 640 and the number of data dependencies between the tasks is 2128), while the latter is an application with a complex configuration (Figure 12b, in the evaluated case, the number of tasks is 1496 and the number of data dependencies is 3960). The number of tasks and block sizes was configured to allow for a sufficiently high number of tasks with relevant individual processing.

The applications are executed on an NVIDIA Jetson AGX Xavier with an 8-core NVIDIA Carmel Arm v8.2 64-bit CPU and 32GB 256-bit RAM, with 4 and 8 threads and under spread and close bindings. The applications are executed using the off-the-shelf operating system for the board (Linux), executing in the highest real-time priority class, with the used platform being fully dedicated to the experiment.

To enhance the confidence of results, each evaluation is executed in r*i iterations, where r is the number of runs (30 runs, in this evaluation) and i is the number of iterations on each run (50 iterations for each run, in this evaluation). The first 10% experiments are also removed from each run to avoid anomaly numbers at the warm-up stage, as well as the outliers (due to spurious kernel events) are removed from all the results. Both MTET and MET algorithms use execution time as the mapping metric. In order to reduce runtime overhead, mapping is based on offline estimation of worst-case execution time (as shown in D3.4[4]).

Figure 13 shows the response times of the Heat application under different configurations. The outliers are removed from the results (2.93% for Dynamic in '4 close', where the response time is above 7 s). The results indicate that the application response time is decreased in all the cases by increasing the number of threads, showing the scalability of the algorithms. The difference between the heuristics and the Dynamic algorithm (LLVM's default scheduler) is not very considerable in the first case, while it is noticeable in the remaining cases. The reason is that (i) less cache misses (i.e., L2 Data Cache) in the first case, where the spread binding is used, which cause the algorithms to behave very similarly, and (ii) Dynamic uses a work-conserving policy

[4] https://gitlab.bsc.es/ampere-sw/wp2/general-information/



(a) Example of Heat DAG with 16 tasks.

(b) Example of SparseLU DAG with 30 tasks.

Figure 12: Sample DAGs

without considering the temporal features (e.g., task execution time) in the mapping process, which causes to increase the response time rather than the heuristics. Furthermore, the performance of MTET-MET is slightly better than MNTP-MET in the cases with 4 threads (where most of the threads are busy during the mapping process), while MNTP-MET works slightly better in the cases with 8 threads (where some of the threads may be idle in the mapping, depending on the number of tasks executing in parallel). This achievement indicates that selecting the thread based on considering the total execution time of tasks in the queues is a suitable procedure for the mapping when the number of threads is low, but selecting the thread with considering the number of tasks in the queues is an appropriate strategy when the number of threads is high. The reason is that the overhead in MNTP (by considering the number of tasks in each queue, which is easier to determine) is lower than that in MTET (by calculating the total execution time of tasks in each queue based on the execution time of each task). However, this overhead is mostly sensible in the results by increasing the number of threads to 8.

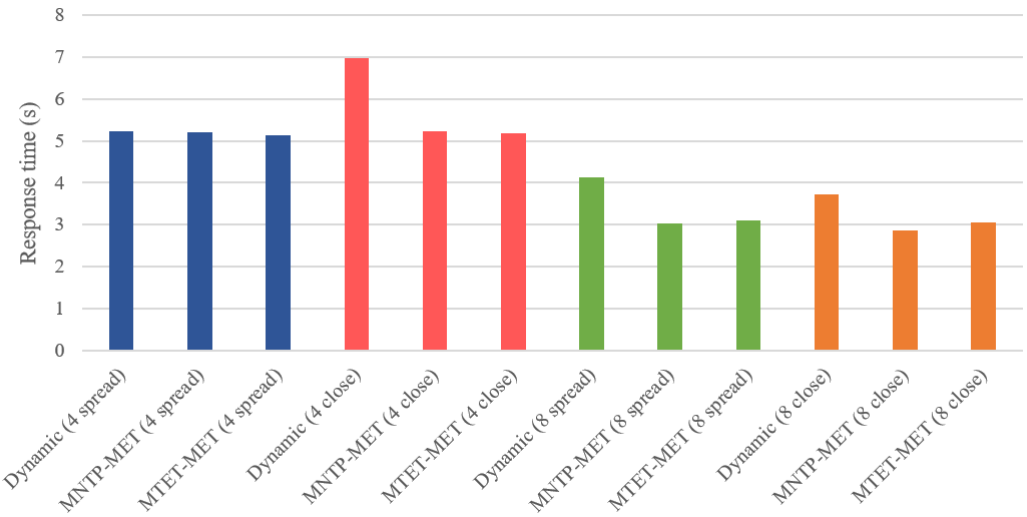


Figure 13: Results of evaluation with Heat

Figure 14 illustrates the implementation results for the SparseLU application. The outliers are removed from

the experiments (0.2% for Dynamic, 0.67% for MNTP-MET, and 0.13% for MTET-MET in ‘4 spread’ where the response time is above 17 s, 0.2% for Dynamic in ‘4 close’ where the response time is above 17 s, 0.53% for MNTP-MET in ‘8 spread’ where the response time is above 8 s, as well as 0.27% for MNTP-MET in ‘8 close’ where the response time is above 8 s). The results represent the scalability of the algorithms, similar to the Heat application. They show that the performance of the heuristics is higher than Dynamic, where the number of threads is 4, while Dynamic works slightly better, where the number of threads is 8. However, the difference between them in the cases with 8 threads is not very high (the difference between Dynamic and MNTP-MET in ‘8 spread’ is 1.08686 s and the difference between Dynamic and MTET-MET in ‘8 close’ is 0.142967 s). The reason is that there is a considerable number of tasks executing in parallel, in this application, but the heuristics work efficiently when the number of threads is not high (that is, the workload in the queues is high). In addition, the efficiency of MTET-MET is better than MNTP-MET in the results, except in ‘8 spread’ (however, their difference in this case is 0.429553 s). The reason is that MTET-MET considers task execution time for selecting the thread, and therefore it works well in the applications with a complex configuration, where the workload in the allocation queues is high. Note that the efficiency of MTET would overcome its weakness on overhead (rather than MNTP) in the applications with a complex configuration.

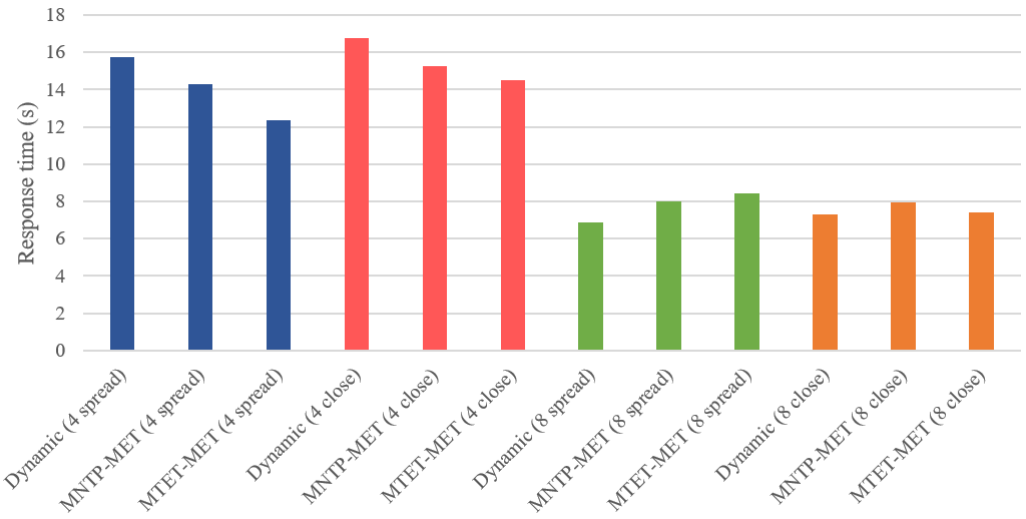


Figure 14: Results of evaluation with SparseLU

4 Conclusions

In this document, we have summarized the latest activities carried out by AMPERE partners in the context of WP4, contributing to the design and implementation of the final run-time architecture supporting deployment of real-time workloads on highly heterogeneous platforms with GPU and FPGA acceleration. We showed results from the experimental evaluation carried out by partners on various components of the final run-time architecture that have been realized in AMPERE, including experimentation with synthetic workloads and benchmarks, as well as with models coming from the AMPERE use-cases. From these results, we can conclude that the AMPERE final run-time architecture, coupled with the off-line toolchain described in WP2 and WP3 deliverables (and evaluated in D3.4 [4]), and the operating system and hypervisor components described in WP5 deliverables (and evaluated in D5.4 [1]), is a viable and suitable solution to deploy software components on highly heterogeneous platforms, under a set of non-functional constraints including real-time responsiveness, reliability and energy efficiency.

5 Acronyms and Abbreviations

BSP	Board Support Package
CPU	Central Processing Unit
CUPTI	CUDA Profiling Tools Interface
CSV	Comma Separated Values
D	Deliverable
DPR	Dynamic Partial Reconfiguration
DVFS	Dynamic Voltage and Frequency Scaling
EDF	Earliest Deadline First
FPGA	Field-Programmable Gate Array
GPU	Graphics Processing Unit
KPI	Key Performance Index
MDE	Model-Driven Engineering
MET	Minimum Execution Time
MILS	Multiple Independent Levels of Safety/Security
MNTP	Minimum Number of Task Parts
MPSoC	Multi-Processor System on Chip
MTET	Minimum Total Execution Time
MS	Milestone
NFR	Non-Functional Requirement
ODAS	Obstacle Detection Avoidance System
OPP	Operating Performance Points
OS	Operating System
PAPI	Performance API
PL	Programmable Logic
PMC	Performance Monitoring Counter
PMU	Platform Management Unit
PS	Processing System
SLG	Synthetic Load Generator
SoC	System on Chip
T	Task
TDG	Task Dependency Graph
WP	Work Package
WCET	Worst-Case Execution Time

6 References

[1] AMPERE, "Deliverable D5.4, Evaluation of the operating systems and hypervisors," June 2023.

[2] ——, "Deliverable D4.3, Integrated run-time energy support, and predictability, segregation and resilience mechanisms," September 2021.

[3] A. Biondi, A. Balsini, M. Pagani, E. Rossi, M. Marinoni, and G. Buttazzo, "A framework for supporting real-time applications on dynamic reconfigurable fpgas," in *Proc. of the IEEE Real-Time Systems Symposium (RTSS 2016)*, December 2016, pp. 1–12.

[4] AMPERE, "Deliverable D3.4, Evaluation of multi-criteria optimizations," June 2023.

[5] C. Scordino, L. Abeni, and J. Lelli, "Energy-aware real-time scheduling in the linux kernel," 2018.

[6] K. Yang and J. Anderson, "On the soft real-time optimality of global edf on uniform multiprocessors," in *2017 IEEE Real-Time Systems Symposium (RTSS)*, 2017, pp. 319–330.

[7] S. Tang, J. H. Anderson, and L. Abeni, "On the defectiveness of sched_deadline w.r.t. tardiness and affinities, and a partial fix," 2021.

[8] Linux kernel community, ""deadline task scheduling" from the linux kernel documentation," 2023. [Online]. Available: https://docs.kernel.org/scheduler/sched-deadline.html

[9] L. Abeni and T. Cucinotta, "Adaptive partitioning of real-time tasks on multiple processors," in *Proceedings of the 35th Annual ACM Symposium on Applied Computing*, ser. SAC '20. New York, NY, USA: Association for Computing Machinery, 2020, p. 572–579. [Online]. Available: https://doi.org/10.1145/3341105.3373937

[10] A. Stevanato, T. Cucinotta, L. Abeni, and D. B. de Oliveira, "An Evaluation of Adaptive Partitioning of Real-Time Workloads on Linux," in *2021 IEEE 24th International Symposium on Real-Time Distributed Computing (ISORC)*, 2021, pp. 53–61.

[11] AMPERE, "Deliverable D3.3, Energy optimisation framework, predictable execution models and analysis, and Software resilient techniques," September 2022.

[12] ——, "Deliverable D2.3, Programming model extensions and the multi-criteria performance-aware component," September 2022.

[13] NVIDIA Corporation, "Jetson AGX Xavier developer kit," 2018. [Online]. Available: https://developer.nvidia.com/embedded/jetson-agx-xavier-developer-kit

[14] AMPERE, "Deliverable D5.1, Reference parallel heterogeneous hardware selection," 2020.

[15] NVIDIA Corporation, "CUPTI v11.6 user guide," 2021. [Online]. Available: https://docs.nvidia.com/cupti

[16] 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

[17] D. B. de Oliveira, D. Casini, and T. Cucinotta, "Operating system noise in the linux kernel," *IEEE Transactions on Computers*, vol. 72, no. 1, pp. 196–207, 2023.

[18] D. B. de Oliveira, D. Casini, R. S. de Oliveira, and T. Cucinotta, "Demystifying the Real-Time Linux Scheduling Latency," in *32nd Euromicro Conference on Real-Time Systems (ECRTS 2020)*, ser. Leibniz International Proceedings in Informatics (LIPIcs), M. Völp, Ed., vol. 165. Dagstuhl, Germany: Schloss Dagstuhl–Leibniz-Zentrum für Informatik, 2020, pp. 9:1–9:23. [Online]. Available: https://drops.dagstuhl.de/opus/volltexte/2020/12372

[19] AMPERE, "Deliverable D2.5, Evaluation of performance-aware model transformations," June 2023.

[20] C. Lattner and V. Adve, "Llvm: A compilation framework for lifelong program analysis & transformation," in *Proceedings of the International symposium on code generation and optimization*, 2004.