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

D4.2 Independent run-time energy support, and predictability, segregation and resilience mechanisms

Version 1.0

Documentation Information

Contract Number	871669
Project Website	www.ampere-euproject.eu
Contratual Deadline	31.03.2021
Dissemination Level	[PU]
Nature	DEM
Author	Alexandre Amory, Tommaso Cucinotta, Alessandro Biondi (SSSA)
Contributors	Sara Royuela (BSC) Björn Forsberg (ETHZ) Viola Sorrentino (THALIT) Alessio Medaglini (UNISI) Enkhtuvshin Janchivnyambuu (SYS) Tiago Carvalho and Miguel Pinho (ISEP) Marco Pagani (SSSA)
Reviewer	Sara Royuela (BSC)
Keywords	run-time, hypervisor, FPGA, pedestrian tracking, dynamic partial reconfiguration

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

Change Log

Version	Description Change
V0.1	Initial skeleton and contents sketch
V0.2	First consolidated draft after integration of contributions from partners
V0.3	Formatting adjustments to comply with the official deliverable template
V1.0	Final version after integration of reviewer's comments

Table of Contents

1 Introduction

The AMPERE project aims at providing, among others, mechanisms capable of simplifying development of embedded high-performance applications under non-functional constraints (timing, reliability and energy), with the help of predictable execution models being manipulated in a model-based design flow. To that aim, WP2 is studying annotations and transformations at the model level, and tools needed at the code synthesizer and compiler levels, in order to generate predictable and/or reliable code or, if not possible, at least code augmented with self-monitoring capabilities. Furthermore, WP3 is in charge of investigating predictable execution models to address timing constraints, including microprocessors and accelerators, and timing analyses able to support DNN workloads. Finally, WP4 is responsible for designing and realizing a run-time architecture with the ability to: manage at run-time the predictability and reliability of the deployed parallel applications as analyzed in WP3; support the predictable and energy-efficient execution of heterogeneous applications, also coordinating their access to GPU-offloaded or hardware-accelerated functions deployed using dynamic partial reconfiguration (DPR) of FPGA fabrics; implement mapping and scheduling algorithms that ensure the predictability of the execution models analysed in WP3; provide adaptive run-time mechanisms able to cope with dynamicity in the deployed workload and/or environmental conditions.

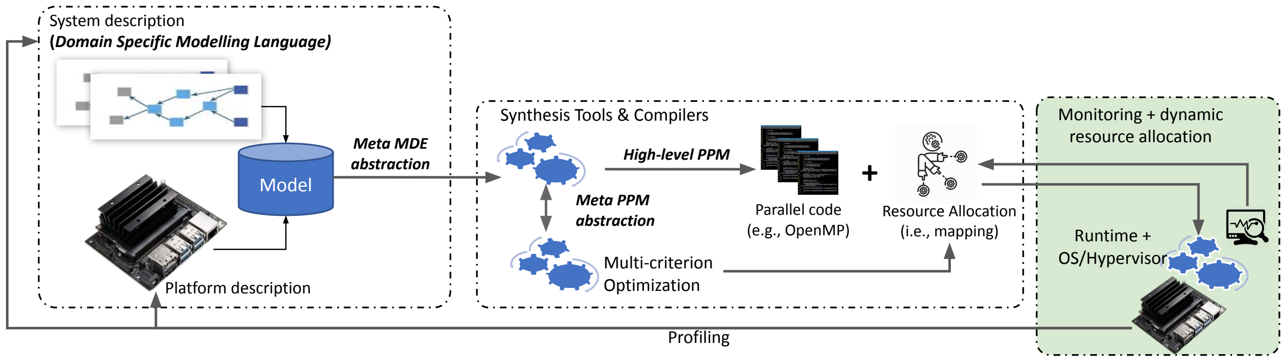
This document corresponds to Deliverable 4.2, targeting milestone MS2 (i.e., from M7 until M27). The Deliverable 4.2 consists of a set of independent methods containing:

1. Dynamic energy management policies;
2. Runtime mechanisms and scheduling algorithms for managing computation and communication in a predictable way;
3. Segregation mechanisms for safety/security support;
4. Mechanisms for resilience support.

The green box of Figure 1 presents D4.2 in the context of the overall AMPERE software architecture. This deliverable comprises three tasks that contribute to D4.2 and D4.3, described next:

- Task 4.2. Run-time energy support (m7:m27);
- Task 4.3. Run-time for predictable parallel heterogeneous computing (m7:m27);
- Task 4.4. Run-time resilience methods (m7:m27).

Figure 1: AMPERE software architecture.


System description
(Domain Specific Modelling Language)
Model
Platform description
Meta MDE abstraction
Synthesis Tools & Compilers
High-level PPM
Meta PPM abstraction
Parallel code (e.g., OpenMP)
+
Resource Allocation (i.e., mapping)
Multi-criterion Optimization
Monitoring + dynamic resource allocation
Runtime + OS/Hypervisor
Profiling

Task 4.2 builds on top of (a) available sensing, e.g., hardware counters, power monitors, etc., (b) platform-specific and workload-specific parameters such as heterogeneity, e.g., type and number of cores, inherent system parallelism, etc., and (c) actuation knobs, e.g., shutdown, dynamic voltage and frequency scaling (DVFS), etc., to implement the run-time framework supporting the energy models developed in Task 3.2. The aim of the run-time mechanisms is to augment the platform monitoring and introspection capabilities, and to deploy scheduling policies derived from the multi-criteria analysis. Finally, this framework is used to test, monitor and implement efficient resource management strategies in Task 3.2. The target at MS2 is a set of mechanisms addressing only energy optimizations.

Task 4.3 implements the support for managing the predictable framework developed in Task 3.3 for parallel heterogeneous computing. The task developed: (1) run-time techniques to allow proper management of platform resources at run-time, and (2) innovative mapping and scheduling algorithms to jointly allocate and schedule the computation and communication phases of the execution models analysed in Task 3.3. Such execution models are of little (if no) use without a global framework that monitors and manages the access to computing and shared resources in a predictable and efficient way. For this purpose, this task designs smart computation and communication mapping algorithms to reduce contention on accessing shared resources. To address FPGA predictable reconfiguration, a novel design for FPGA slots is realized for controlling the execution of hardware accelerators and their DPR, while handling data communication. Then, a set of drivers is developed to handle the shared-memory communication between the processors and the FPGA and to manage the dynamic reconfiguration of the FPGA slots; and a daemon is implemented for managing the reconfiguration requests issued at run-time while ensuring predictable reconfiguration delays. Finally, this task develops solutions to enable Unified Virtual Memory support for efficient and transparent computational offloading between host and reconfigurable FPGA accelerators. These techniques are also extended for the case of DNN workloads. The target at MS2 is a set of independent mapping and scheduling algorithms, which address computation, communication and reconfiguration in isolation.

Task 4.4 is in charge of developing resilience-aware run-time scheduling techniques and a software infrastructure for mapping the parallel computation. To this aim, this task incorporates the parallel programming model extensions proposed in Task 2.2 and integrates the software resilience analysis mechanisms from Task 3.4. Moreover, this task explores the feasibility of deploying software detection and correction run-time techniques such as automatic techniques for redundant computation, including mapping and scheduling support, and algorithmic-based fault tolerance techniques. The techniques designed in this task is exhaustively evaluated via an extensive fault injection campaigns that considers different error rates. Such campaigns are performed via high-level software-oriented fault injection (i.e. injection of faults into application data).The target at MS2 is a first set of independent services for resilience. The target at MS3 is the complete set of resilience mechanisms, integrated (within Task 6.3) with the mechanisms for predictability, segregation and energy of tasks 4.2, 4.3 and 4.5. This task contributes to D4.2 and D4.3.

To reach the goals of WP4, this deliverable provides the following contents, where we also highlight the related requirements as from the traceability matrix of the amended version of Deliverable 1.1 [1]:

- Chapter 2 describes FRED, the run-time framework to support real-time predictable dynamic partial reconfiguration in FPGAs, contributing to Task 4.3; this is the run-time component responsible for predictable FPGA offloading, needed to address requirement SYS-PCC-REQ-110, and, coupled with AMALTHEA-related off-line tools from WP1, also requirement SYS-PCC-REQ-113; FPGA offloading will contribute to higher determinism in execution and energy efficiency, addressing also requirement SYS-PCC-REQ-101, SYS-PCC-REQ-103 and SYS-PCC-REQ-105;
- Chapter 3 describes extensions for exploitation at run-time of the energy models presented in D3.2, which contribute to Task 4.2, and address requirements SYS-PCC-REQ-105 and SYS-ODAS-REQ-116;
- Chapter 4 describes extensions to the PikeOS real-time hypervisor, relevant to improve task segregation and isolation, and supporting the run-time monitoring of the platform via performance counters, contributing to Task 4.3; the real-time hypervisor is a key component in addressing time predictability requirements SYS-ODAS-REQ-101 to SYS-ODAS-REQ-105, and, coupled with a ROS/ROS2 middleware deployed in the guest partitions, the messaging and component-based requirements from SYS-ODAS-REQ-106, SYS-ODAS-REQ-107 and SYS-ODAS-REQ-114; its capability to isolate a hard RTOS like Erika and a general-purpose or soft real-time OS like Linux addresses SYS-ODAS-REQ-108, SYS-ODAS-REQ-112, SYS-ODAS-REQ-119, SYS-ODAS-REQ-123, SYS-ODAS-REQ-124 and the segregation and safety requirements in SYS-ODAS-REQ-120 and SYS-PCC-REQ-106 to SYS-PCC-REQ-111; the access to performance counters is a basic block needed to address energy-related requirements SYS-PCC-REQ-105 and SYS-ODAS-REQ-116;

- Chapter 5 introduces the runtime mechanisms to support predictable execution of parallel and dependable applications, fulfilling the constraints derived at analysis time, contributing to Tasks 4.3 and 4.4; this addresses requirements related to failure detection and handling SYS-ODAS-REQ-104, SYS-ODAS-REQ-105 and SYS-ODAS-REQ-117, to parallel execution related to SYS-ODAS-REQ-112 and SYS-ODAS-REQ-113; parallel execution plays a key role in AMPERE to control execution latency and address SYS-PCC-REQ-103;
- Chapter 6 illustrates how the above mentioned elements of the run-time architecture will be integrated with the application-level components of the AMPERE target use-cases, with a focus on algorithms for tracking pedestrians in the real-time data fusion and tracking, based on deep-learning, as needed in the railway use-case, contributing to Task 4.3; these components address requirements SYS-ODAS-REQ-110 and SYS-ODAS-REQ-111.

Note that, in line with the DEM nature of this deliverable, a number of the tools presented in this document are showcased through demonstration videos that have been made available at: `https://b2drop.bsc.es/index.php/s/yPGEn2wjGrAddeN` (access is restricted using the password `RRAXqgZ3`).

2 Run-time support for FPGA acceleration

The FRED framework has been selected to contribute to Task 4.3 and to meet the requirements related to FPGA-based acceleration reported in Deliverable D4.1, Section 4.4) FPGA and DPR. FRED was originally presented and theoretically studied by Biondi et al. [2], followed by a preliminary Linux implementation presented by Pagani et al. [3].

In the context of the AMPERE project, a new version of FRED for Linux, denoted as Fred-Linux, has been realized and it will be soon released as open source. It provides the following improvements and new features compared to the preliminary version presented in [3]:

1. While the previous version only supported Xilinx Zynq-7000 platforms, the new version of Fred-Linux introduces the support for the Xilinx Ultrascale+ platform [4], the FPGA-based SoC selected by AMPERE.
2. The central software component of Fred-Linux has been entirely redesigned from scratch in a more modular fashion to better support future extensions of the FRED framework.
3. Fred-Linux introduces new features such as the support for asynchronous acceleration requests and watchdog timers for detecting HW-tasks stalls.
4. On the kernel side, device reconfiguration is now managed by a new low-level driver based on the Linux vendor-independent FPGA manager framework, which has been essential to support the Zynq UltraScale+.
5. On the developer side, the client API has been significantly extended to be more flexible.
6. A Python API has been also introduced to support the popular PYNQ framework [5].

The rest of this chapter is organized as follows. Section 2.1 describes FRED-related hardware design aspects. Section 2.2 focuses on OS-based extensions to support FRED. Finally, Section 2.3 presents the FRED API, used at the application level.

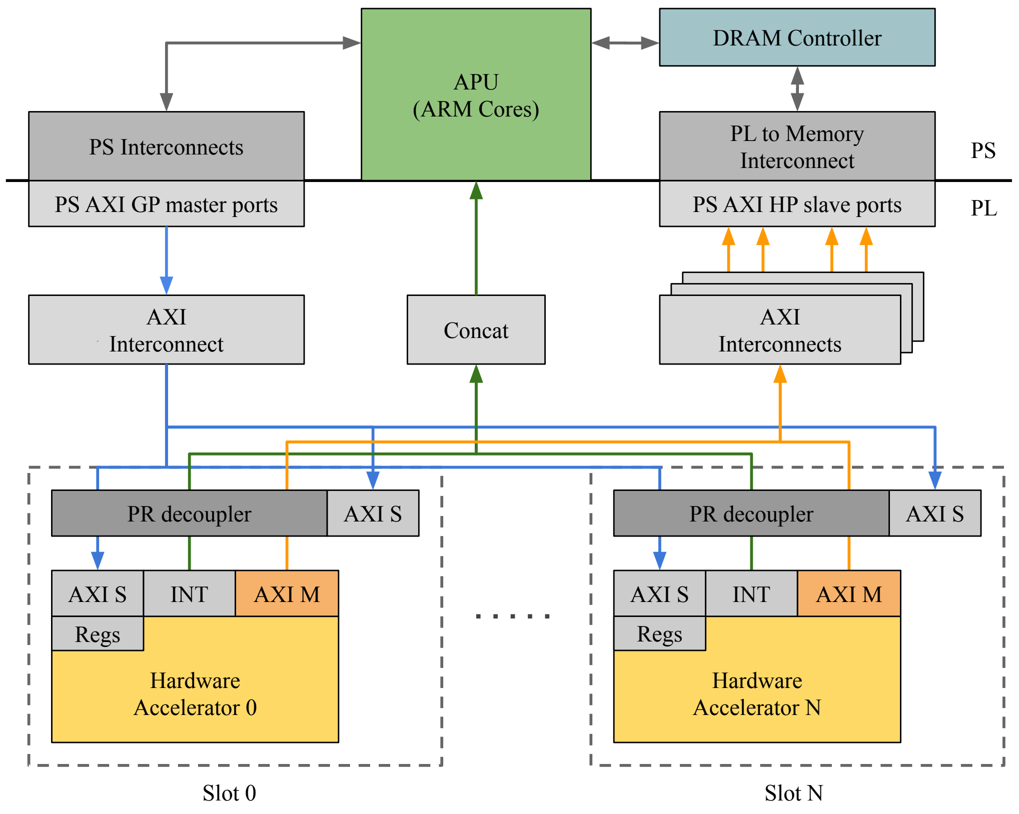
2.1 Platform support

This section describes a system design for supporting the FRED framework on top of the Xilinx Zynq UltraScale+ SoC. The SoC (thoroughly described in D5.1 [6]) includes ARM Cortex-Av8 processors tightly coupled with a reconfigurable FPGA fabric. The internal structure of Zynq Ultrascale+ is divided into two main functional blocks: (i) the *processing system* (PS) block and, (ii) the *programmable logic* (PL) block [7]. The PS block includes the cluster of Cortex-A processors, a set of memory controllers for driving external memories, a small amount of on-chip RAM, and various I/O peripherals. The PS also includes a cluster of Cortex-R processors and two MicroBlaze-based processing units named platform management unit (PMU) and configuration security unit (CSU). The PMU is in charge of platform and power management, while the CSU monitors system integrity and safety. The PL block includes a reconfigurable FPGA fabric consisting of a bidimensional array of programmable logic resources. The specific quantity and type of logic resources depend on the specific generation and SoC model. The units included in the PS side, i.e., ARM cores, memory controllers, and peripherals, are interconnected through the AMBA AXI bus. The same AXI infrastructure is exported to the PL though a set of memory-mapped AXI interfaces exported. These AXI interfaces can be used to extend the system by connecting additional logic modules deployed on the PL.

2.1.1 System support design

The FRED support design provides the foundations for the software support, enabling the interleaving of dynamically-reconfigured HW-tasks on the PL fabric. Figure 2 provides a schematic representation of the design. The PL area is partitioned into two main regions: (i) a *static* region, and (ii) a *reconfigurable* region for hosting hardware accelerators. Following the model introduced in Deliverable D4.1, the static region contains the AXI interconnection infrastructure, namely a set of AXI Interconnects, and may host other support modules in an application-dependent fashion. The reconfigurable region is sub-partitioned into a set of slots that are logically grouped into partitions. Note that Xilinx tools have native support for static partitioning (using Pblocks [8]), which allows constraining the implementation of a logic module to a geometrical region of the FPGA. Therefore, a FRED design flow can be implemented using official Xilinx design tools without relying on third-party experimental solutions, which would instead be required for a slotless approach.

Figure 2: Fred-Linux support design for Xilinx platforms.



According to shared-memory communication paradigm between SW-tasks and HW-tasks discussed in Deliverable D4.1, each HW-task must be able to autonomously access memory regions that are also available to the processors. Xilinx's SoC-FPGAs provide three alternatives for implementing such memory regions: (i) using the internal on-chip memory; (ii) using PL resources to build custom memories on the PL (using BRAM logic blocks); or (iii) using the main (off-chip) DRAM memory. Alternative (i) is not viable since the on-chip memory is limited to 256 KB on the Zynq UltraScale+ [9], and hence may be unsuitable for supporting shared-memory communication for systems with multiple HW-tasks. Alternative (ii) may cause a loss of available FPGA resources since implementing large memory buffers on PL consumes a significant amount of BRAM logic blocks. Conversely, alternative (iii) allows taking advantage of the high-performance AXI ports (HP ports) that grant direct access to the DRAM controller from the PL. Fred-Linux follows the latter approach, implementing the shared-memory communication paradigm using the off-chip DRAM memory, i.e. the orange path in Figure 2.

In FRED, each HW-task has affinity to a partition and can be configured and executed in any slot belonging to that partition. This requirement implies that each slot must be able to host any HW-tasks associated with his partition. Given the technological constraints of Xilinx's SoC-FPGA platforms [8], this requirement can be fulfilled by defining a *common interface* that must implement by each HW-task deployed on the system, which is presented next.

2.1.2 Common interface and hardware design

The realized interface for HW-tasks consists of (i) at least one AXI master interface, (ii) an AXI-Lite slave interface exporting a predefined set of control registers and eight data registers, and (iii) an interrupt signal to send notifications. The AXI master interfaces (denoted as AXI M in Figure 2) allow HW-tasks to access the main memory through the PS DRAM controller, hence implementing the shared-memory paradigm. In this way, HW-tasks can autonomously retrieve the data they need to process without the need for intervention by the processors. It is worth noting that *bus mastering* is also crucial for supporting high-performance hardware accelerators that need to process large amounts of data.

The AXI-Lite slave interface (denoted as AXI S in Figure 2) allows mapping the control and data register of HW-tasks into the address space seen by the processors so that they can be controlled by Fred-Linux. All HW-tasks must implement the same set of control and data registers map to allow the usage of a common software driver. The eight data registers are used to exchange memory pointers whose meaning depends on the specific function implemented by the HW-task. Finally, the interrupt signal (denoted as INT in Figure 2) is meant to be connected to the interrupt controller for notifying the completion of the HW-task to the processors.

Fred-Linux has been designed with high-level synthesis support in mind to simplify the implementation of computationally-intensive functions as FPGA-based hardware accelerators using the popular Vivado HLS tool. The system designer can generate Fred-Linux-compliant HW-tasks by wrapping the C or C++ code of the functions to be accelerated into the common top-level wrapper reported in Listing 1. Vivado HLS will automatically generate the standard interface logic thanks to its interface synthesis capabilities. In addition to HLS, it is also possible to develop HW-tasks directly using hardware description languages such as VHDL or Verilog for achieving higher performances. In this case, a VHDL stub is provided by Fred-Linux, while the Xilinx Vivado suite already provides HDL code stubs for implementing the AXI master and AXI-Lite slave interfaces.

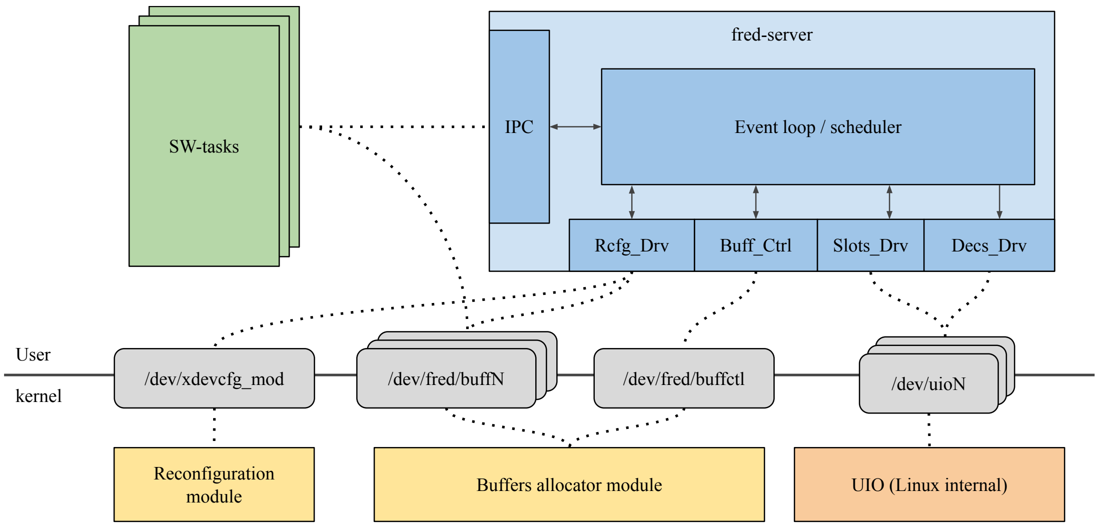
Listing 1: Vivado HLS code for implementing HW-tasks.

```
void slot_i(args_t *id, args_t args[ARGS_SIZE], volatile data_t *mem_in, volatile data_t
    *mem_out)
{
  // AXI Lite control bus
  #pragma HLS INTERFACE s_axilite port=return bundle=ctrl_bus
  #pragma HLS INTERFACE s_axilite port=id bundle=ctrl_bus
  #pragma HLS INTERFACE s_axilite port=args bundle=ctrl_bus

  // AXI Master memory ports
  #pragma HLS INTERFACE m_axi port=mem_in offset=slave bundle=mem_bus
  #pragma HLS INTERFACE s_axilite port=mem_in bundle=ctrl_bus
  #pragma HLS INTERFACE m_axi port=mem_out offset=slave bundle=mem_bus
  #pragma HLS INTERFACE s_axilite port=mem_out bundle=ctrl_bus

  fred_hwacc_body(id, args, mem_in, mem_out);
}
```

2.2 Linux support

This section describes the architecture of the Fred-Linux software support, which is built on top of the hardware design presented in previous section. The software support has been designed in a modular fashion, relying as much as possible on user-space components for improving *maintainability*, *portability*, and *extendability*. Its internal architecture is shown in Figure 3, and further explained in Section 2.2.1. The central component of the software support is a user-space server process, named *fred-server* (blue boxes in the figure), which is in charge of managing acceleration requests from SW-tasks.

Figure 3: Overview of the fred-server.



Periodic SW-tasks (green boxes in the figure) can be implemented as regular Linux processes or threads using the POSIX-compliant SW-task body presented in Listing 2. The SW-task body repeatedly (i) performs computations by invoking one or more HW-tasks, and (ii) makes use of POSIX's `clock_nanosleep()` function for suspending and waiting for the next activation. Since Linux makes use of virtual memory, each SW-task process can access only its own private virtualized address space. On the other hand, HW-tasks directly access the physical address space where the DRAM memory is mapped through the AXI bus. Fred-Linux relies on a zero-copy design, using coherent memory buffers as communication channels between SW-tasks and HW-tasks. Such a zero-copy design allows SW-tasks and HW-tasks to share data without any associated overhead.

Listing 2: Pseudo-code stub for a SW-task.

```
void sw_task_stub(void *args)
{
  struct timespec ts;
  int period_ms = <task_period>;

  /* Get current time */
  clock_gettime(CLOCK_MONOTONIC, &ts);
  /* Set next activation */
  time_add_ms(&ts, period_ms);

  while (1) {
    /*
     * SW-task body:
     * <First software chunk>
     * <Call HW-task>
     * <Second software chunk>
     * <Call HW-task>
     * <Third software chunk>
    */

    /* Sleep until next activation */
    clock_nanosleep(CLOCK_MONOTONIC, TIMER_ABSTIME, &t, NULL);
    /* Set next activation */
    time_add_ms(&t, period_ms);
  }
}
```

2.2.1 User-space components

Fred-server is the central user-space component of Fred-Linux. From an architectural perspective, fred-server is an event-driven application that handles service requests coming from multiple event sources like SW-tasks issuing acceleration requests, HW-tasks notifying their completion, and other hardware events like the completion of the FPGA reconfiguration process. From a functional perspective, fred-server interacts with the rest of the system by means of two main software interfaces, one dedicated to inter-process communications with SW-tasks and the other to communicate with Linux and the kernel support, as illustrated in Figure 3. The communication interface between fred-server and SW-tasks is implemented using UNIX domain sockets. In this way, SW-tasks are entirely decoupled from fred-server.

During the initialization phase, fred-server reads a set of files describing the system layout and the available HW-tasks. Then, according to such a system description, it initializes the support, using the allocator kernel module to instantiate the memory buffers used for both bitstreams and data sharing. After the initialization phase, the server opens a listening socket used by SW-tasks to establish a new connection. Once the connection is established, the SW-task can send requests to the server.

From a client programmer perspective, communication functions between SW-tasks and fred-server are encapsulated into a client support library (see Sec. 2.3) to ease the development process. It is worth noticing that SW-tasks never directly interact with the hardware, nor they are required to perform privileged operations. Indeed, fred-server mediates any interaction between client SW-tasks and the platform hardware.

Fred-server is written in standard C99 using POSIX and Linux APIs. In the context of AMPERE, fred-server has been redesigned in a modular fashion according to the reactor design pattern [10]. The reactor pattern is an event handling pattern used for implementing event-driven applications capable of serializing and dispatching service requests concurrently issued from multiple event sources. In particular, the reactor pattern decouples the responsibility of receiving and demultiplexing events from the responsibility of actually handling events. This characteristic is particularly useful within FRED, since multiple software and hardware event sources like SW-tasks, HW-tasks, FPGA reconfiguration interface, etc. need to be handled in different ways. In this context, the reactor pattern allows implementing each event handler with a different class derived from a common abstract base class. This approach conforms to the *single responsibility* principle [11] since each handler class needs to change only if the handling logic of the corresponding event needs to change. Moreover, it also respects the *open-close* design principle [11] since new classes of events (e.g., handling IPC signals) can be managed by implementing a new handling class without the need of modifying existing handler classes. Please note that, although C99 does not provide native language support for object-oriented programming, there are established techniques for supporting the object-oriented programming paradigm in C99 [12].

2.2.2 Kernel-space components

Fred-server leverages a custom kernel-level support for performing the low-level operations required to control the hardware components of the system support design. It consists of (i) a custom kernel module for allocating the memory buffers employed to share data between SW- and HW-tasks, and (ii) a semi-custom low-level FPGA driver for managing device reconfiguration with the FPGA manager framework. These components are discussed next in details. Fred-server also relies on the UIO framework for managing HW-tasks, i.e., accessing control and data registers, and observing the interrupt lines.

2.2.2.1 *Buffers allocator module*

The shared-memory paradigm introduced in Deliverable D4.1 has been implemented through a set of memory buffers allocated by a custom kernel allocator module. The allocator module uses the Linux DMA layer to allocate physically-contiguous, uncached memory buffers to exchange data between SW-tasks and HW-tasks. When loaded, the allocator module creates a new character device named `fred_buffctl`, which is used by fred-server during the initialization phase for requesting the allocation of memory buffers. Each allocation request is performed using the `ioctl` syscall, passing the required buffer size as an argument. On the kernel side, when the driver receives an allocation request, it creates a new character device named `fred_buffN` (where `N` refers to the buffer identifier that is assigned by the module) and allocates a new contiguous memory buffer, associated with the device, using the `dma_alloc_coherent()` function of the Linux DMA layer. The character device is the means by which the buffer is accessible from user-space.

Once the buffer device has been created, it can be accessed by a SW-task through memory mapping using the Linux `mmap()` syscall. When a SW-task calls (from user-space) the `mmap()` on a buffer character device, the associated memory buffer will be mapped into its virtual address space. Inside the allocator module (on the kernel side), the mapping is performed using the `dma_common_mmap()` function of the Linux DMA layer. Once the buffer is mapped into the SW-task virtual address space, it can be accessed by the SW-task to read and write data without any system overhead. Since the buffer is uncached, no flush and invalidate operations are required on the cache. On the other side, a HW-task can access the same buffer through a physical memory address. The buffers physical addresses are passed to the HW-tasks by fred-server using the set of data registers of their interface (see Section 2.1.2). In this way, SW-tasks and HW-tasks can efficiently share data without any additional copy operation or operating system overhead. It is worth observing that, with this design, the SW-tasks never deal directly with memory management operations.

From the programmer's perspective, the process of mapping these buffers, likewise all other interactions with fred-server, is assisted by the client support library described in Section 2.3. During the system shutdown phase, fred-server releases the buffer devices created during the initialization phase using the `ioctl` syscall on the `fred_buffctl` control device.

2.2.2.2 *FPGA driver*

The FPGA Manager is a component of the Linux's FPGA subsystem that allows reprogramming FPGAs under Linux in a vendor-agnostic fashion [13]. The FPGA Manager has multilayered architecture consisting of an upper layer and a lower layer. The upper layer presents to the programmer a vendor-agnostic API that hides all the platform-specific details. Conversely, the lower layer consists of an FPGA driver, which provides a concrete implementation of the operations invoked by the upper layer for reconfiguring the FPGA. The FPGA driver is in charge of low-level operations such as modifying the bitstream and programming the configuration DMA, which are necessary for reconfiguring the FPGA.

2.3 Client support library API

The client support library provides a lightweight API that can be used by programmers for developing FPGA-accelerated applications with Fred-Linux. Both a C and a Python version of the library are available. For simplicity, only the former is discussed. Note that, since SW-tasks are completely decoupled from fred-server through UNIX domain sockets, additional client APIs for other languages can be easily developed as long as they follow the same communication protocol with fred-server.

Listing 3 reports the functions composing the client support library API. The `fred_init` function initiates the communication with fred-server by initializing an opaque handler of type `struct fred_data` that holds the state of the connection. After the initialization phase, a SW-task can request the association with one or more HW-tasks using the `fred_bind` function. Such a function takes as input the `id` of the HW-task and initializes an opaque handler `fred_hw_task`, which contains a set of references to the data buffers used to share the data between the SW-task (i.e., the current process or thread) and the HW-task. These buffers can be mapped into the address space of the calling SW-task using the `fred_map_buff` function, which takes as input the `fred_hw_task` handle of the HW-task and the index of the buffer, returning a pointer to the mapped buffer. The service functions `fred_get_buffs_count` and `fred_get_buff_size` can be

Listing 3: Client support library API functions.

```
struct fred_data;
struct fred_hw_task;

/*-----------------------------------------------------*/

int fred_init(struct fred_data **self);

int fred_bind(struct fred_data *self, struct fred_hw_task **hw_task, uint32_t hw_task_id)
    ;

int fred_accel(struct fred_data *self, const struct fred_hw_task *hw_task);

void fred_free(struct fred_data *self);

/*-----------------------------------------------------*/

int fred_get_buffs_count(const struct fred_data *self, struct fred_hw_task *hw_task);

ssize_t fred_get_buff_size(const struct fred_data *self, struct fred_hw_task *hw_task,
    int buff_id);

/*-----------------------------------------------------*/

void *fred_map_buff(const struct fred_data *self, struct fred_hw_task *hw_task, int
    buff_id);

void fred_unmap_buff(const struct fred_data *self, struct fred_hw_task *hw_task, int
    buff_id);
```

used to query the number and the size of the buffers used by an HW-task, respectively.

Once the SW-task has completed its initialization phase, binding with its associated HW-tasks and mapping the respective data buffers, it can proceed with its computations. Each computation starts with the SW-task filling the shared buffers with the input data and proceeds by calling one or more HW-tasks using the `fred_accel` function. The `fred_accel` is a blocking function that suspends the SW-tasks until the invoked HW-task completes its execution. After the HW-task completion, the SW-task will resume its execution and can retrieve the data processed by the HW-task by accessing the shared buffers as regular memory. Finally, during the system shutdown phase, the SW-task can unmap all the shared buffers using the `fred_unmap_buff` and close the session with fred-server by calling the `fred_free` function. Listing 4 shows the pseud-code of a SW-task implemented using the C API provided by the client support library. For the sake of clarity, the SW-task only uses a single HW-task the code to handle errors is omitted.

Listing 4: Pseudo-code stub of a SW-task using the C API.

```
void sw_task(void *args)
{
  struct timespec ts;
  int period_ms = <task_period>;

  struct fred_data *fred;
  struct fred_hw_task *hw_task;
  uint32_t hw_task_id = <hw_task_id>;

  void *buff_in = NULL;
  void *buff_out = NULL;

  /* Initialize communication and bind a HW-task */
  fred_init(&fred_data);
  fred_bind(fred_data, &hw_task, hw_task_id);

  /* Map the buffers */
  buff_in = fred_map_buff(fred, hw_task, 0);
  buff_out = fred_map_buff(fred, hw_task, 1);

  /* Get current time */
  clock_gettime(CLOCK_MONOTONIC, &ts);
  /* Set next activation */
  time_add_ms(&ts, period_ms);

  while (1) {
    /* Fill input buffer */
    buff_in[i] = <....>

    /* Call the HW-task */
    fred_accel(fred_data, hw_task);

    /* Read output data buffer */
    <....> = buff_out[i]

    /* Sleep until next activation */
    clock_nanosleep(CLOCK_MONOTONIC, TIMER_ABSTIME, &t, NULL);
    /* Set next activation */
    time_add_ms(&t, period_ms);
  }
}
```

3 Run-time support for energy constraints

Energy-efficiency is a key building block of the AMPERE framework, and this section outlines the developments in the online *actuation* and *monitoring* up until the end of Milestone MS2. The main task that contributes to this part of the deliverable is Task 3.2, in which the sensing and actuation components identified in D3.1 during MS1 are used to provide improved introspection capabilities for the energy usage of the system.

The online monitor described in this document is closely linked to the energy optimization component, part of the offline multi-criteria optimization phase, that is outlined in D3.2. The envisioned interactions with the remainder of the AMPERE framework are shown in Figure 4. The process starts with the Domain-Specific Modeling Language (DSML) (WP1), in which all tasks and their runnables are defined. With each such unit, the energy budget will be defined for that specific task (refer to D1.3, Section 3.2). When the system designer is happy with the modeled system, source code will be automatically generated and passed to the OpenMP Compiler (WP2), which will transform (expand) the code and inserts the OpenMP run-time functions that orchestrate the parallel execution of the system. This produces OpenMP tasks which map to the corresponding runnables in the DSML. In addition to this, the OpenMP compiler generates an abstract representation of the program, referred to as the Task Dependency Graph (TDG), which is further described in Deliverable D2.2. In the next phase of the project, when considering multi-criteria optimizations in a holistic manner, the AMALTHEA SLG will transform the custom property into an OpenMP clause attached to the task; then, the OpenMP compiler will include this information in the TDG. Thus, the TDG will contain the necessary inputs for the offline optimization framework (WP3), further described in Deliverables D1.3 and D3.2.

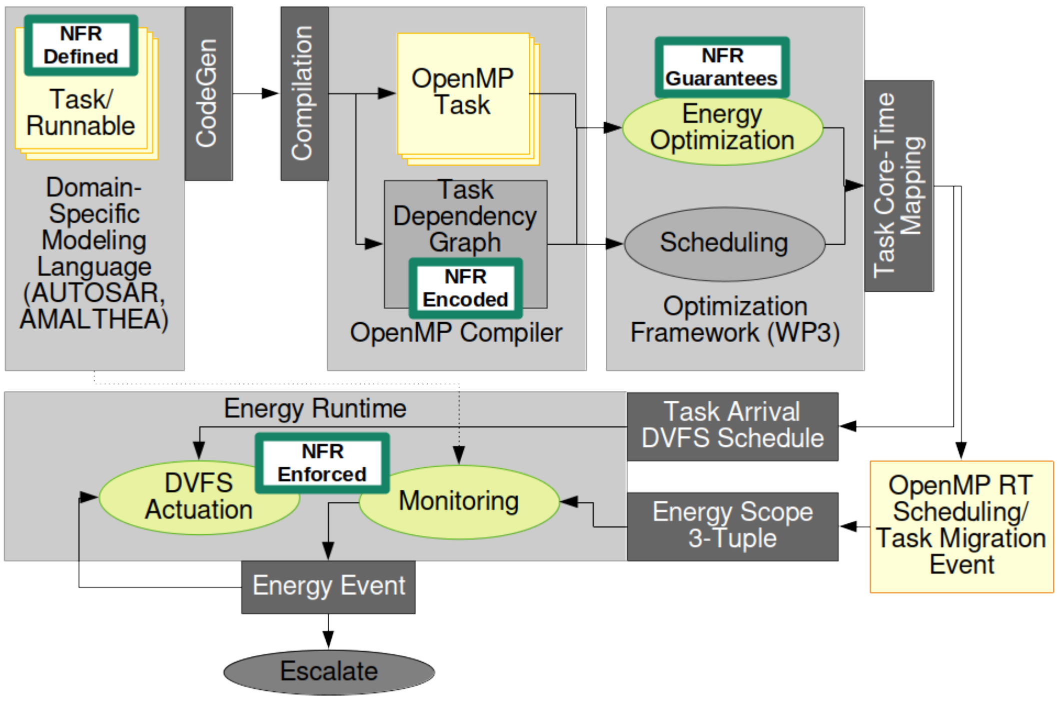
During the offline phase, the energy optimization component (described in Deliverable D3.2) estimates the operating point at which the workload executes most efficiently, giving raise to a task-DVFS operating point mapping, or *Task Arrival DVFS Schedule*. As the project integration kicks off towards Milestone 3, with the goal of turning the single-criterion into a multi-criteria optimization framework, this is envisioned as the DVFS operation point being annotated to each task arrival in the system schedule. In the current single criterion optimization phase (Milestone 2) this simplifies to explicitly configuring the system's DVFS operating point at the launch of the demonstrator. Thus, the scheduling component of the multi-criteria offline optimization stage specifies when each task will be executing on each core. The energy optimization framework has further determined at which DVFS operating point the system should execute once new tasks arrive. The combination of these two optimization outputs from the energy and scheduling optimizers respectively, we collectively refer to as the *task core-time mapping*.

This information is encoded in the run-time orchestration framework, built around OpenMP, which in turn interacts with the Energy Runtime described here. The online energy run-time has two important supervisory roles. First, it is responsible for actuating the DVFS components of the system, by enforcing the generated *task arrival DVFS schedule* when new task arrivals are triggered by the OpenMP run-time. Second, it monitors the energy usage of each task to ensure that the tasks do not violate their energy budgets. Thanks to the offline generated schedule, the tracking of each task comes down to attributing the energy usage of the hardware components to the task that triggers them. Fundamentally, this means that the run-time monitor is informed by the OpenMP run-time environment[1] when a task is executing on a specific core, and for how long (e.g., task started, task completed, task preempted) – such that the energy usage of that core is correctly attributed to that task, and can be compared against its defined energy budget. This ensures that the energy run-time monitor has access to non-functional requirements (NFR) defined in the modeling framework as outlined in the previous Milestone 1, in Deliverable D4.1, and forwarded through the Task Dependency Graph to the offline energy optimizer (D3.2) to be embedded in the configuration of the online monitor. This is illustrated by the forward arrows in Figure 4, showing how the energy-related non-functional requirements provide input at each stage.

The fundamental concept used to achieve this is the *scoped energy usage* defined as a 3-tuple

[1]Should the need arise, the offline optimization phase enables the injection of additional run-time functions that can transfer additional information to the run-time, but currently this is not expected to be required.

Figure 4: The interactions of the Energy Runtime system with other components of the AMPERE framework. Note that the Optimization Framework contains additional components that are not shown, as they will only have minor impact on the Energy Runtime.



$S = (core,\ t_{start},\ t_{end})$. The online monitor uses this to extract an energy estimate $E_{est} = E_{freq}(S)$, accounting for the energy spent by $core$ running at frequency $freq$ during the period $[t_{start},\ t_{end})$, by tracing the activity α from the performance counters. This provides a good mapping to the task granularity provided by the model-based design frontends, e.g., AMALTHEA, used in AMPERE, and their subsequent mappings throughout the meta parallel programming model (Deliverable D2.2), and seamless integration with the OpenMP scheduling points. As such, energy estimates can be collected at (at least) the same granularity as energy budgets are assigned in the modeling framework.

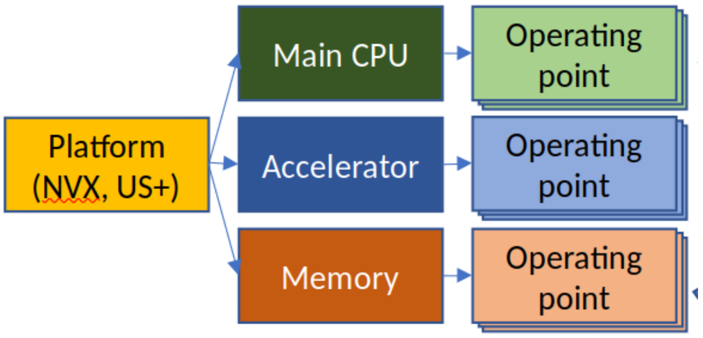
Should the energy usage of a task significantly deviate from the expected, an *energy event* is triggered. At MS2, this leads to a warning message, illustrated by the *escalate* node in Figure 4. It is envisioned that custom handlers can be integrated at this point based on the use-case needs. As part of the next Milestone MS3, multi-criteria optimization will be implemented, at which point the plan is to use additional information from the scheduler to perform slack reclamation: If a task is exceeding (or not exhausting) its energy budget, it may be possible to run the task's cores at a slower speed (scaling down voltage) if there is enough slack until the next task arrival in the system schedule.

The rest of this chapter is organized as follows. Section 3.1 refers to the energy monitoring methods to be used in the project. Section 3.2 presents the actuation methods that adapt the task energy profile. Section 3.3 shows how the proposed system is used.

3.1 Monitoring

The online monitoring capabilities are centered around the linear energy model the draft and intuition for which was presented in Milestone MS1, as part of Deliverable D3.1. This model has now been extended to a *set* (or a table) of multiple linear models $E_{freq}(S)$. Each model in the table estimates the energy usage at one of the supported frequencies $freq$ for a specified *energy scope* S, as outlined earlier in this section. Additionally, one model is created for each subsystem for which the energy is to be estimated. An example is shown in Figure 5. Internally, the model collects a set C of performance counters for each core, approximating the hardware switching activity α, which is used to infer the power P. Energy is the power P used during a specified amount of time t, in this case the $[t_{start}, t_{end})$ interval provided by S. In the following presentation we will therefore show the power estimate P plotted against time t. We begin by evaluating the energy models for the entire CPU complex, and then introduce the $core$ component of the *energy scope*.

Figure 5: The multi-dimensional table containing a set of energy models per operating point (i.e., frequency) for each sub-system to be monitored.



Recalling from D3.1 as part of the previous Milestone, the accuracy of the underlying linear model E_{freq} is sensitive to selecting the most representative performance counters. While the system provides multiple dozen of events that can be counted, only a small amount can be counted at a time (3-6 in the systems selected for AMPERE). Furthermore, which counters that are most representative changes with each frequency, i.e., each sub-model in the model table presented in Figure 5, leading to a large combinatorial space that needs to be explored to train the model. In light of this, performing the by-hand mapping (as in MS1) for each individual frequency is no longer feasible, and the process of generating the model-table has been automated. This is further outlined in Section 3.3. This process does only need to be performed once per supported platform, and as such the final AMPERE framework version will include this information in the *platform characteristics* model. This is in line with the goals set out in D4.1 in the last milestone, where the run-time provides the infrastructure necessary to generate the platform characteristics database, but is not part of a running end-system designed using the AMPERE framework.

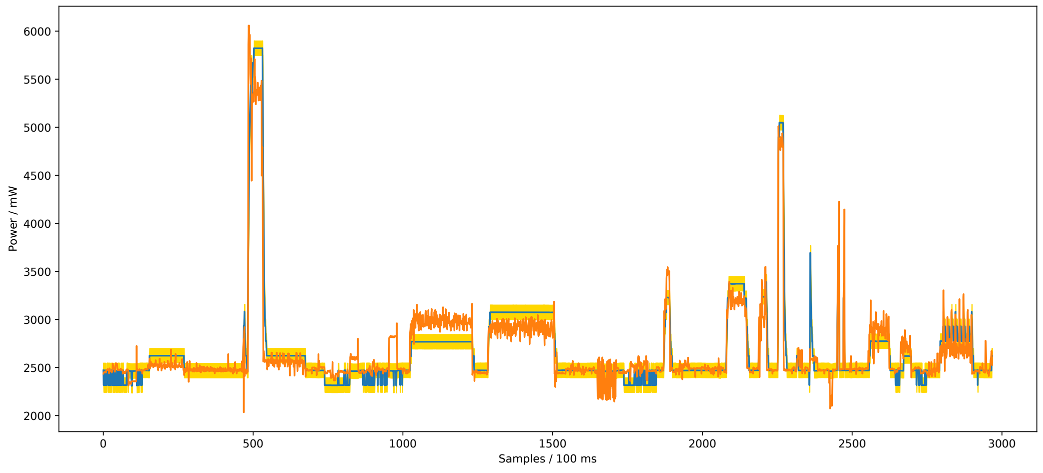
A presentation of the platform characterization for the NVIDIA Xavier, for three different frequencies, is shown in Table 1. These counters were found to best estimate the energy usage of each sub-system using the automatic techniques described in Section 3.3, and delivered as part of the demonstrator. As can be expected, the CPU contribution to the energy usage mainly correlates to the number of active cycles (accounting for power gating), as well as counter `0x24`, which is an undocumented counter specific for NVIDIA SoCs that seems to account for floating point operations. Interestingly, at lower frequencies the energy is best estimated when the instruction cache access counter is included, while higher frequencies only count events related to the processor pipeline itself. The pattern of shifting from memory-centric to compute-centric (or vice versa) counters as the frequency changes can be generally observed for all power domains on the Xavier. Importantly, with the same characterization workload[2] the relevant counters for energy estimation changes with the frequency. This, in combination with the non-linear effects of supply voltage changes for different frequencies[3], motivates the new multi-layer table outlined at the beginning of this section and presented in Figure 5.

[2]The workload used to generate energy and counter traces that are used to train the linear energy model E_{freq}.
[3]Power, and thus energy, scales quadratically with voltage, and higher frequencies require higher voltages. This aspect is not well-captured by linear models.

Domain	700 MHz	1.2 GHz	2.2 GHz
CPU	CYCLES, 0x24, L1I	CYCLES, INST RETIRED, 0x24	CYCLES, 0x24, IRQ
GPU	IRQ, INST RETIRED, L1D	L1I, INST RETIRED, L1D	IRQ, 0x24, LOADS
VDDRQ	0x24, L1D, CYCLES	0x24, CYCLES, INST RETIRED	0x24, CYCLES, MEM ACCESS
CV	L1I, L1 TLB, L1D	L1I, L1 TLB, L1D	L1I, L1 TLB, L1D
SOC	L1D, MEM ACCESS, 0x24	L1D, MEM ACCESS	L1D, MEM ACCESS

Table 1: The performance counters selected for platform characterization of the NVIDIA Xavier at three different frequencies.

Figure 6: Measured versus estimated power consumption of the SoC domain at 2.2 GHz. The measured power is depicted in blue with the measurement uncertainty in yellow, the prediction is given in orange.



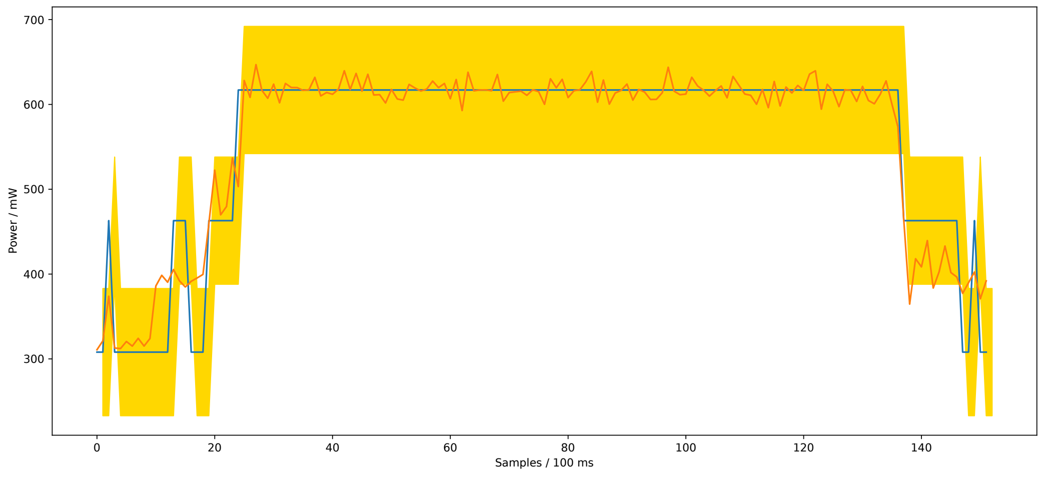
3.1.1 Energy Estimation and Validation

Following the steps described in Section 3.3, the platform characteristics and energy models have been generated and validated for a diverse set of benchmarks from the Rodinia[4] suite. In this experiment, the benchmarks are executed on the CPU of the NVIDIA Xavier with the frequency locked to 2.2GHz. The benchmarks have not been altered, but some benchmarks have non-standard dataset sizes configured, to ensure that they run for a sufficient amount of time to be clearly visible in the plot[5]. The benchmarks are not locked to a specific core, and the threads associated with each benchmark can therefore freely migrate between the eight cores. Figure 6 shows the estimated power numbers in orange, as compared to the values provided by the SoC INA hardware power monitor in blue. The INA has a minimum resolution of $\pm 75mW$, highlighted with the yellow field in the figure. When executing at 2.2GHz, the average error is within 4.26%. The benchmarks with the worst estimation behavior are *srad_v1*, *heartwall_1* and *heartwall_8* with errors of 15.5%, 13.19%, and 10.53% respectively. This occurs because these benchmarks are only showing a low power consumption leading to the quantization noise of the INA and noise caused by OS interference to dominate. Utilizing the multi-frequency support of the model-table, we rerun the experiment at two lower frequencies. The corresponding numbers at 700 MHz (not shown in plot) are 4.1% average error, and 18.67% for the worst behaving one (again, *srad_v1*, a benchmark with a very low power consumption). At 1.2 GHz we see the same behavior, with an average error of 2.63% and a worst-case value of 12.24% for *srad_v2*. This shows that the multi-model table provides an effective way to estimate energy usage at different frequency and voltage levels for AMPERE, and thus effectively enforcing and monitoring the non-functional energy requirements of the system.

The main goal of the AMPERE energy monitoring framework is to improve the introspection capabilities, to

[4]http://www.cs.virginia.edu/rodinia/doku.php
[5]The dataset sizes are provided in `benchmarks.txt` in the demonstrator.

Figure 7: Measured (blue / yellow) and estimated (orange) power consumption of the CPU domain running at 1.2 GHz. The application running on the CPUs is constantly hopping cores every ≈ 8 frames. The tread migration is not visible from observing the (measured / estimated) total power consumption of the SoC.



enable the fine-grained tracking of energy usage at a per-task granularity. This introspection capability is not provided by the built-in INA power sensors, which only provide numbers for coarse-grained power domains, e.g., the entire CPU cluster. To highlight how our model-table and performance counter-based methodology improves upon this, we run an additional evaluation with the goal of tracing the activity of a specific task. Figure 7 shows the CPU domain power over time for *branch_hop*, a custom benchmark that stresses the CPU with branches running on a single core. The benchmark has been configured such that the program migrates between the different cores of the system every 700 ms[6]. We begin by comparing the accuracy of our performance monitor-based approach with the INA. The numbers correspond to the full CPU complex (i.e., not per individual core), as the INA can only provide power numbers for the entire CPU complex. In the figure, we can see how our performance-monitor based approach closely traces the INA-provided values, and staying within the measurement uncertainty (again in yellow) due to the low granularity reported by the INA. If the performance-counter based estimation is more accurately tracking the power usage or if the higher variance is a result of noise can not be determined. However, having determined that our energy monitor is providing the correct total energy for the CPU complex, we move on to inspecting the tracking of the energy usage per-core, such that we can trace the application as it migrates between cores – making it possible to track energy usage at a sufficient level to realize the energy optimization and introspection goals of AMPERE.

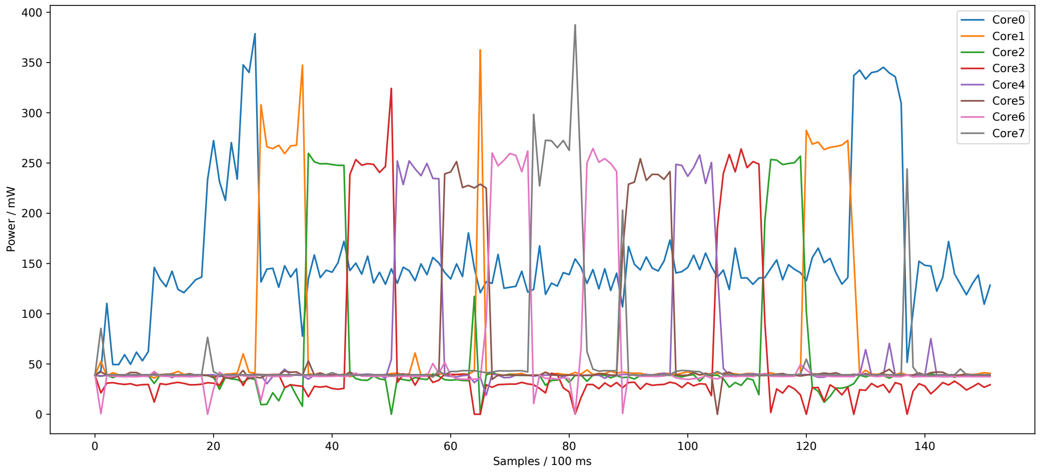
3.1.2 Tracking Granularity

The modeling framework (e.g., AMALTHEA) allow energy requirements to be specified at a per-task (or the even finer *runnable* level) granularity. For this, the INA does no longer provide sufficiently detailed information, as it is impossible to attribute the energy spent to a specific task. For this, energy numbers must be known at a per-core level, such that the energy spent can be mapped to the specific task currently scheduled on that core. The performance-counter based approach provides this level of detail, and is therefore critical to achieve the introspection capatibilities envisioned by AMPERE. The performance counters are implemented at several key positions in the system, for our current purposes importantly the system provides one set of counters for each core, and using them the AMPERE energy monitor can capture the behavior at component and sub-component level.

To demonstrate this, Figure 8 shows the contribution to the overall power consumption from each individual core, running the same experiment as in Figure 7. Here we can clearly see how the energy usage is correctly

[6]This is achieved using the `run_smart_counters.sh` script, provided as part of this deliverable.

Figure 8: Per-core power consumption of the individual CPUs displayed in the same plot, again running the threat hopping application. The task starts being executed on core 0, and is switching to cores 1, 2, 3.... This method allows us to extract the individual power consumption of the cores, allowing us to track the total energy consumption of an application hopping threads.



attributed to the core on which the task is currently running. The sum of the contributions have already been validated to be correct in the previous section, i.e., in Figure 7, but can also be reconstructed from the figure: The core on which the application is running is using approximately $c_{cur} = 250mW$, and an idle core is consuming approximately $c_{idle} = 40mW$. Additionally, core 0 spends an additional $c_{meas} = 100mW$ for continuously writing the results to output[7]. For one active core, seven idle cores, and the measurement overhead we get $P_{total} = c_{cur} + 7 \times c_{idle} + c_{meas} = 250mW + 7 \times 40mW + 100mW = 630mW$, showing again the total power usage validated against the INA in Figure 7.

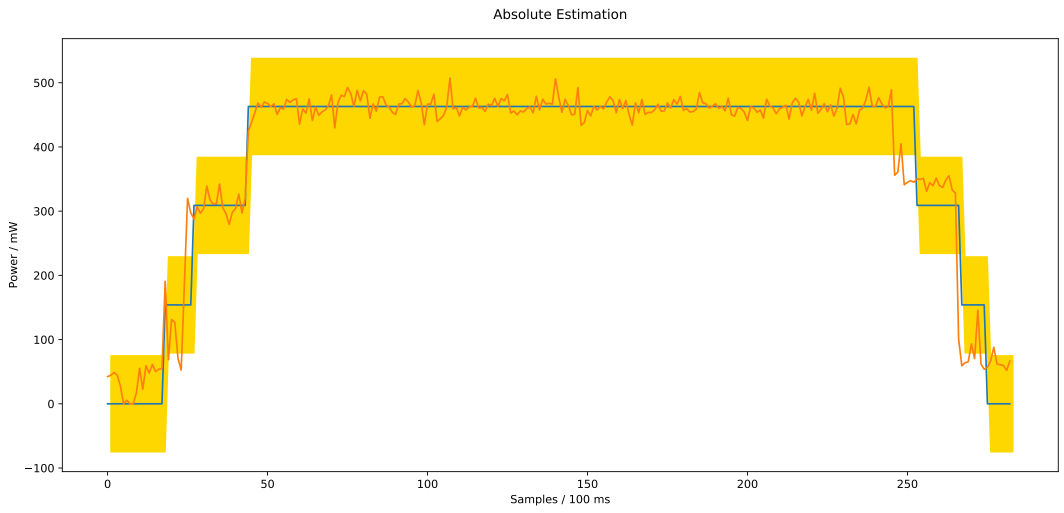
In this experiment, the demonstrator script is enforcing the migration of the task at known intervals, thus emulating the behavior of a fixed schedule determined by the offline optimization phase (which will be the case in future milestones). Under these circumstances, we can demonstrate how the *scoped energy* tuple $S = (core, t_{start}, t_{end})$ provides the means to capture the energy usage of each individual core by attributing the energy of the cores is running at any given moment to its energy footprint, providing the link between workload, system schedule, and the improved energy introspection. Thanks to this, AMPERE provides a way to track the system energy usage at the granularity required to attribute it to the scope at which tasks are defined in the DSML design systems used as frontends in the AMPERE system, fulfilling the requirements outlined as part of the previous Milestone 1.

3.1.3 Memory Subsystem

As presented in Figure 5, the AMPERE run-time tracing framework consists of a model-table, to capture the energy contribution from each system component. By utilizing this approach the demonstrated run-time monitor can also track the energy usage of the memory system, and attribute the activity to the CPU core (and thus task) that causes it. Important to note is that a model for each component can be executed in parallel. Figure 9 shows how the memory system component of the model-table tracks the power usage of the memory system, which we validate against the INA-provided numbers. We run the memory intensive benchmark *memory_rand*, a synthetic application accessing large portions of DRAM in a random fashion. The estimation provided by the model is shown as an orange line, and the INA-reported values as a blue line. The yellow field again represents the error due to the low measurement granularity of the INA. As opposed to Figure 6 which showed the CPU complex's power-domain, this Figure 9 shows the INA for the memory subsystem's power-domain, and the

[7]This overhead is due to the generation of these plots, and is not representative for the power use during standard operation.

Figure 9: The model memory energy estimate (orange) compared to the reference value from the INA (blue) with measurement uncertainty (yellow).



memory sub-model of the model table. The figure shows that the performance-counter based approach can correctly capture also the energy usage of the memory sub-system.

As with the CPU power measurements, the INA does not provide the sufficient level of detail to attribute the energy usage to an individual task. To highlight how the energy monitor included in this deliverable increases these introspection capabilities, we run the memory intensive benchmark, *memory_rand*, and configure it to migrate cores every 750 ms, showing the result in Figure 10. Again, the known execution pattern enforced by the demonstrator mimics a static schedule, and enables the use of the *scoped energy usage* approach for attributing the energy usage to a specific task. The figure clearly shows how the memory system energy usage can be attributed to the core on which the task is executing. Thus, the increased introspection capabilities required for fine-grained energy optimization in the AMPERE system has been achieved for tasks executing on the CPU.

Overall, running the accuracy experiments for the Rodinia benchmarks for the memory subsystem (corresponding to Figure 6), we see an average error of 2.6% at 700 MHz, 2.7% at 1.2 GHz, and 4.72% at 2.27 GHz. This shows that the energy model is able to introspect the energy usage of the memory system to a high degree. For the benchmarks *lavaMD* at 700 MHz and 1.2 GHz, and *particle_filter* at 2.27 GHz we see the worst-case errors, which are significant at 60-173% – however, these benchmarks have a very low level of memory activity, and as such these errors account for small differences in absolute numbers, and lie at the boundary of the INA measurement error (yellow field in previous figures). As such, the average error numbers above provide a more representative measure of the accuracy, showing that the energy estimations is well within the acceptable bounds for the AMPERE framework.

3.1.4 Accelerators

To cover the entire system, portions of tasks that execute on accelerators also need to be accounted for in the energy tracking of tasks. To attribute the energy usage of an accelerator to a specific task, the offloading operation to the accelerator needs to be instrumented. As the OpenMP programming model has been selected for the AMPERE project, this step is significantly simplified, as a unified interface is provided regardless of what accelerator is actually used. This provides a hook into the OpenMP run-time, which provides the necessary information for the Energy Runtime monitor to act on the *OpenMP Runtime Scheduling/Migration Event* described in connection to Figure 4. Thus, similarly as for the mapping of tasks to per-core energy usage by means of *energy scopes*, the offloading operation seamlessly specifies the task-accelerator mapping during the offloading operation in the run-time. Due to the non-preemptive ("run to completion") nature of OpenMP offloading operations this mapping remains valid until the offload finishes.

The estimation of energy through performance counters, as outlined in D4.1 in MS1, comes down to approximating the switching activity in the system through the load of each functional unit. The performance counters provide a measure of this activity, and is also applicable for accelerators. For the GPU accelerators on the NVIDIA Xavier board used in the project, NVIDIA provides an extensive performance counter infrastructure, and their values can be extracted through the NVIDIA CUPTI[8] library, which provides an interface to the GPU counters.

The same approach could be used for offloads to the FPGA, although this introduces an additional level of concern. As FPGA accelerators are defined by custom hardware descriptions (bitstreams) that are instantiated on the programmable logic, there exist no generic interface for tracing events for an arbitrary accelerator. Instead, the bitstream itself must be generated with performance counter support on signals that provide representative abstractions for the functional units underlying the energy estimate. In the AMPERE project FPGA offloading is managed through the FRED framework, developed by partner SSSA, which provides means for accelerator generation, offloading and dynamic reconfiguration of the FPGA. The offloading operation is wrapped by the common OpenMP offloading interface. There are initial discussions between project partners ETHZ and SSSA on the possibility to explore the explicit insertion of performance counters on the interfaces generated by FRED. As work on accelerator support has been proceeding in parallel to this work, this work could not be started within the current Milestone MS2. Instead, any such activities would take place as part of the work for the coming Milestone 3, during which there is additional time assigned to the relevant Task 4.2.

3.2 Actuation

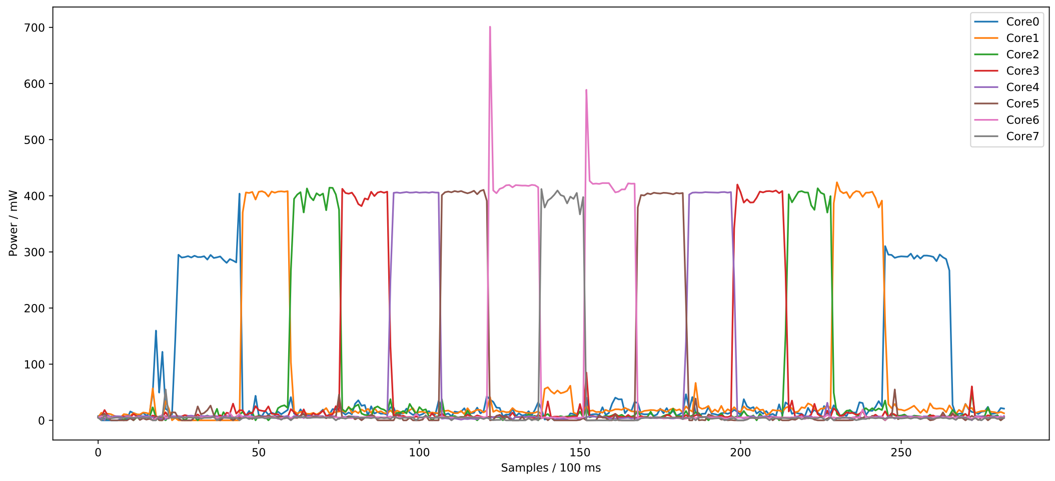
The OpenMP run-time is responsible for the orchestration of AMPERE tasks, and provides a direct mapping to the system as defined in the AMALTHEA model, as outlined in Figure 4. All task events (i.e., arrival, completion, preemption) that are important for the actuation of the energy control hooks, e.g., DVFS, are thus managed by the OpenMP run-time. The definition of the *energy scopes* and *task arrival DVFS schedules* enables the necessary information for the Runtime Energy Actuator to be extracted from the OpenMP run-time in combination with the scheduling information generated as part of the offload Optimization Framework. Thus, the actuation aspect of the current run-time monitor does not include any decisions of its own, but only configures the system in accordance with the *task core-time mapping* determined in the offline optimization phase.

Currently all interaction with the DVFS hardware is handled through Linux' `cpufreq` utility. This provides a common interface between both the supported platforms, and limits the need for reverse engineering of commercial platforms, the internals of which are not well documented. This is a decision that can be changed later, should the need arise, and can be done in collaboration with the operating system as part of Work Package 5.

As part of the next Milestone, in which the current single-criterion optimization (i.e., energy in isolation, scheduling in isolation, etc.) performed so far is extended to the envisioned multi-criteria optimization framework, additional opportunities for intelligent management of DVFS will arise. The main enabler for this is the integration of the energy optimizer with the scheduling framework, which will enable the implementation of slack reclamation to save further energy. With the additional information about the execution time, release time and computational intensity of each task, the slack provided by a task that finishes earlier than expected can be re-assigned to another task, enabling it to run at lower voltage. This offers an opportunity for additional energy efficiency beyond the guarantees provided by the offline optimization phase. Such online optimizations are envisioned to be triggered by the *energy event* backloop outlined in Figure 4. The design of the presented demonstrator thus fulfills the actuation requirements described in Deliverable D4.1 of the previous Milestone

[8]https://docs.nvidia.com/cupti/Cupti/index.html

Figure 10: The model energy estimate for each code for the core-hopping memory-intensive benchmark.



MS1.

3.3 Usage

This section describes how to set up and use the delivered Energy Runtime demonstrator, and how to reproduce the results presented in this report. The main development platform has been the NVIDIA Xavier, and the following instructions apply directly to that system. With minor adjustments, the instructions are also applicable for the other platform selected for AMPERE; the XILINX UltraScale+ ZCU102. In the following instructions *xavier-sh* means that the command should be executed in a user shell on the board, while *xavier-root* means that the command must be executed as root.

The demonstrator has been implemented as a C program without reliance upon third party libraries, which is in accordance with the intuitions presented as part of MS1. Additionally, a number of *bash* and *python* scripts are provided to drive the demonstrator, in absence of a fully integrated AMPERE system, which is the target of future milestones.

3.3.1 Initial Setup

The first step is to clone the code repository to both the board (*xavier*) and the workstation machine (*ws*), followed by the initial setup of the environment.

```
xavier-sh> git clone git@iis-git.ee.ethz.ch:bslk/ampere/arm\_pmu.git
xavier-sh> cd arm_pmu/Xavier
ws-sh> git clone git@iis-git.ee.ethz.ch:bslk/ampere/arm_pmu.git
ws-sh> cd arm_pmu/Xavier
```

After cloning the repository, we have to build the kernel module allowing us to configure and read the arm performance monitoring unit (PMU) from userspace. The current version of the kernel module is derived from an earlier project, and is called `prem_module.ko` for legacy reasons. The module needs to be built once with the following commands:

```
xavier-sh> cd carmel-kernel-module
xavier-sh> make
```

The kernel module has to be loaded on every reboot of the system. Its successful load can be checked with dmesg. For now, the kernel module is not automatically loaded on boot, requiring a manual load after each reboot.

```
xavier-sh> sudo su
xavier-root> insmod prem_module.ko
xavier-root> exit
```

Once the initial setup has been completed, the board (in this case the Xavier) is configured using the following commands. Please note that these steps need to be repeated after every reboot, as the Xavier boots into some default configuration that does not allow us to use the full frequency capabilities the SoC has to offer.

```
xavier-sh> cd arm_pmu/Xavier
xavier-sh> sudo su
xavier-root> /usr/sbin/nvpmodel -m 0
xavier-root> /home/nvidia/jetson_clocks.sh
xavier-root> echo userspace > /sys/devices/system/cpu/cpufreq/policy0/scaling_governor
xavier-root> echo 2265600 > /sys/devices/system/cpu/cpufreq/policy0/scaling_max_freq
xavier-root> echo 0 > /sys/devices/system/cpu/cpufreq/policy0/scaling_min_freq
xavier-root> echo 2265600 > /sys/devices/system/cpu/cpufreq/policy0/scaling_setspeed
xavier-root> /home/nvidia/jetson_clocks.sh --show
xavier-root> exit
```

Setting the power mode with nvpmodel to 0 disables all power caps and the jetsonclocks.sh script sets the clock frequencies to its maximum value. The four writes to *sysfs* serve multiple purposes, they activate the userspace scheduler, set the available frequencies from 115MHz to 2.2GHz and set the current frequency of all cores to 2.2GHz. Custom frequency numbers can be substituted in their place. We use the userspace governor to influence the frequency of the system during the profiling and the run-time phases.

3.3.2 Create Correlation Database

As described in deliverable D3.1, section 3.2 *On the importance of representative counters*, we need to identify the counters with a high correlation to the energy usage. To achieve this, we create a full profile of all counters, running all benchmarks at all the considered frequencies as part of the platform description. This is a very lengthy process, which can take multiple days to complete. This full profile run is done once with a representative set of benchmarks, and the results can be reused for all future experiments. To run a full trace, we have to create a new folder on Xavier and run a bash script. This script runs all benchmarks at all selected frequencies for all counters found in the system. Before starting a benchmark, it starts and stops data logging to a binary file. This script is planned to be replaced by *Extrae*. The next steps describe how we can archive the database and use scp to copy the data to the local workstation for further processing.

```
xavier-sh> mkdir -p traces
xavier-sh> sudo ./run_all_counters.sh
xavier-sh> mv traces corr_db
xavier-sh> tar -cJvf corr_db.tar.xz corr_db/
xavier-sh> rm -rf corr_db
iis_ws-sh> scp user@xavier_host:/path_to_repo/arm_pmu/Xavier/corr_db.tar.xz \
        /scratch.../...
iis_ws-sh> cd /scratch.../...
iis_ws-sh> tar -Jvf corr_db.tar.xz
```

3.3.3 Evaluate Correlation

Once a full activity and power database has been created, we use a python script to analyze the correlation between the counters and the power consumption for each power domain and frequency. We assume a linear dependency between the activity of a given counter and the power usage. To quantify the relation we calculate the Pearson correlation coefficient using *SciPy*. The counters with the highest absolute value of r^2 are selected. This script uses a full profile run (all selected frequencies, all counters) in a binary file format. The location of the database is handed to the script. Its output will be a graphical table of the counters that should be selected going forwards. Additionally a *json* file is created, which holds the correlation information in a machine-readable format for usage in the further steps.

```
iis_ws-sh> cd arm_pmu/Xavier/scripts
iis_ws-sh> python3.7 do_correlation.py\
        /scratch2/tbenz/projects/ampere/ampere_traces/full_trace/
```

3.3.4 Run A Profile Run

The profile run used to create the correlation information as part of the platform description is meant to be static, e.g. done once for each platform with a well-defined set of reference benchmarks. Currently, the script will read the *json* file produced in section 3.3.3 - once integrated, this information is stored in the Amalthea model. For the actual usecase of the AMPERE run-time, smaller profiling runs need to be started featuring the usecase benchmarks. These profiling runs only contain relevant data by tracking the activity of the N highest correlating counters.

The *run_smart_counters.sh* script needs to be adapted with the information given in the *json* file produced in section 3.3.3. Currently, this is a manual process. The script is run and produces again a binary trace database for a given set of frequencies and a set of benchmarks. It will now only profile the important counters found in section 3.3.3. Again, this script is planned to be replaced by *Extrae*. The benchmarks are defined in the `benchmarks.txt` file, which contains comma separated values. The first field is the name of the benchmark, and the second is the command to execute. These are extracted from the Rodinia benchmark suite for the experiments above. The next steps describe the the process of copying the data on the workstation for further processing.

```
xavier-sh> cd arm_pmu/Xavier
xavier-sh> replace counters=... in run_smart_counters.sh
        with the profiling string from counters.json
xavier-sh> mkdir -p traces
xavier-sh> sudo ./run_smart_counters.sh
xavier-sh> mv traces profile
xavier-sh> tar -cJvf profile.tar.xz profile/
xavier-sh> rm -rf profile
iis_ws-sh> scp user@xavier_host:/path_to_repo/arm_pmu/Xavier/profile.tar.xz \
        /scratch.../...
iis_ws-sh> cd /scratch.../...
iis_ws-sh> tar -Jvf profile.tar.xz
```

3.3.5 Train And Evaluate Model

Once we have an activity trace with power data, we can train our linear model presented in deliverable D3.1, section 3.1 *Model Rationale*. We use a python script to read-in the best correlating counters (from the above mentioned *json* file) for each frequency and power domain and the activity and power trace from the previous profiling step. The location of the *json* file and the profiling database from section 3.3.4 need to be updated in the script manually. A unified interface for all the presented scripts and applications is currently under development, but not yet finished.

We use linear regression implemented by *SciPy* to train our linear model for each frequency and each power domain. The weights generated in the training process are then written to a set of C header files, representing the trained model. These header files will be used in the next step to compile the online monitor.

The header files contain the weights of the linear model used to estimate the power from the counter activity in the online phase. As of now, one header per frequency and domain is created. This information is part of the platform description.

The script also calculates the error between the estimated and measured power consumption and creates a series of plots showing how well the model tracks the power consumption of the system. The results presented in this chapter were created using the `train_model_db_pc.py`, which generates and (if enabled) plots the power series of the model's estimate and the reference measurements from the INA (including measurement error, if enabled). The steps can be used to reproduce the non-core-hopping results.

```
iis_ws-sh> cd arm_pmu/Xavier/scripts
iis_ws-sh> adapt read_data('...') in train_model_db_pc.py
iis_ws-sh> python3.7 train_model_db_pc.py
```

3.3.6 Compile And Run Online Monitor

Once the model is trained and exported in the form of a set of header files, we can compile the online monitor binary. The online monitor then is launched taking 4 arguments: the frequency the cores are currently running in *kHz*, the frame number the energy needs to start being accounted for, the end frame, and the core number.

Once the online monitor is run, it stays idle until the start frame is reached, it monitors the energy of a task running on a given core until the end frame is reached. The monitor will now display the energy consumed by the task in all power domain and exit.

```
iis_ws-sh> cd arm_pmu/Xavier/online_monitor_cli
iis_ws-sh> ./compile.sh
iis_ws-sh> ./online 2265600 0 2 1
```

As outlined in section 3.1.1, we can achieve a higher degree of introspection by using the online monitor by defining an *energy scope* S. In the AMPERE framework, we will be handed a task dependency graph (TDG), describing the scheduling and the binding of all the tasks. We can use the information from the TDG for a given task, e.g. cores it is using when, to create a sequence of calls to the online monitor. A task for example may use core zero from frame 0 to frame 10 and then switch to core 3 until it terminates at frame 15. The calls to the online monitor derived from this pattern would be:

```
iis_ws-sh> ./online 2265600 0 10 0
iis_ws-sh> ./online 2265600 11 15 3
```

To get the total energy consumed by this task, we can simply add the energy contributions estimated by the two calls of the online monitor.

To monitor a binary, simply start the online monitor and run tasks on the system, and it will produce the current energy usage on the standard output. The online monitor can be used to reproduce the core-hopping experiments presented earlier in this section. Simply launch the online monitor and run the provided `*_hop.sh` scripts, which execute either the compute- or memory-bound benchmark with different core affinities to generated the presented results.

4 PikeOS Hypervisor

The PikeOS [14] hypervisor is designed to be modular and to manage virtualization along with the special requirements of complex embedded systems. Among other needs, it addresses real-time responsiveness, deterministic, and diverse hardware and software support. While server or desktop virtualization mainly targets space partitioning to make better use of an x86 hardware platform, PikeOS embedded virtualization offers more flexibility through time and space partitioning. PikeOS embedded virtualization provides partitions for multiple guest operating systems and supports diverse hardware platforms.

4.1 Support for Xilinx ZYNQ UltraScale+ ZCU102 board

In the context of Work Package 5, the project's platform selection process has been successfully finished by listing the Xilinx ZCU102 board with a Zynq UltraScale+ MPSoC [4] and NVIDIA Jetson AGX Xavier Developer Kit [15]. For additional information regarding a selection of reference hardware platforms, please see a deliverable D5.1 Reference parallel heterogeneous hardware selection [16].

During the first months of the project, SYSGO has ported PikeOS on the ZCU102 board. Currently, the PikeOS Architecture Support Package (ASP) is available on ARMv8 architecture and the Board Support Package (BSP) supports the following drivers:

- Serial
- Network
- Mass storage
- CAN
- Watchdog
- GPIO.

4.2 Support for OpenMP

The OpenMP 4.5 [17] support has been implemented for PikeOS 5.0 on the ARMv8 architecture. In particular, the library `libgomp`, which is part of the GNU GCC 7.5.0 compiler suite, has been ported to support the PikeOS Native API. This library from the GNU Compiler Collection enables high-level parallel programming through OpenMP's directive-based syntax instead of explicit calls to the low-level PikeOS API. However, due to the nature of the PikeOS real-time capabilities, the OpenMP implementation comes with the following limitations:

- **No Automatic Load Balancing.** Threads in the PikeOS Native application do not automatically migrate from one CPU core to another to perform some automatic load balancing. The same limitation will exist with the threads created by OpenMP run-time support code.
- **No Dynamic Adjustment of the Number of Available Threads.** Calling the omp_set_dynamic() with "true" argument does not affect since PikeOS implementation does not support dynamic adjustment of the number of available threads.
- **PikeOS Thread Scheduling and Busy Waiting.** PikeOS threads with the same priority are executed with the FIFO order, therefore busy waiting in threads might not work in all cases, for example, if a thread preparing a result is not scheduled.
- **Configuration.** Abstract names "cores" and "sockets" are not supported in the environment variable `OMP_PLACES`.
- **No Thread Deletion Support.** Cannot delete thread before the thread finishes a task.

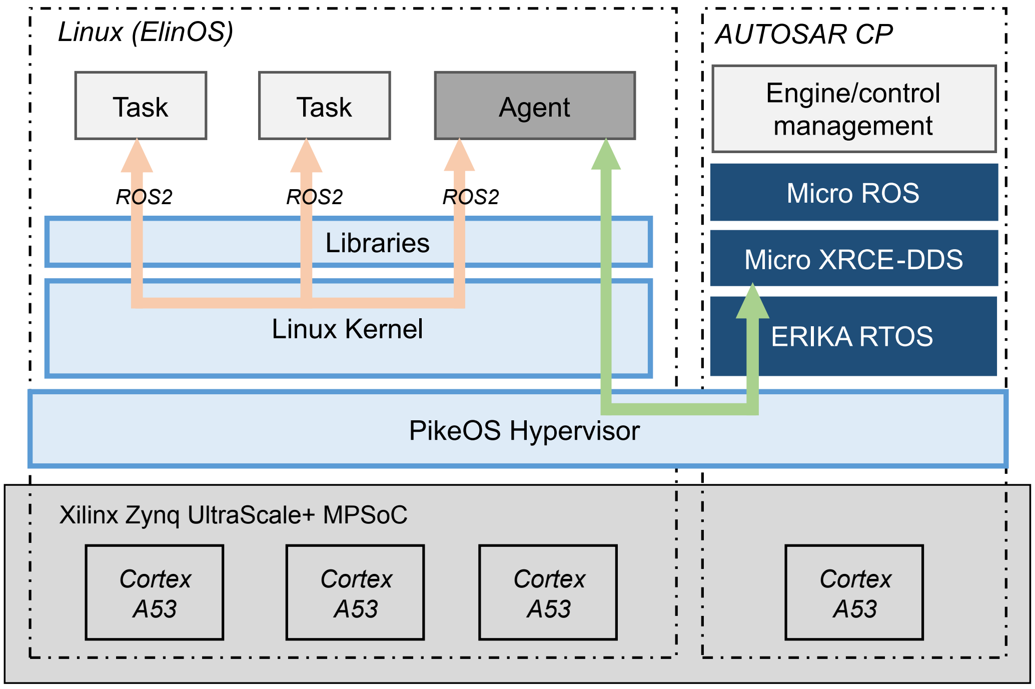
These limitations will be better analyzed with the other AMPERE partners during the second half of the project.

4.3 Support for the ERIKA RTOS

In the scope of Work Package 5, SYSGO and Evidence have collaborated for porting Evidence's ERIKA Enterprise RTOS [18], compliant to the AUTOSAR Classic specifications [19] on top of PikeOS hypervisor. Now the PikeOS hypervisor officially supports ERIKA RTOS. This is another step in integrating embedded systems in a cost-saving and functionally safe manner. This aims to serve the interest of the automotive industry which is focused on the possibility of executing non-critical tasks (e.g. infotainment, navigation, logging, human-machine interfaces) alongside safety-critical tasks (e.g. engine/brake control). The platform must also be capable of executing the HPC activities needed by the forthcoming assisted/autonomous driving functionalities.

As shown in Figure 11, the envisioned software architecture will consist of a general-purpose (or soft real-time) OS, like Linux, and a safety-critical / hard real-time OS, like ERIKA Enterprise, concurrently executing on top of the PikeOS hypervisor on a Xilinx ZCU102 embedded platform. The use of PikeOS allows for partitioning the physical resources available on the board, like CPU cores, across the general-purpose and the safety-critical domains, so as to minimize any possible contention in the access to physical resources. However, possible interferences among different partitions can be monitored and controlled through specific mechanisms as detailed in the next section.

Figure 11: Multi-domain software architecture.



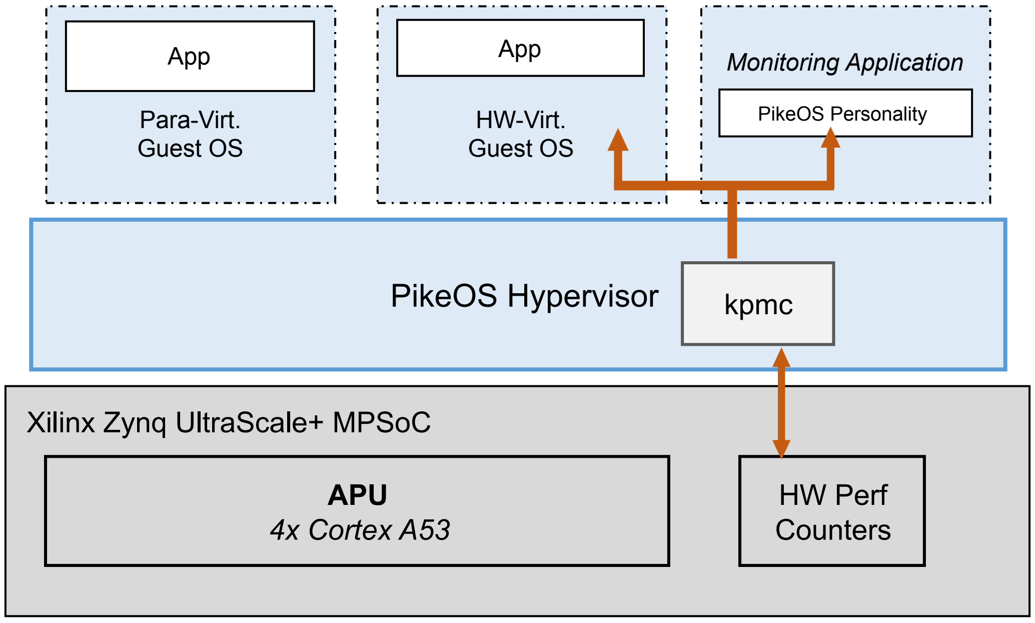
4.4 Measurement of the Contention For Shared Resources

Multi-core processors impose a risk on safety-critical applications as processor cores are not completely decoupled from each other. Especially, when non-critical applications are running in parallel with the critical applications. This is because safety-relevant systems depend on the determinism of the execution of algorithms. With PikeOS, fixed timing slices can be set which are met precisely. To assure that the algorithm finishes in its allocated time frame, many tools for static and dynamic code analysis are used.

Even for a single core, such analysis is complex due to many considerations including interrupts, scheduling behavior, priority settings and memory controllers. When it comes to multi-core, the situation gets more complicated, as those processors have been designed primarily for average-case timing rather than the worst-case situations. For multi-core, a contention between multiple cores, that raises because of shared resources (e.g. caches, buses, and main memory), is the biggest problem. For the application level, conflicting access to shared resources can be avoided at an architectural level, mainly through efficient scheduling settings combined with core allocation.

In the context of Work Package 4, the PikeOS makes use of the hardware performance counters to measure the contention on shared platform resources and for the online energy estimation by ETHZ. As shown in Figure 12, the PikeOS architecture foresees `kpmc`, a kernel driver which configures the performance counters and provides services (APIs) for the user-space to use performance counters. This kernel driver will be developed by SYSGO in the scope of Task 4.3.

Figure 12: PikeOS kernel driver for hardware performance counter.



5 Runtime support for predictable execution of parallel and reliable applications

This Section summarizes the work done withing WP4 to fulfill MS2 of Task 4.3.

5.1 Runtime performance analysis for scheduling policies adaptation

Deliverable D3.2 [20] describes the approach for timing analysis targeting resource allocation and timing constraints, which focuses on offline analysis based on performance traces obtained from program executions, as described in deliverable D1.3 [21]. The offline analysis results in an annotated model that contains performance data, potentially from executions in different platforms. Moving the approach to run-time analysis requires important changes. When dealing with offline analysis, more time is available to make decisions but it is based on profiling and it might not perfectly match the actual execution. On the other hand, the online analysis has access to real-time information, however faster (and possibly not optimal) decisions must be performed. Taking this into account, we adapt our approach to include both performance trace extraction and analysis to be performed at run-time.

Our run-time program adaptation approach is still under development and will be focused on providing OpenMP schedulability decisions based on execution time and memory management. For instance, we intend to analyze and discover tasks heavily burdened with memory contention and schedule them not to execute concurrently with other tasks with the same issue.

While Extrae [22] is used for the offline time analysis, for the run-time analysis we are developing a low overhead approach based on perf tools [23]. With perf tools, we are able to access performance counters information, but this time we are accessing that information at run-time. The main difference between the two approaches, despite the many features of Extrae, is that Extrae is a more high-level API that focuses on providing easy to access data regardless the overhead imposed. Also, Extrae actually uses perf in the background to access several performance counters. When the run-time overhead is an issue, a more low-level and faster approach should be used. Instrumenting the application with direct calls to perf tools has less run-time overhead in the application performance.

We are currently in the development of an approach for automatic code instrumentation using perf. The approach focuses on instrumenting code parcels of the OpenMP tasks, named *task parts*, that monitor the beginning and end of these task parts and provide, at run-time, information similar to that accessible by Extrae. This will allow the analysis to be performed at run-time.

5.2 Run-time support for execution of parallel components

Deliverable D2.2 [24] provides a preliminary version of the meta parallel programming abstraction derived from the code synthesis tool and the compiler techniques developed within WP2 targeting (at MS2) performance as single-criterion optimization. More specifically, the deliverable describes the mechanisms implemented in the AMALTHEA synthetic load generator (SLG) to automatically generate OpenMP task-based code out of an AMALTHEA model, exploiting the data dependencies expressed in the runnables of each Amalthea task. In order to address the holistic approach for multi-criteria analysis foreseen for MS3, a set of mechanisms at compile-time and run-time have to be provided to allow predictable execution of the OpenMP framework.

Previous works already tackle the analysis of the OpenMP framework from three different angles:

1. The OpenMP *specification* has been analyzed in terms of functional safety [25], concluding some restrictions are needed, but minor changes would allow later compiler and runtime techniques to ensure the correctness of the system. Furthermore, the specification has been analyzed in terms of time-predictability [26, 27, 28], concluding the model as time-analyzable and proposing augmentations to support predictable scheduling techniques.
2. Several *compiler* analysis techniques have been presented towards correctness [29, 30] and programmability [31, 32, 33], enabling a most robust and safe system.
3. At the *runtime* level, there is an effort towards the analysis of current implementations towards a possible certification, particularly in the context of the RTEMS RTOS [34]. Furthermore, some works propose the use of static analysis techniques to generate a TDG statically in order to reduce the overall memory consumption [35] and avoid the use of dynamic memory [36], enhancing the predictability of the overall system.

We draw from the works using a TDG computed statically at compile time, and enhance this mechanism to allow a novel *define-once-run-repeatedly* execution model that computes the TDG dynamically and reuses it for enhanced performance and predictability, and lower usage of the memory system needed to orchestrate the parallel execution. There are hence two versions of the same TDG: one computed at compile-time (when possible, i.e., control-structures surrounding tasks and task dependencies can be resolved at compile-time), addressed in Section 5.2.1, and the other computed at runtime (when it cannot be computed at compile-time), addressed in Section 5.2.2.

5.2.1 Static generation of the TDG

The generation of a static TDG is implemented in the Mercurium [37] source-to-source compiler. It is based on a series of classic compiler analysis, including parallel control flow graph (PCFG), use-definition chains, induction variables, and optimizations, including constant propagation and loop unrolling, adapted to OpenMP. The different techniques allow the compiler to first expand all the control flow structures involved in the instantiation of OpenMP tasks, and then resolve the predicates associated to the edges of the graph corresponding to task synchronizations. As an illustration, Figure 13a shows a code snippet implementing a series of tasks generated within two nested loops, and Figure 13b shows the TDG generated by Mercurium where dotted edges correspond to transitive dependencies, and solid edges correspond to true dependencies (transitive dependencies are computed at compile-time but they are not propagated to the runtime system because they are redundant).

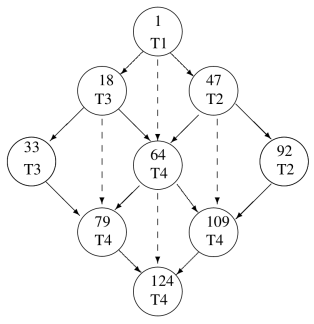
After the generation of the static TDG, the compiler generates a sparse matrix holding the information of the TDG to be used later, at run-time, to orchestrate the execution. Figure 13c shows the TDG structure generated for the TDG in Figure 13b. There, each entry contains a unique task identifier, and stores in separate arrays the tasks it depends on (input dependencies), and the tasks depending on it (output dependencies). Moreover, the sparse matrix is sorted using the identifier, so a dichotomic search can be applied at run-time for matching dependencies with cost $\mathcal{O}(n \log n)$, where n is the number of entries of the matrix.

This allows two optimizations of the execution targeting the predictability of the system, described as follows:

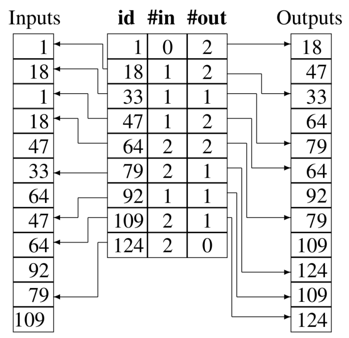
1. *Preallocation of task data-structures.* This technique addresses the static allocation of the data-structures needed by the runtime to handle OpenMP tasks. The methodology is to first determine the *maximum degree of parallelism* exposed in the application by approximating the maximum width of the TDG; then, this information is used to allocate the task data-structures that are required at run-time to never force a ready task to wait for a task data-structure to be freed by a previous task. The compiler can allocate the data in two manners: (a) in the data segment (of C programs), meaning that data is defined and initialized statically; or (b) in the heap, meaning that the compiler introduces the calls to dynamically preallocate the data in the initialization phase of the application. Both options push the allocation of memory before the execution of the parallel kernel. This makes the execution of the system more predictable, because even if the system has insufficient memory at runtime, the error will occur in the

```
int A[N][M][BS_N][BS_M];
#pragma omp parallel \
      num_threads(4) shared(A)
#pragma omp single
  for (int i = 1 ; i < N ; i ++)
    for (int j = 1 ; j < M; j ++)
      #pragma omp task
            firstprivate (i,j) \
            shared(A) \
            depend(in:A[i][j-1]) \
            depend(out:A[i][j])
      compute_block(A[i][j], A[i][j
          -1]);
```

(a) OpenMP program example



(b) TDG



(c) Runtime structure

Figure 13: TDG static generation withing the Mercurium compiler.

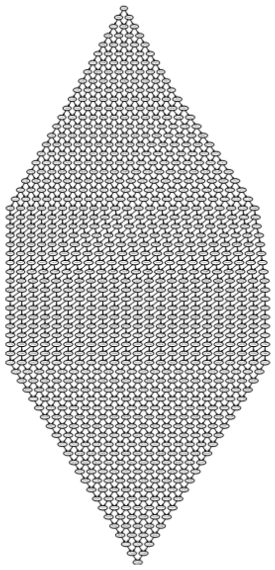
initialization phase, so it can be handled in a similar way to a Power-On Self-Test failure; furthermore, the amount of memory needed by the OpenMP runtime is also bounded at compile-time. In this sense, another measurement to consider in the allocation of task data-structures is the amount of available cores (or hardware threads), as the system will be able to execute only that amount of tasks at the same time.

2. *Lazy task creation.* Opposite to regular OpenMP runtimes, which allocate tasks as soon as they are encountered, this mechanism allows to allocate tasks only when they become ready for execution. The maximum number of tasks that can be ready at some point of the execution, i.e., the maximum degree of parallelism, can be determined at compile-time by approximating the maximum width of the TDG. Then, at run-time, if this computation was too optimistic, new structures can be created, ensuring the functionality is not affected by the parallelism.

These features have been implemented in the Mercurium compiler and the OpenMP runtime library from GCC, named *libgomp* [38]. Figure 14 presents the TDGs of two applications, the one in Figure 14a corresponds to a pedestrian detector based on a histogram of oriented gradients (HoG) that exposes a sweep dependency parallel pattern, and the one in Figure 14b shoes an image sampling application for infra-red H2RG detector (ESA). Later, Figure 15 shows the evaluation of the memory space (in KBs) that different runtime configurations require to allocate the task data-structures for the two applications. The graphics show the use of static memory (STATIC memory), which is valid for any configuration because it only respects to the static allocation of the TDG, and the use of dynamic memory of three configurations: (1) the compiler preallocates all tasks (Prealloc full TDG), (2) it preallocates the TDG's maximum parallelism (Prealloc max parallelism), and (3) it preallocates a number of tasks equals to the number of cores (Prealloc num cores) (here, the number of available task structures is sufficient to execute in parallel all ready tasks. The original libgomp implementation uses an amount of memory equivalent to our framework when the full TDG is preallocated.

As the figures reveal, the version that preallocates all tasks, although bounds the memory usage to the number of tasks, results in a very intensive use of memory. Instead, the version that preallocates the maximum amount of parallelism in the TDG or the number of available cores results in a huge fall in the consumption of dynamic memory. Finally, the use of static memory increases with the number of tasks, as the structures to manage the TDG are statically stored by the compiler.

Furthermore, Figure 16 shows the execution traces obtained with Extrae [39] and plotted with Paraver [40] for the HoG application represented in the TDG in Figure 14a. The x-axis represents time, and the y-axis represents threads (only one in this case). Particularly, Figure 16a shows the execution using the original implementation of libgomp, where the allocation of memory overlaps with the parallel execution, and Figure 16b shows the use of memory when preallocating task data-structures, in which case the allocation is pushed to the initialization phase, and then the parallel execution starts. In each figure, the trace on top, *Dynamic memory call*, represents


(a) Pedestrian detector based on a histogram of oriented gradients (HoG)

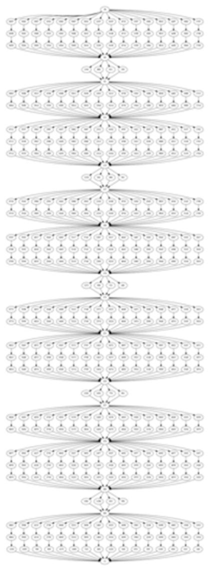
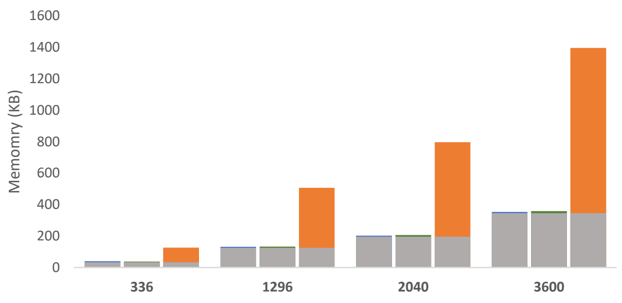
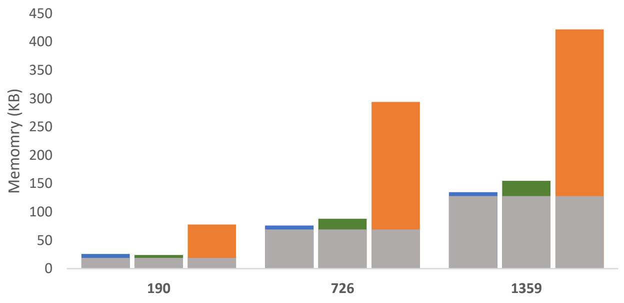

(b) Image sampling application for infra-red H2RG detector (ESA)

Figure 14: TDGs of two OpenMP task-based applications



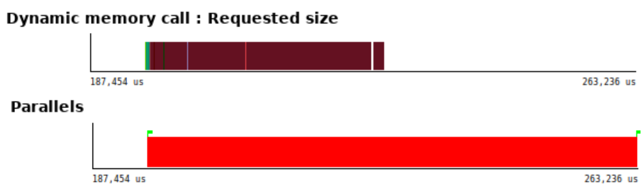
(a) HoG



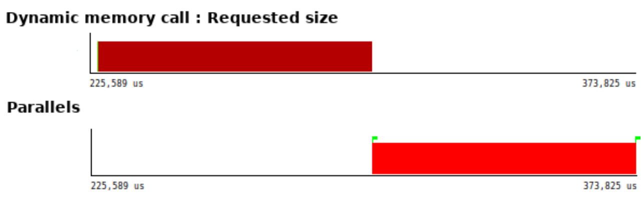
(b) ESA

Figure 15: Evaluation of the memory consumption using preallocation and lazy task creation.

calls to malloc and calloc, and different colors represent different allocation sizes; and the trace below, *Parallel*, represents a portion of the parallel execution, particularly its beginning.



(a) Original libgomp



(b) Task data preallocation

Figure 16: Execution traces differentiating memory allocation and computation.

At this phase of the project, each non-functional constraint is considered separately. Nonetheless, Task 4.3 includes the evaluation of the runtime mechanisms for predictable execution regarding their impact on performance, to avoid losing productivity due to timing or safety constraints. Figure 17 shows the speedup of the *HoG* and the *ESA* applications in two different architectures: an Intel Xeon Platinum 8160 [41] from the Marenostrum IV supercomputer [42], and a Texas Instruments Keystone II [43]. The former features a 24-core

Intel Xeon Platinum, and the latter a quad-core ARM Cortex-A15 host and a 8-core DSP fabric acceleration device. In the TI Keystone II, only the DSP is considered in the execution, and the OpenMP accelerator model is used to offload computation from ARM cores to DSPs. The results show how the speedup obtained with the preallocation methods is the same as with the original OpenMP runtime, hence no performance loss occurs.

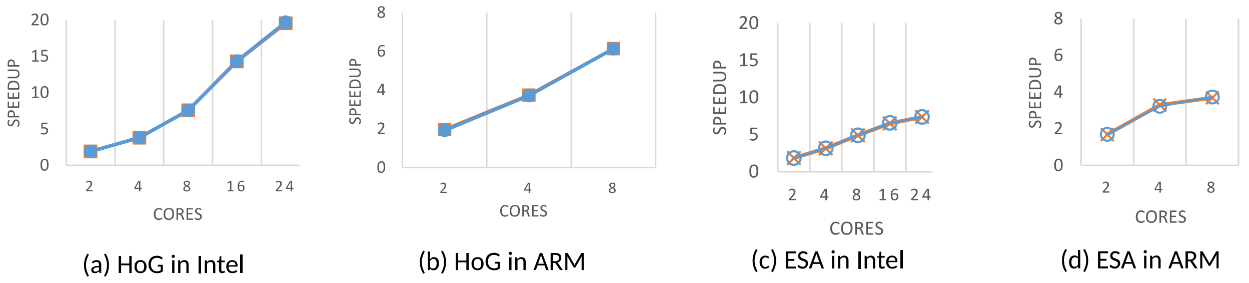


Figure 17: Performance speedup of the preallocation mechanism (blue) vs. vanilla libgomp (orange).

5.2.2 Dynamic generation of the TDG

When the information required to derive the TDG is not available at compile-time, there is still a chance to benefit from the mechanisms implemented in the runtime to exploit the static TDG. This occurs when a given TDG is executed several times, meaning that the shape of the TDG does not change and the data used by the tasks always points to the same memory locations. When this occurs, the TDG can be dynamically captured by recording it the first time it is executed, and re-executing the recorded TDG the subsequent times (in an define-one-run-repeatedly execution model, equivalent to that defined by *CUDA graphs* [44] to exploit NVIDIA GPUs).

Figure 18 shows the preliminary evaluations of the dynamic recording of the TDG on a 48-core Intel Xeon Platinum with two different benchmarks (i.e., gauss-seidel algorithm and N-body simulator) when executing the TDG of each of the applications 128 times. The speedup of the dynamic recording is compared with that obtained with the original libgomp. In both cases, the dynamic recording enhances the performance of the system, obtaining an almost perfect speedup in the case of Nbody with 48 cores, because it reduces the contention on the shared resources. Further analysis of this feature will be performed on the use-cases being developed within AMPERE.

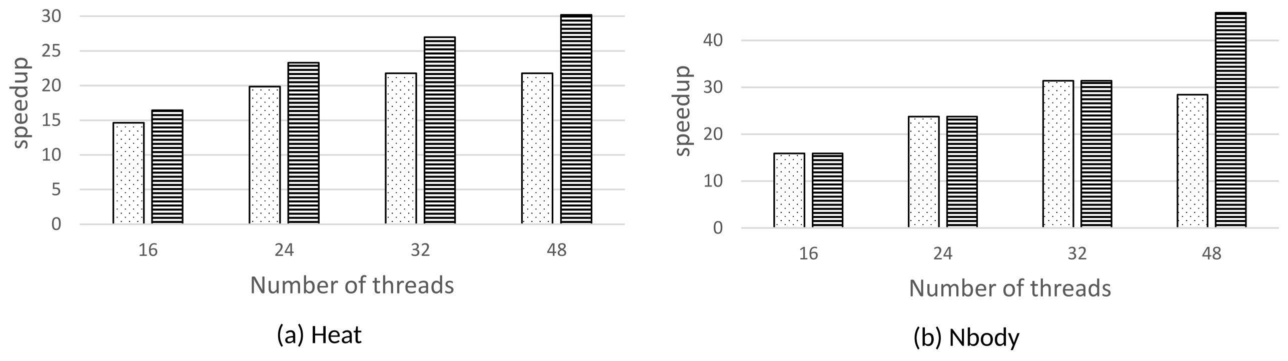


Figure 18: Performance speedup of the dynamic TDG recording. dotted represent the original libgomp and striped represent the dynamic recording approach.

5.3 Real-Time scheduling at run-time of parallel applications

Once deployed on the AMPERE platform, parallel applications will be able to run taking advantage of the parallelism and heterogeneity available on the underlying hardware, respecting the non-functional constraints attached to the applications at design-time. Specifically, for what concerns timing constraints, a key role will be played at run-time by the real-time scheduling algorithm used for scheduling the parallel computational activities running on the platform, over the available cores. Real-time scheduling algorithms are able to ensure a timely execution of applications, provided that the overall system respects the configurations and constraints imposed by the multi-criteria optimization and analysis phase, as being designed in WP3 [20].

It is foreseen that real-time applications, ultimately composed of a number of real-time threads interacting in a direct acyclic graph (DAG) data-flow topology, will be able to be scheduled either by using fixed-priority scheduling, or dynamic priority disciplines such as based on earliest deadline first (EDF). These disciplines are available already both on the Erika hard real-time OS, and on Linux. Precisely, on Linux the SCHED_DEADLINE scheduler [45] can be used as an EDF-based policy. In AMPERE, this policy is being modified to match better the needs of real-time applications being deployed on heterogeneous multi-cores, in an energy-aware fashion, and in presence of dependencies and acceleration of parts of the DAG-like computations [46].

A preliminary part of these modifications has been published as the Adaptively Partitioned EDF scheduler (apEDF) [47]. However, the implemented patch-set will need a set of improvements in the second half of the project, in order to integrate energy awareness, for example as shown in [48], and be ported to a version of the Linux kernel that allows for its integration within the AMPERE target platforms. More details on this part of the AMPERE run-time will follow in deliverables specific to WP5.

5.4 Resilience-aware run-time

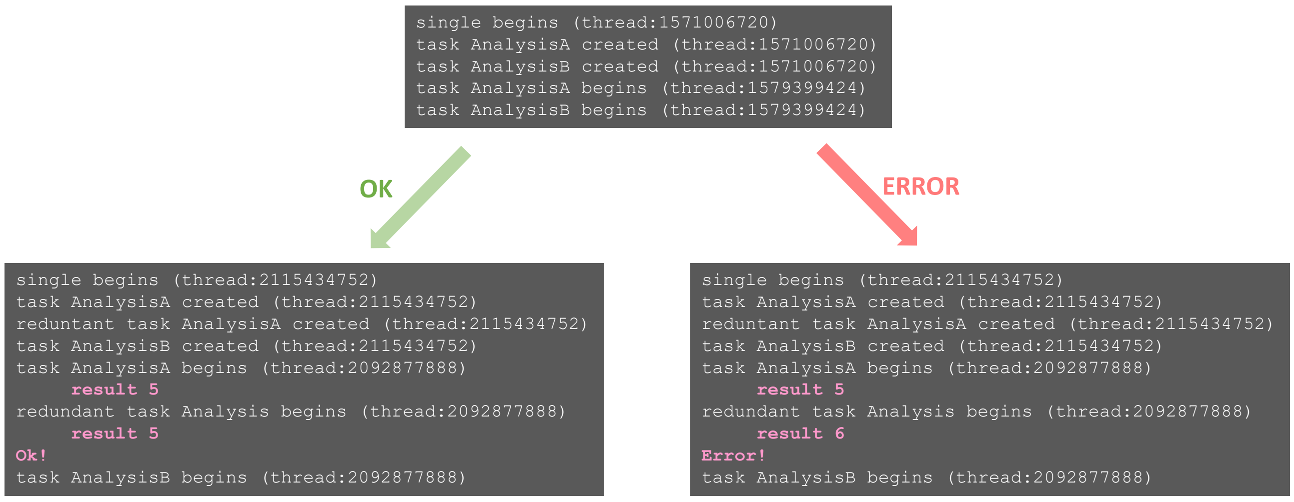
Task 4.4 incorporates (1) the extensions proposed in Task 2.2, Meta parallel programming abstraction and parallel programming model extensions, and (2) the resiliency analysis mechanisms from Task 3.4, Resilient software techniques. On one hand, Deliverable D2.2 [24], First release of the meta parallel programming abstraction and the single-criterion performance-aware component, presents the augmentations proposed in the Amalthea model in order to address resiliency. These augmentations considers a new *redundancy* attribute that can be attached to a runnable in order to indicate how many replicas of that particular runnable (which will be transformed in an OpenMP task) have to be generated. On the other hand, Deliverable D3.2 [20], Single-criterion energy optimisation framework, predictable execution models and software resilient techniques, explains how these annotations are taken into account in the Amalthea Synthetic Code Generator in order to generate redundant tasks. Deliverable D3.2 also features the Observer-based resilience approach proposed by THALIT/UNISI, which detects in runtime values outside the expected bounds.

The current implementation of redundant tasks only reports whether the execution is correct (based on the comparison of the results obtained with the different redundant tasks). Figure 19 shows the results provided by the GCC libgomp OpenMP runtime system supporting redundant tasks. The square on top corresponds to the execution of a simple code with two tasks. The square on the left, corresponds to an execution that replicates the task $AnalysisA$ and the results of the original and the redundant task are the same. Finally, the square on the right, corresponds to an execution with replica where the results are not the same and an error is warned to the user.

This preliminary resilience approaches presented in D3.2 shall be enhanced, at least, in the following directions:

1. The execution of the original task and the replicas shall be synchronous, meaning that anything depending on the original task cannot initiate its execution before all replicas have finished and the results have been compared.
2. The recovery mechanism is yet to be defined.
3. The tasks to be replicated are currently annotated by the designer. Automatic mechanisms to derive which tasks need replication are under study, considering, for example, the ASIL level of the functionality.

Figure 19



```
single begins (thread:1571006720)
task AnalysisA created (thread:1571006720)
task AnalysisB created (thread:1571006720)
task AnalysisA begins (thread:1579399424)
task AnalysisB begins (thread:1579399424)
```

```
single begins (thread:2115434752)
task AnalysisA created (thread:2115434752)
reduntant task AnalysisA created (thread:2115434752)
task AnalysisB created (thread:2115434752)
task AnalysisA begins (thread:2092877888)
     result 5
redundant task Analysis begins (thread:2092877888)
     result 5
Ok!
task AnalysisB begins (thread:2092877888)
```

```
single begins (thread:2115434752)
task AnalysisA created (thread:2115434752)
reduntant task AnalysisA created (thread:2115434752)
task AnalysisB created (thread:2115434752)
task AnalysisA begins (thread:2092877888)
     result 5
redundant task Analysis begins (thread:2092877888)
     result 6
Error!
task AnalysisB begins (thread:2092877888)
```

6 GPU-accelerated DNNs, data fusion and tracking

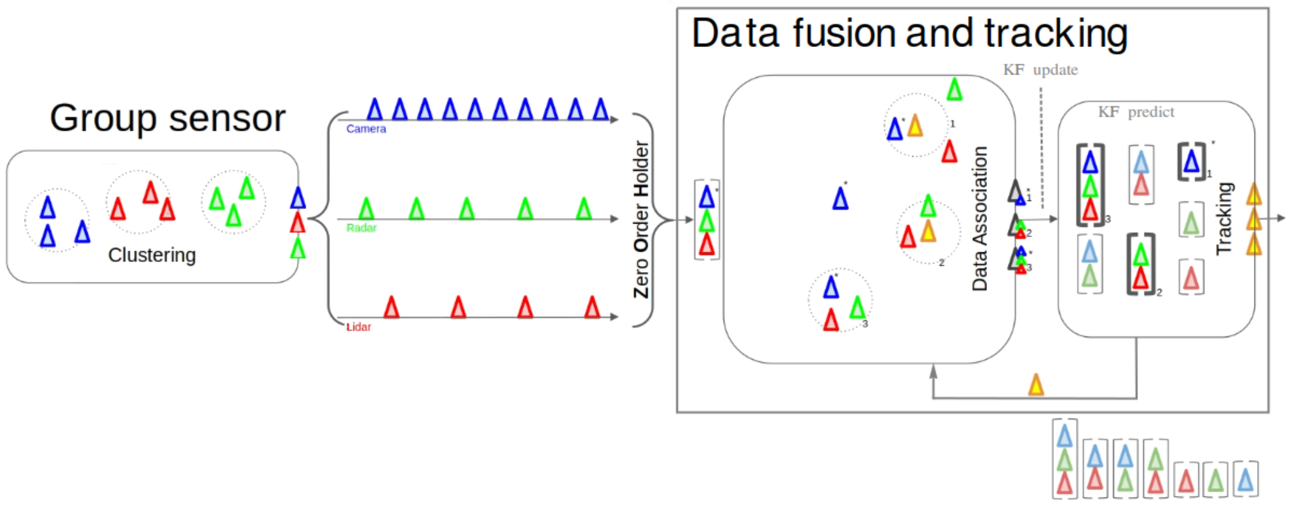
This section aims at showing the connection and usefulness of the elements of the run-time architecture as detailed above, with parts of the AMPERE use-cases. As an example, here we focus on the railway use-case, focusing on a time-critical aspect of its computations, namely real-time object detection and tracking via GPU-accelerated deep neural networks (DNNs) and data fusion among multiple types of sensorial data.

The THALIT use-case [1] within the AMPERE project is mainly focused on a railway automated driving system, which will be able to perform autonomous driving activities in all road and environmental conditions manageable by a human counterpart. In order to obtain this level of autonomy, a huge effort is required in various research fields, including multi-object detection and real-time tracking. In fact, understanding the dynamic properties of multiple entities in the environment is critical, as it directly affects the quality of the location of detected objects, their tracking and decision planning at higher levels. This use-case concerns the Florence tramway system, and it will take advantage of a set of sensors installed onboard the tram vehicle. In particular, the sensors are represented by optical camera, radar and LiDAR, which are in charge of collecting a large amount of raw data which needs to be analyzed in real-time by the data association and tracking algorithms currently under development. In order to be useful, this analysis has to be performed within tight bounds, as detailed in [1], thus computations will need GPU acceleration.

THALIT and UNISI are working in tight conjunction for developing a multi-object tracking method based on two macro-aspects: tracking and data association, which consist of prediction, measurement, association, and update steps. Predictions are usually made using a Kalman Filter (KF) algorithm, processing the positions of the current set of objects of interest. These predictions are then matched to the actual measurements. Many data association algorithms have been proposed to robustly link predictions and measurements. In fact, at this stage, ambiguities may arise: predictions may not be supported by measurements (objects occlusion), there may be unexpected or missing measurements (variance in number of objects), more than one measurement may correspond to a predicted feature or a single measurement may match to more than one feature. These ambiguities generate uncertainty and make the data association process a challenging problem.

Fig. 20 shows the entire process, from the sensor data acquisition to the data fusion and tracking algorithm.

Figure 20: Data association model of the data fusion and tracking algorithm.



The measurements from the different sensors are transmitted in real-time using the ROS[1] communication system, with each sensor transmitting on its ROS topic a message of a customized type based on the specific information provided by that sensor. The information provided by each sensor refers to the Vehicle Body Frame

[1]Robot Operating System (ROS): www.ros.org

(VBF) that is an XYZ reference system with metric coordinates centered in the front of the train and oriented along the motion direction of the train. As can be seen, there are three input traces, represented by triangles of different colors, related to the different sensors with their different operating frequency. Each triangle in Fig. 20 does not represent a single measurement but, instead, represents a set of objects: a vector of points in case of the radar and LiDAR sensors or a vector of detections for the camera data. The indication "Zero Order Holder" is used to represent how the measurements arriving from different sensors are received and synchronized. In fact, measurements from multiple sensors must be merged according to some scheduling. On the one hand, if the measurements are all aligned (or have a negligible misalignment) and arrive at the same time instant, it is possible to combine all the measurements into a single vector to be sent to the data fusion and tracking algorithm. On the other hand, if the measurements are not synchronized, then data must be used as soon as it arrives and we need to proceed with the data association separately for each sensorial data as soon as it is received.

The data provided by the camera is analyzed, frame by frame, using the Yolo [49] neural network (GPU-accelerated), which provides as output the coordinates (x, y, height, width) of the bounding boxes related to the detected objects, together with the relative labels and the estimated likelihood values. About the radar and LiDAR data, instead, these sensors provide the point clouds relating to the objects detected at any time instant. These information shall be processed to extract useful data for the tracking stage. The single points are composed by the x,y coordinates of the object plus an Intensity value which represents the Radar Cross Section (RCS) for the radar's points and the belonging layer for the LiDAR's ones. These additional information, will be used by the data association algorithm to improve the accuracy of the data binding. In the data association box, the yellow triangles represent the forecasts obtained from the tracking module while the circles represent the gated regions around each target. This is important, since the data association algorithm needs to access both the data from the tracking module and the measurements arriving from sensors in order to properly make the association between measurements and targets. In fact, the position estimations obtained by the tracking module, in addition to being sent as output, are backwards propagated towards the data association module. The associated data are then sent as a new input to the tracking module, which is in charge of providing a new prediction, based on the association performed using the Kalman filter algorithm.

The whole data fusion and tracking module is managed by a software developed by UNISI and THALIT which will take care of both the object position estimation and their tracking. In order to do this, some commonly used algorithm in the literature have been taken as a reference, namely the Global Nearest Neighbor (GNN), the Joint Probabilistic Data Association (JPDA) and the Multi-Hypothesis Tracking (MHT). In addition to the algorithms for the data association procedure, further attention is reserved to the heuristics related to the management of the create and delete process of the individual traces that are identified during the execution of the algorithms. This is an orthogonal process with respect to the association algorithm used, but the criteria associated with its management play a crucial role in the correct development of the data association process. About the tracking module, its function is carried out by a Kalman filter integrated into the data association and tracking algorithm. As with the data association algorithm, also for the tracking module the different possibilities are currently under investigation in order to select the more appropriate solution for the THALIT use-case.

In conclusion, regarding the data association and tracking part, some components will be deployed on the platform at run time. First of all, a ROS module will be running to provide data in real time from the sensors. These data are then transmitted to the data association and tracking algorithm which proceeds with the association between the existing traces and the detected targets. For each track created, which will monitor the trajectory of a specific target, a Kalman filter is initialized and will remain active for the entire presence of the object on the scene. The type of Kalman filter to be used is currently under investigation, given that the use of an Unscented Kalman Filter (UKF) instead of a linear Kalman filter on the one hand could bring a benefit in terms of precision but, on the other hand, it would require a major effort in terms of computational cost, which has to face the in-place timing and energy-efficiency constraints. This could be crucial in terms of performance as the number of KFs/UKFs that have to be run in parallel depends on the number of objects simultaneously present on the scene.

As evident, the final processing pipeline in the use-case can be viewed as a direct acyclic graph (DAG) of computations, that need to be performed under tight timing and reliability requirements, deployed as a set of components interacting via middleware and component frameworks such as provided by ROS, and with acceleration of part of the computations using multi-core processing and GPUs. Interestingly, the need for supporting ROS-based (or ROS2-based) publish-subscribe communications is shared with the BOSCH use-case, for which there is already a plan to support DSML-specific modeling abstractions to represent accurately this kind of interactions (as described in D1.3 [21], Section 7). Therefore, this use-case will greatly benefit from the integration with the above discussed elements of the AMPERE run-time architecture, and particularly the support for predictable execution of real-time software applications (as analyzed during the off-line analysis), in presence of GPU-acceleration of compute-intensive deep neural network (DNN) tasks, as will be supported by AMPERE on the NVIDIA AGX platform (one of the two reference platforms chosen in D5.1 Reference parallel heterogeneous hardware selection [16]).

7 Conclusions

This document presented the contributions for WP4 related to independent run-time energy support for predictability, segregation and resilience mechanisms proposed in the AMPERE project. These contributions can be classified in:

1. Run-time for predictable parallel heterogeneous computing (Chapters 2, 6, 4, and 5), contributing to Task 4.3;
2. Run-time for energy optimization (Chapter 3), contributing to Task 4.2;
3. Run-time for resilience support (Section 5.4), contributing to Task 4.4.

All three tasks of D4.2 continue until the milestone MS3, to be presented in D4.3. Thus, all methods presented in this document are expected to compose an *integrated* run-time support for timing predictability, energy, segregation and resilience. More specifically, the following actions will be developed for D4.3.

Methods to improve timing predictability, related to Task 3.4 (in D3.2) and Task 4.3, are the core NFR in terms of run-time optimization. There are several challenges related to the integration required for D4.3. First, the timing analysis is a fundamental step done offline, in design time, and the analysis results change according to the underlying platform and the workload being executed. The supporting data required for the run-time system (presented in WP2, meta model for parallel programming) must be generic enough to accommodate different platforms and workloads. Second, moving offline timing predictability methods for run-time require fundamental changes. For instance, online methods need to have minimal performance impact (i.e., be lightweight) but, at the same time, the optimization method needs to be fast enough to decide a course of action. It means that methods designed in Task 3.4 require a non-obvious adaptation for the run-time environment.

The methods proposed for energy management, related to Tasks 3.2 (energy optimisation strategies) and 4.2, need to be integrated. This integration is not only in terms of energy NFR (because complementary energy-saving researches were developed in different institutions such as ETHZ, ISEP, and SSSA), but also integration with the other NFRs supported by the AMPERE project. For instance, once additional criteria are introduced for optimization during run-time, the DVFS operation point with the minimum energy usage may not be the best trade-off with other optimization criteria. Thus, the AMPERE project will extend the Energy Runtime system presented in Chapter 3 by proposing an online slack reclamation mechanism to improve the energy efficiency (using for instance DVFS, as mentioned before) based on runtime data collected by the energy monitors. The Energy Runtime system will also allow interaction with other run-time systems responsible for optimizing other NFRs. Finally, there is energy-related integration also in WP5, about operating systems and parallel platforms. For instance, the types of energy-related sensors/actuators available in different platforms can vary drastically, imposing one additional challenge of having a uniform Energy Runtime system across the two platforms supported by the AMPERE project. Similarly, the resilient software techniques presented in D3.2 proposed by BSC and THALIT, related to Tasks 3.4 (resilient software techniques) and 4.4, also need to be integrated with the other NFRs for their use in run-time.

List of Acronyms

ASP	PikeOS Architecture Support Package
ATF	ARM Trusted Firmware
BSP	Board Support Package
CPU	Central Processing Unit
CSU	Configuration Security Unit
D	Deliverable
DAG	Direct Acyclic Graph
DPR	Dynamic Partial Reconfiguration
DSML	Domain Specific Modeling Language
DVFS	Dynamic Voltage and Frequency Scaling
EDF	Earliest Deadline First
EEMI	Embedded Energy Management Interface
FPGA	Field-Programmable Gate Array
GNN	Global Nearest Neighbour
GPU	Graphics Processing Unit
HPC	High-Performance Computing
JPDA	Joint Probabilistic Data Association
KF	Kalman Filter
MDE	Model-Driven Engineering
MHT	Multi-Hypothesis Tracking
MPSoC	Multi-Processor System on Chip
MS	Milestone
NFR	Non-Functional Requirement
ODAS	Obstacle Detection Avoidance System
OS	Operating System
PL	Programmable Logic
PCFG	Parallel Control-Flow Graph
PMU	Platform Management Unit
POSIX	Portable Operating System Interface
PPM	Parallel Programming Model
PS	Processing System
ROS	Robot Operating Syste
RTEMS	Real-Time Executive for Multiprocessor Systems
SLG	Synthetic Load Generator
SoC	System on Chip
SLG	Synthetic Load Generator
T	Task
TDG	Task Dependency Graph
UKF	Unscented Kalman Filter

VBF	Vehicle Body Frame
WP	Work Package

8 References

[1] AMPERE, “D1.1: System models requirement and use case selection,” 2020.

[2] 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.

[3] M. Pagani, A. Balsini, A. Biondi, M. Marinoni, and G. Buttazzo, “A linux-based support for developing real-time applications on heterogeneous platforms with dynamic FPGA reconfiguration,” in *Proceedings of the 30th IEEE International System-on-Chip Conference (SOCC 2017)*, September 2017, pp. 5–8.

[4] *Zynq UltraScale+ MPSoC ZCU102 Evaluation Kit*, https://www.xilinx.com/products/boards-and-kits/ek-u1-zcu102-g.html, 2021.

[5] Xilinx. Pynq - python productivity for zynq. [Online]. Available: http://www.pynq.io/

[6] AMPERE, “D5.1: Reference parallel heterogeneous hardware selection,” 2020.

[7] *Zynq-7000 AP SoC Technical Reference Manual*, Xilinx.

[8] *Vivado Design Suite User Guide: Dynamic Function eXchange*, Xilinx.

[9] *Zynq UltraScale+ Device - Reference Manual*, Xilinx, uG1085.

[10] D. C. Schmidt, “Reactor: An object behavioral pattern for concurrent event demultiplexing and dispatching,” 1995.

[11] R. C. Martin, “Design principles and design patterns,” *Object Mentor*, vol. 1, no. 34, p. 597, 2000.

[12] M. Samek, *Object-Oriented Programming in C*, 2019.

[13] The linux driver implementer’s API guide, FPGA subsystem. [Online]. Available: https://www.kernel.org/doc/html/latest/driver-api/fpga/fpga-mgr.html

[14] *The PikeOS Hypervisor*, https://www.sysgo.com/pikeos, 2021.

[15] *Jetson AGX Xavier Developer Kit*, https://developer.nvidia.com/embedded/jetson-agx-xavier-developer-kit, 2021.

[16] “Deliverable D5.1 Reference parallel heterogeneous hardware selection,” https://ampere-euproject.eu/results/deliverables, 2021.

[17] *OpenMP specifications*, https://www.openmp.org/specifications/, 2021.

[18] *ERIKA Enterprise RTOS*, http://www.erika-enterprise.com/, 2021.

[19] *AUTOSAR Classic specification*, https://www.autosar.org/standards/classic-platform/, 2021.

[20] AMPERE, “D3.2: Single-criterion energy-optimization framework, predictable execution models, and software resilient techniques,” 2021.

[21] ——, “D1.3: First release of the meta model-driven abstraction release,” 2021.

[22] A. Munera, S. Royuela, G. Llort, E. Mercadal, F. Wartel, and E. Quiñones, “Experiences on the characterization of parallel applications in embedded systems with extrae/paraver,” in *49th International Conference on Parallel Processing-ICPP*, 2020, pp. 1–11.

[23] A. C. De Melo, “The new linux’perf’tools,” in *Slides from Linux Kongress*, vol. 18, 2010, pp. 1–42.

[24] AMPERE, “D2.2: First release of the meta parallel programming abstraction and the single-criterion performance-aware component,” 2021.

[25] S. Royuela, A. Duran, M. A. Serrano, E. Quiñones, and X. Martorell, “A functional safety OpenMP for critical real-time embedded systems,” in *International Workshop on OpenMP*. Springer, 2017, pp. 231–245.

[26] M. A. Serrano, A. Melani, R. Vargas, A. Marongiu, M. Bertogna, and E. Quiñones, “Timing characterization of OpenMP4 tasking model,” in *2015 International Conference on Compilers, Architecture and Synthesis for Embedded Systems (CASES)*. IEEE, 2015, pp. 157–166.

[27] J. Sun, N. Guan, Y. Wang, Q. He, and W. Yi, "Real-time scheduling and analysis of OpenMP task systems with tied tasks," in *2017 IEEE Real-Time Systems Symposium (RTSS)*. IEEE, 2017, pp. 92–103.

[28] M. A. Serrano, S. Royuela, and E. Quiñones, "Towards an OpenMP specification for critical real-time systems," in *International Workshop on OpenMP*. Springer, 2018, pp. 143–159.

[29] S. Royuela, R. Ferrer, D. Caballero, and X. Martorell, "Compiler analysis for OpenMP tasks correctness," in *Proceedings of the 12th ACM International Conference on Computing Frontiers*, 2015, pp. 1–8.

[30] Y. Lin, "Static nonconcurrency analysis of OpenMP programs," in *International Workshop on OpenMP*. Springer, 2005, pp. 36–50.

[31] S. Royuela, A. Duran, C. Liao, and D. J. Quinlan, "Auto-scoping for OpenMP tasks," in *International Workshop on OpenMP*. Springer, 2012, pp. 29–43.

[32] S. Royuela, A. Duran, and X. Martorell, "Compiler automatic discovery of ompss task dependencies," in *International Workshop on Languages and Compilers for Parallel Computing*. Springer, 2012, pp. 234–248.

[33] M. Norouzi, F. Wolf, and A. Jannesari, "Automatic construct selection and variable classification in OpenMP," in *Proceedings of the ACM International Conference on Supercomputing*, 2019, pp. 330–341.

[34] Embedded Brains GmbH, "RTEMS SMP - Improvement for LEON multi-core," 2017. [Online]. Available: http://www.esa.int/Enabling_Support/Space_Engineering_Technology/Software_Systems_Engineering/RTEMS-SMP_Improvement_for_LEON_multi-core

[35] R. E. Vargas, S. Royuela, M. A. Serrano, X. Martorell, and E. Quiñones, "A lightweight OpenMP4 run-time for embedded systems," in *2016 21st Asia and South Pacific Design Automation Conference (ASP-DAC)*. IEEE, 2016, pp. 43–49.

[36] A. Munera, S. Royuela, and E. Quiñones, "Towards a qualifiable OpenMP framework for embedded systems," in *2020 Design, Automation & Test in Europe Conference & Exhibition (DATE)*. IEEE, 2020, pp. 903–908.

[37] BSC, *Mercurium*, https://pm.bsc.es/mcxx, 2021.

[38] GCC, "The GOMP project," 2021. [Online]. Available: https://gcc.gnu.org/projects/gomp/

[39] BSC, "Extrae," 2021. [Online]. Available: https://tools.bsc.es/extrae

[40] ——, "Paraver," 2021. [Online]. Available: https://tools.bsc.es/paraver

[41] Intel, "Intel Xeon Platinum 8160," 2018. [Online]. Available: ark.intel.com/products/120501/Intel-Xeon-Platinum-8160-Processor-33M-Cache-2-10-GHz-

[42] BSC, "Marenostrum IV," 2019. [Online]. Available: https://www.bsc.es/marenostrum/marenostrum/technical-information

[43] Texas Instruments, "Multicore Keystone II System-on-Chip (SoC)," 2012. [Online]. Available: www.ti.com/product/66AK2H12

[44] C. Yu, S. Royuela, and E. Quiñones, "OpenMP to CUDA graphs: a compiler-based transformation to enhance the programmability of NVIDIA devices," in *Proceedings of the 23th International Workshop on Software and Compilers for Embedded Systems*, 2020, pp. 42–47.

[45] J. Lelli, C. Scordino, L. Abeni, and D. Faggioli, "Deadline scheduling in the linux kernel," *Software: Practice and Experience*, vol. 46, no. 6, pp. 821–839, 2016. [Online]. Available: https://onlinelibrary.wiley.com/doi/abs/10.1002/spe.2335

[46] A. Mascitti and T. Cucinotta, "Dynamic Partitioned Scheduling of Real-Time DAG Tasks on ARM big.LITTLE Architectures," in *29th International Conference on Real-Time Networks and Systems (RTNS 2021)*, Nantes, France, April 2021.

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

[48] A. Mascitti, T. Cucinotta, M. Marinoni, and L. Abeni, “Dynamic partitioned scheduling of real-time tasks on ARM big.LITTLE architectures,” *Journal of Systems and Software*, vol. 173, p. 110886, 2021. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0164121220302764

[49] J. Redmon and A. Farhadi, “Yolov3: An incremental improvement,” *arXiv preprint arXiv:1804.02767*, 2018.