



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

D3.4 Evaluation of multi-criteria optimizations

Version 1.0

Documentation Information

Contract Number	871669
Project Webpage	https://www.ampere-euproject.eu/
Contractual Deadline	30.06.2023
Dissemination Level	Public (PU)
Nature	R
Authors	Luis Miguel Pinho, Tiago Carvalho, Mohammad Samadi Gharajeh (ISEP)
Contributors	Sara Royuela (BSC) Tommaso Cucinotta, Francesco Paladino (SSSA), Gabriele Ara (SSSA) Sergio Mazzola (ETHZ)
Reviewer	Eduardo Quinones (BSC)
Keywords	Energy-efficiency, Timing Predictability, Resilient Software, Multi-criteria Optimization



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

Change Log

Version	Description Change
V0.1	Initial version with deliverable description
V0.2	Updates on Resilience
V0.3	Updates on Energy
V0.4	Updates on Multi-criteria Optimization
V0.5	Updates on Predictable Execution Models
V0.6	Updates on Resilience
V0.7	Updates on Use case evaluation
V0.8	Updates on optimization flow
V0.9	Release for revision
V1.0	Final version after revision

Table of Contents

1	Introduction	2
2	AMPERE Non-functional Optimizations	4
2.1	Updates on Resilient Software Techniques	4
2.1.1	Evaluation of the replication mechanism	5
2.2	Updates on Energy Modelling, Estimation and Optimization	7
2.2.1	API for Power & Energy Estimation	8
2.2.2	Update to the CPU Energy Model	9
2.3	Updates on Predictable Execution Models	10
2.3.1	Evaluation of the Mapping Algorithms	11
3	Multi-criteria Optimization	15
3.1	Updates on Multi-Criteria Configuration Flow	15
3.1.1	Profiling phase	15
3.1.2	Optimization Phase	18
3.2	Updates on Multi-criteria Optimization	20
3.2.1	Validating the optimizer	21
3.2.2	Optimizer extension to support OpenMP	23
3.2.3	Extension to support task chains	23
3.3	Evaluation on PCC Use Case	23
3.3.1	Minimum power optimization	24
3.3.2	Maximum robustness optimization	24
3.4	Evaluation on ODAS Use Case	27
3.5	Evaluation on ODAS Use Case Using Mapping Heuristics	28
4	Conclusions	33
5	Acronyms and Abbreviations	34
6	References	36

Executive summary

This deliverable covers the work done during the last phase of the project within WP3. The deliverable spans 9 months' work, as defined in the Grant Agreement [1] (from month 34 until month 42, including the postponement as approved) to reach milestone 4 (MS4).

This Deliverable summarizes the progress of Work Package 3 up until Milestone 4, and is a report describing the evaluation of the multi-criteria optimization approach (considering energy-efficiency, predictable execution models and software resilient solutions for parallel execution).

All goals set out to be achieved by the end of Milestone 4 have been achieved, and the techniques presented in this report are integrated in the multi-criteria optimization framework in the AMPERE ecosystem of WP6.

1 Introduction

This documents describes the deliverable D3.4. *Evaluation of multi-criteria optimizations*, lead by ISEP, due in month 42, in a report format. D3.4 is part of WP3 - *Multi-criteria optimization*.

This report describes the evaluation of the analyses and tools for multi-criteria optimization for the AMPERE framework, integrated into the synthesis tool, which is provided within WP6 [2].

The tasks within Work Package 3 interact closely with components developed in Work Packages 1, 2, 4, and 5. An overview of the interactions of WP3 is presented in Figure 1, described here for completeness.

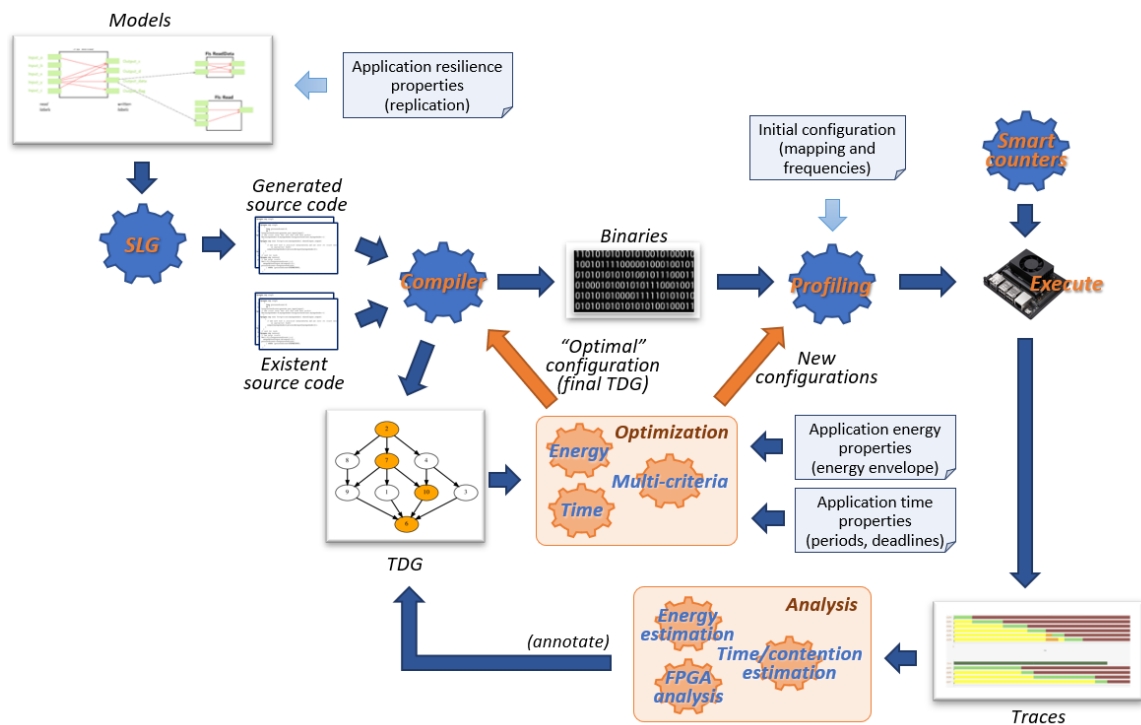


Figure 1: An overview of the interaction of Analyses and Optimization components with the other components within AMPERE.

One of the main challenges of the AMPERE project is to enable Model Driven Engineering (MDE) of physically entangled systems of systems, accounting for parallelism and heterogeneous in high-end embedded systems. As such, MDE tools provide the front-end to the entire AMPERE ecosystem, and they are represented at the top-left of the figure where Domain Specific Modeling Languages (DSML), e.g., AMALTHEA, AUTOSAR, CAPELLA, are used to describe the system in a modular and composable manner, and annotated by system designers with functional and non-functional requirements that determine how the system is generated. These are encoded in the system model as properties of system components. In particular, AMPERE focuses on the addition of non-functional requirement annotations that promote the automatic optimization of MDE driven systems of systems, with respect to energy, timing guarantees, reliability, and heterogeneity. The primary DSML used in AMPERE is AMALTHEA¹, which is representative also for AUTOSAR.

Once the system has been modeled in (or converted to) AMALTHEA, the code generator generates the corresponding source code, including annotations for the OpenMP parallel programming model (PPM), describing the dependencies and parallelism exposed in the modeled system, and passed to the OpenMP compiler for compilation into binaries. Importantly, the OpenMP compiler has been extended to not only produce the

¹The AMPERE project also develops a CAPELLA to AMALTHEA *bridge*, to which translates CAPELLA models into their AMALTHEA counterpart.

binary images themselves, but also structured information about the system, which is used for the optimization phase, described in this deliverable, to ensure that the final system fulfills all requirements modeled in the DSML. The fundamental data structure generated as part of the structured information is the Task Dependency Graph (TDG).

The TDG provides the structure of dependencies between tasks and runnables as outlined in the DSML, as well as additional meta-information that is used for the optimization. Additionally, the TDG contains profiling information from the generated system, through the execution of specific profiling tools. As part of the compilation process, the generated binary image is profiled, and the information embedded in the TDG. As such, the TDG provides the necessary abstraction for determining the modeled requirements and dependencies of every task in the system, as well as deep information about the behavior of each task as achieved through profiling. This TDG is the input to the optimization phase developed in WP3, and which this deliverable reports on. Common to all optimization components in the system is that they use the profile-based information to populate the additional TDG members, e.g., the execution time.

The analyses and optimization phases, as developed in WP3 and evaluated in this deliverable, are highlighted in the green boxes in Figure 1. It consists of multiple components developed in parallel: the timing analysis and optimization, energy optimization, and scheduling. Heterogeneity and resilience techniques are implemented at the model and compiler level, through the definition of variants (components which have multiple implementations) and replicas (for fault-tolerance). These are inserted into the TDG, and taken into consideration by the analyses.

Returning to Figure 1, once the optimization phase has completed, it is either finished, i.e., all functional and non-functional requirements are guaranteed to be upheld, or another round of optimization is required. The AMPERE optimization relies on an optimization loop as reported in D2.4 [3], and additional profiling information may be required to complete the optimization stage. This is achieved using the outcome of the optimization stage, as shown in the figure (red arrows), which either exits the optimization loop once all optimization components have successfully optimized the respective non-functional properties (at that point, the binary system as compiled from the sources corresponding to the TDG is final) or require additional information, e.g., a new profiling run (another iteration of the optimization loop is triggered).

As the TDG is a common data structure between the work packages, it also enables a second important feature, at the end of the optimization phase. The encoded information in the TDG allows for later use by the earlier components in the AMPERE pipeline, such that information in the model could either be updated, or warnings emitted to the MDE framework, and made available to the end user.

At the end of the optimization pipeline, the TDG information can also be used to inject runtime hooks and configuration headers based on the optimization outcome into the generated source code. This enables actuation and monitoring of the non-functional requirements optimized for in WP3 at runtime, in accordance with the expressed goals of the project.

As part of Milestone 3, reported in deliverable D3.3 [4], the optimization phase considers a multi-criteria optimization phase, in which all components are co-operating to ensure that the system fulfills *all* requirements modeled in the DSML. The remainder of this deliverable presents the updates on optimization analyses and tools, related to deliverable D3.3 [4], and the updates on the multi-criteria approaches provided in deliverable D2.4 [3]. These analysis and tools are then evaluated in the AMPERE use cases [5].

2 AMPERE Non-functional Optimizations

2.1 Updates on Resilient Software Techniques

The AMPERE ecosystem includes two different techniques to achieve resilience: (1) task-level parallel replication through OpenMP (based on the definition of SIL/ASIL levels in the models), and (2) dynamic monitoring through fine-grained proactive orchestration. These mechanisms detailed in D3.3 [4], and their features are included in demonstrator D2.3 [6].

The proactive orchestration mechanism has not changed from D3.3. On the other hand, the replication mechanism has evolved to include the mechanisms that were missing at MS3, i.e., (1) further types of replication (besides spatial replication) that include temporal replication and the combination of temporal and spatial replication, and (2) support for the *MooN* safety architecture specification.

```
#pragma omp task replicated(n, var:func [;var:func...] [, spatial|temporal|spatial_temporal])  
{ /*functionality to replicate*/ }
```

, where:

- *n* is the number of replicas to be created (besides the original task),
- *var:func* is a tuple variable:function used to check the results, used by calling *func* with the original and the replicated values of *var* as arguments, and
- *spatial|temporal|spatial_temporal* defines the type of replication, where *spatial* forces each replica and the original task to be executed in a different processor, allowing them to run in parallel; *temporal* forces each replica and the original task to be executed in mutual exclusion among them, so they have to be sequentialized, and *spatial_temporal* includes both cases, i.e., each replica and the original task run in a different processor, and no pair can run concurrently. If no type is defined, then the default behavior is to allow parallelism between the replicas and not imposing any restriction of the executing core.

The syntax and semantics of the proposed OpenMP mechanism have been implemented in the LLVM compilation framework, including support in Clang (the C/C++ frontend), LLVM (the compiler) and KMP (the OpenMP runtime). In the extended LLVM, when a task with a replicated clause is found, $n+1$ tasks are created and associated with the same task region. One of the tasks consumes the original data, while the rest consume copies of the written data to avoid race conditions. A synchronization task is inserted after the creation of the tasks. This task has input dependencies with all the tasks in the replication set, and inherits the output dependencies from the original task. Afterwards, for each *var : func* pair, one consolidation task per replica is generated. Consolidation tasks contain a pointer to the specified user function, which performs the comparison between the original results and those from the replica, and returns a boolean expressing the correctness (equality) of the results. The function must define a header with two parameters, one pointer to the original data, and one pointer to the replicated data. See further details in D3.3 and D4.3 [7].

For spatial replication, isolation is ensured by automatically generating a unique identifier for each replication set. This identifier is passed to the runtime to prevent any thread from executing more than one task with the same identifier. To avoid migration of threads between cores (hence preventing one thread idle while replicated tasks are still pending), the `proc_bind` runtime variable is used to pin threads to cores. For temporal replication, sequentialization is enforced by adding dependencies among all tasks in the replication set.

Moreover, the proposal supports replication optimizations targeted for safety architectures. Safety architectures demand a certain level of consensus to ensure fault tolerance, but usually it does not require all nodes to agree. In this sense, the user can specify how many replicas with a correct result are required, and the rest of replicas can be canceled before they execute, obtaining a benefit in performance. This optimization is implemented through a new flag in Clang, where the user specifies the number of replicas to create and the number of replicas that must have a correct result (notice that this value overwrites the number of replicas specified in the replicated clauses). The syntax is as follows:

`-fopenmp-replication-arch=MoonN`

, where

- N is the total number of replicas and
- M is the number of replicas required to finish correctly.

The evaluation of the task-level replication mechanism on top of an implementation provided by the University of Siena (UNISI) of the tracking sub-system of the ODAS use case was presented in D3.3 and derived the following conclusions:

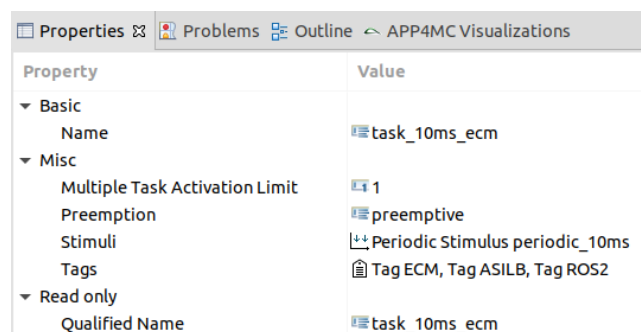
1. The overhead introduced by the technique highly depends on the amount of resources that are idle while executing the non-replicated application. In the case addressed, a maximum overhead of 85% was reached when using triple replication on the most time consuming part of the application.
2. The mechanism is suitable for detecting erroneous results, but it is not that good detecting silent errors. An average accuracy of almost 80% is reached when inserting errors in relevant paths of execution.
3. The programmability of the technique is very good as it does not require to modify the original code and only annotations to the OpenMP directives are needed.

2.1.1 Evaluation of the replication mechanism

This deliverable completes previous evaluations of the resilience mechanisms presented in D3.3 [4] by (1) evaluating the synthetic code generated for the AMALTHEA model of the PCC use case with extensions for parallelism and resilience, (2) completing the evaluation presented above on the real code of the tracking sub-system of the ODAS model by testing the features extended during the last phase of the project to reach MS4 (i.e., different types of replication and using safety architectures), and (3) evaluating the synthetic code generated for the AMALTHEA model of the ODAS use case with extensions for parallelism and resilience.

2.1.1.1 PCC use case

The PCC use case exposes one task with an ASIL_B tag, i.e., *task_10ms_ecm* from the ECM component (see Figure 2), which calls 304 runnables of a granularity around $10^{-1} \mu s$ each, and one component that can be considered ASIL_A or ASIL_B, i.e., the ACC component. The rest of tasks have no safety constraint.

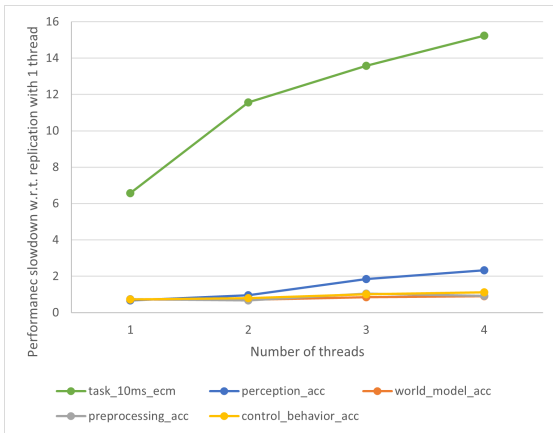


Property	Value
Basic	
Name	task_10ms_ecm
Misc	
Multiple Task Activation Limit	1
Preemption	preemptive
Stimuli	Periodic Stimulus periodic_10ms
Tags	Tag ECM, Tag ASILB, Tag ROS2
Read only	
Qualified Name	task_10ms_ecm

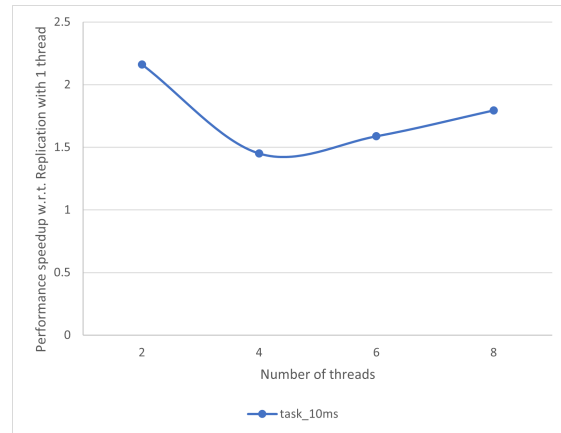
Figure 2: ASIL_B tag defined in *task_10ms_ecm* from the PCC use case.

Applying replication in these tasks not only increases the load of the system considerably in terms of units of work (i.e., replicated tasks plus consolidation functions), but also, in the case of the ECM task, it forces the introduction of OpenMP in a component where our previous analysis had recommended not to use inter-runnable parallelism due to the granularity of the runnables. Consequently, the system experiences a considerable increase in the execution time, that goes up to 16x for the ECM fine grained tasks when executing with 8 OpenMP threads, as shown in Figure 3a. This is mainly due to the constant competition between the OpenMP threads

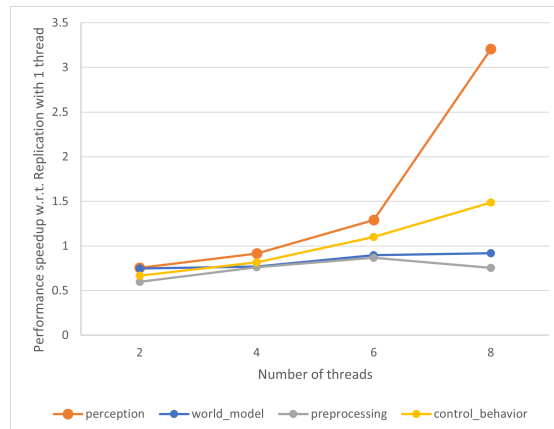
and the ROS threads managing the rest of AMALTHEA tasks. The ACC tasks, instead, do not suffer much overhead from the replication, as they can benefit from parallel resources and have a granularity more suited to the OpenMP runtime management. The slowdown is much contained when running the ECM or the ACC components in isolation. For the ECM case, Figure 3b shows the worse value of 2.2x slowdown is reached with 2 threads, and the minimum 1.5x is reached with 4 threads. For the ACC case, Figure 3c shows the worse value of 3.2x slowdown is reached with 8 threads for the *perception* task. The rest experience (almost) no slowdown at all. This opens the door to investigate additional static and runtime optimizations that could further enlighten the overhead of the runtime and so allow for obtaining benefit out of the parallel execution even in such a fine grained scenario.



(a) All PCC components running concurrently.



(b) ECM running in isolation.



(c) ACC running in isolation.

Figure 3: Performance slowdown when replicating all runnables in *task_10ms_ecm* and/or the ACC component of the PCC use case, while varying the number of threads.

2.1.1.2 ODAS use case

Real code for the tracking sub-system. The code provided by UNISI for the tracking subsystem has been evaluation beyond D3.3 with the extensions developed during the last phase of the project and presented above. Table 1 shows hence the speedup of the tracking sub-module of the ODAS application when defining a 1003 safety architecture, and spatial and temporal replication, using as baseline the default replicated application with two OpenMP threads. Only one phase is replicated at a time. As the table details, enabling the 1003 architecture results in an important benefit in performance, due to to not executing all the replicas. Spatial configuration maintains the performance, since the tasks are already executed in different cores. Meanwhile, temporal configuration forces the sequentialization of all the replicas, affecting negatively the performance

but of course enabling the detection of errors in different scenarios.

Table 1: Speedup of the tracking sub-system of the ODAS use case when using the 1oo3 safety architecture, and spatial, and temporal replication with 2 threads.

	1oo3	Spatial	Temporal
Predict	1,2	1	0,9
Associate	1,1	1	1
Update	1,3	1	0,7
Track	1,5	1	0,6

Synthetic code The ODAS model presents replication only in the UKF component, because it is the only one defining a safety constraint, i.e., SIL4, as shown in Figure 5. This component executes 60 runnables that simulate the execution of an unscented kalman filter each. The model defines these kernels with a workload of 20340000 ticks, which results in an execution time of 1582.26 μ s for the whole UKF component. This is a fine granularity, which additionally exposes a huge amount of parallelism, as the UKF component is embarrassingly parallel. The overhead introduced in the system due to tripling the workload and adding as many consolidation functions as UKFs makes the replicated version to perform poorly when incrementing the number of threads. The competition for resources with other ROS nodes makes the version not suitable for parallelization. However, when isolating the UKF component, the parallelization shows performance benefits up to 2x for 6 threads. Again, there has to be a balance between the amount of parallelism exposed and the amount of resources available, otherwise over-subscription occurs and the overhead of handling the parallel system defeats the benefits of the parallelization.

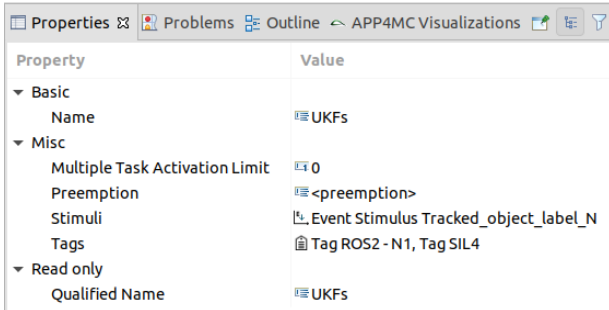


Figure 4: SIL4 tag defined in the UKFs task from the ODAS use case.

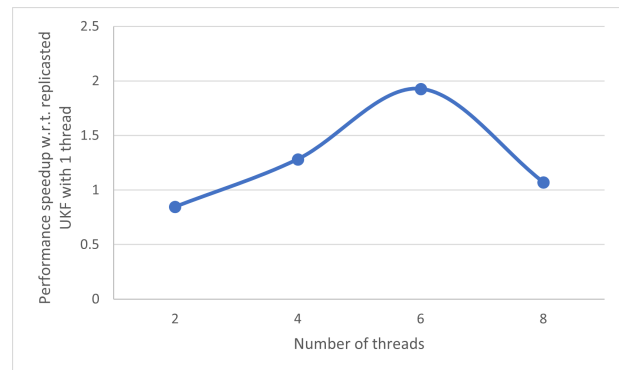


Figure 5: Performance speedup of the UKF component of the ODAS use case when replicated and running in isolation, while varying the number of threads.

2.2 Updates on Energy Modelling, Estimation and Optimization

Energy consumption is one of the non-functional metrics targeted for the multi-criteria optimization in the AMPERE framework. AMPERE energy consumption optimization is built on top of the power models presented in [8], and it is driven by hardware Performance Monitoring Counters (PMCs). Opposed to analog power sensors, PMC-driven energy estimation can provide fine-grain, responsive, and accurate measures with negligible performance overhead in a non-invasive and flexible way. Our methodology can support a broad range of heterogeneous platforms with DVFS capabilities.

The deliverable D3.1 [9] already points out the rationale behind PMC-based power modeling for modern heterogeneous systems, and the workflow set up to obtain an accurate and lightweight power model of an ARM

CPU. The deliverable D3.2 [10] builds on top of that and explains how the devised power model can be used, during the offline single-criterion optimization step, to optimize the energy consumption of a given task in the AMPERE ecosystem. The deliverable D3.3 [4] refines the approach explored in the previous two deliverables, generalizing it and validating it for the NVIDIA Jetson AGX Xavier target platform [11, 12].

This section presents the work on energy consumption estimation carried out in relation to MS4, in the context of the offline multi-criteria optimization flow. In particular, the contribution described in this section is twofold:

- *Application Programming Interface (API) for power and energy estimation*: to allow the usage of our energy estimates during multi-criteria optimization, we develop a flexible, scalable, and user-friendly API easily integrated in the rest of the AMPERE ecosystem;
- *Power model refinement for NVIDIA Jetson AGX Xavier’s CPU*: we update the power model at the basis of the Xavier’s CPU energy consumption estimation for integration in the multi-criteria optimization flow.

2.2.1 API for Power & Energy Estimation

The aim of energy consumption analysis in the AMPERE multi-criteria optimization flow is to obtain energy consumption estimates for each task of a given TDG describing the target use case. The information provided by the task-level energy estimates can subsequently be exploited to optimize the application for other non-functional metrics, under energy constraint, or vice versa.

To allow so, we develop a user-friendly API to query our power models [4] for power and energy estimates. Such API is scalable, as it can be easily extended to transparently provide power and energy estimates for a broad range of platforms and their sub-systems, and flexible, as it can integrate any PMC-based power model and supports fine-tuning.

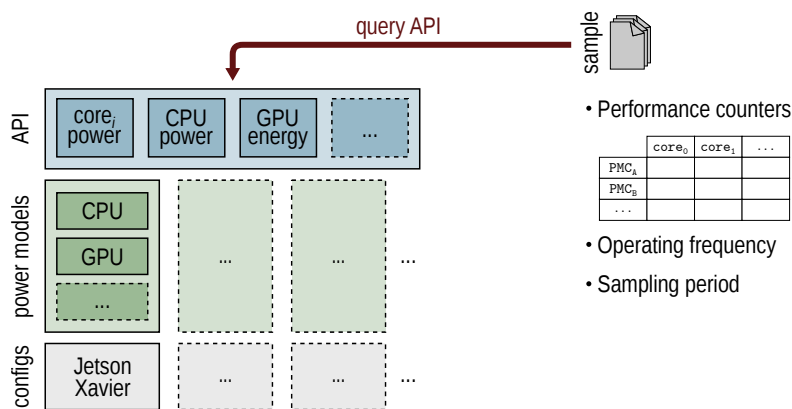


Figure 6: The structure of the API and its underlying resources. Based on the available platform *drivers*, i.e., the available power models for platforms and their sub-systems, the power and energy consumption for different devices can be queried. The structure of the query input is also displayed.

Figure 6 shows the structure of the power and energy API and its underlying resources. The tool is Python-based and easily extensible to support more target platforms. The support for a platform consists in a configuration file, in the lowest level of Figure 6. The configuration file describes the architectural parameters of the platform and its sub-system, along with the trained power models of each sub-system for each desired frequency. Currently, the NVIDIA Jetson AGX Xavier is supported. The power models implementation for each sub-system of each platform builds on top of the platform configuration. This layer (the middle one in Figure 6) actually employs the trained weights and the platform configuration to implement the PMC-based power estimation models. Energy estimates can also be provided based on the modelled power and on time measures provided as inputs. For the Jetson Xavier, the tool currently supports power models for CPU and GPU. The power and energy API builds on top of the power models. By providing

- the PMC samples for a sub-system of a target platform,

- the operating frequency at which the samples have been acquired,
- and the sampling period which they cover,

the API can be queried to estimate the power or energy consumption of any platform sub-system (e.g., a CPU core, the whole CPU, a GPU, ...). The API is fully platform-independent and support for new platforms can be added transparently and easily by extending the power models and configuration layers. The current API is reported in the following.

- `power_cpu_core_i(i: int, pmcs: dict, freq: int, sample_per: float) -> float`
- `power_cpu(pmcs: dict, freq: int, sample_per: float) -> float`
- `power_gpu(pmcs: dict, freq: int, sample_per: float) -> float`
- `power_soc(pmcs_cpu: dict, freq_cpu: int, pmcs_gpu: dict, freq_gpu: int, sample_per: float) -> float`
- `energy_cpu_core_i(i: int, pmcs: dict, freq: int, sample_per: float) -> float`
- `energy_cpu(pmcs: dict, freq: int, sample_per: float) -> float`
- `energy_gpu(pmcs: dict, freq: int, sample_per: float) -> float`
- `energy_soc(pmcs_cpu: dict, freq_cpu: int, pmcs_gpu: dict, freq_gpu: int, sample_per: float) -> float`

The API is easily installed through `pip` and employed in any Python script through an `import` statement.

2.2.2 Update to the CPU Energy Model

For MS4, the power models and the methodology described in the previous deliverables [9, 10, 4] were applied to the overall multi-criteria optimization flow for validation purposes. From the integration process, it resulted that not every performance counter required by the models trained in D3.3 [4] could be correctly retrieved by the rest of the AMPERE toolset, in particular from Extrae, extensively used for profiling during multi-criteria optimization. The reason of this issue has to be researched in the low-level approach that we use for PMC samples collection, which relies on direct access to the ARM PMU registers via assembly instructions during profiling. Extrae, on the other hand, relies on PAPI, hence it might inherit limitations depending on PAPI's target platform support.

Table 2: The ten best CPU counters from platform characterization at each frequency, with their Pearson Correlation Coefficient (PCC) with power consumption. The counters selected for the old CPU model are reported in green.

Performance event	Frequency [MHz]		
	730	1190	2266
Cycles counter	0.56	0.57	0.60
Exceptions taken	0.51	0.54	0.57
Instructions retired	0.52	0.57	0.55
Floating-point activity	0.54	0.56	0.59
SIMD activity	0.50	0.52	0.52
Speculative branch	0.49	0.52	0.53
Speculative load	0.52	0.53	0.54
Speculative L/S	0.51	0.53	0.53
L1 instr. cache accesses	0.53	0.54	0.56
L1 data cache accesses	n/d	0.54	n/d
Data memory accesses (read)	0.52	0.53	0.54

The results of the Jetson Xavier's CPU platform characterization (i.e., selection of the best PMCs to model CPU power consumption) are discussed in Section 3.2.2.1 of D3.3 [4]. For completeness, we report in Table 2

a summary of those results, highlighting the initial selection of performance counters. The issue in Extrae is related to the counter of *Exceptions taken*, selected to model power consumption at 2266 MHz. As a solution, we take the next best performance counter which Extrae is able to sample, namely the counter for *Instructions retired*, and we re-train the CPU power model as discussed in D3.3 [4]. We estimate a minimal loss in accuracy due to the fact that both counters correlate in a similar way to power consumption, as indicated by their Pearson Correlation Coefficient (PCC).

The other power models remained unchanged, as in D3.3 [4].

2.3 Updates on Predictable Execution Models

A new version of the task to thread mapping algorithms of Gharajeh et al. [13] (as described in D3.3 [4]) have been implemented, which improve efficiency by considering the existence of per thread mapping queues (similar to LLVM's default scheduler, which implements a version of the work-stealing scheduler, known to be highly performant [14]). Figure 7 shows the main elements of the proposed mapping method, which includes two phases: allocation and dispatching. These phases are carried out simultaneously. In the allocation phase, each task-part from the task system is allocated to one of the thread queues, which includes both sibling and child task-parts. In the dispatching phase, an idle thread selects a task-part from its queue and executes it.

The mapping algorithms in D3.3 [4] have therefore been instantiated for both phases. In the allocation phase, the implemented algorithms include:

- The MNTP (Minimum Number of Task-Parts) heuristic: This algorithm selects the thread queue that contains the minimum number of task-parts with the objective of increasing the load-balance of the system. Since the scheduler is aware of the number of task-parts in each queue, the algorithm has complexity of $O(m)$, where m is number of threads.
- The NT (Next Thread) heuristic: This algorithm selects the queue of the next thread according to the last active thread. The main objective is to balance the propagation of task-parts across all queues. Since the scheduler knows the last active thread, the algorithm is of complexity $O(1)$.
- The MRIT (Most Recent Idle Time) heuristic: This algorithm selects the queue with the longest idle time (i.e., longest time without executable task-parts in the queue), with the objective of balancing the workload of the system by decreasing the idle time of threads. As a thread is selected using a comparison between threads based on their longest idle time, the algorithm has complexity of $O(m)$, where m is number of threads.
- The MTET (Minimum Total Execution Time) heuristic: This algorithm selects the queue with the minimum total execution time based on the objective of enhancing the work-balance across threads. First, it calculates the sum of execution time of the task-parts in each queue. Then, the queue with the minimum time is selected. The complexity of the algorithm is $O(m*n)$, where m is number of threads and n is number of task-parts.
- The MTRT (Maximum Total Response Time) heuristic: This algorithm selects the queue that includes the maximum total response time, with the objective of increasing data locality between tasks. However, this will reduce the balance of ready jobs, and so evaluate the system performance against the worst-case condition. The complexity of the algorithm is $O(m*n)$, where m is number of threads and n is number of task-parts.

The dispatching phase is carried out in each queue separately. A task-part is selected from the queue using one of the following heuristic dispatching algorithms:

- The MET (Minimum Execution Time) heuristic: This algorithm selects a task-part based on the minimum execution time. The main goal is to achieve the work-conserving characteristic of the mapping process by executing shorter task-parts first, and accordingly reduce the waiting time of the tasks to start. The complexity of the algorithm is $O(n)$, where n is the number of task-parts in the queue.

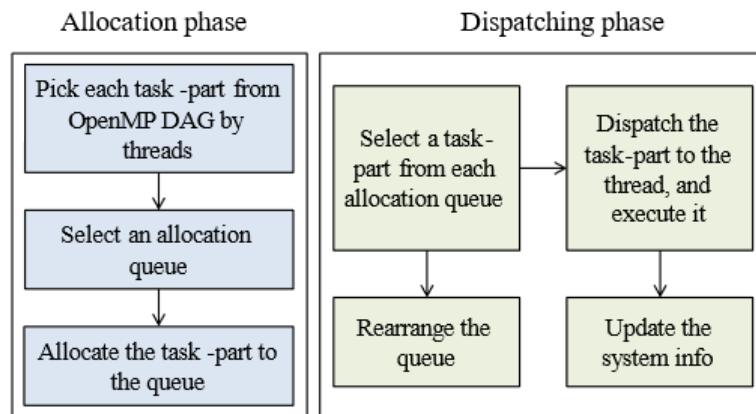
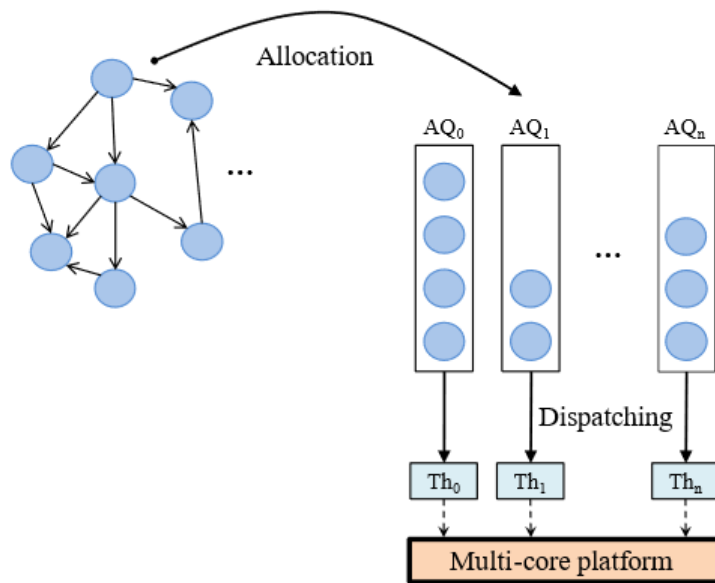


Figure 7: Mapping process

- The MRT (Maximum Response Time) heuristic: This algorithm chooses the most appropriate task-part based on the maximum response time. The purpose of this heuristic is to reduce the workload of each queue and increase its chance of getting new task-parts. The complexity of the algorithm is $O(n)$, where n is the number of task-parts .

These algorithms are included in the AMPERE mapping method [15] (as described in D3.3 [4]), with different algorithms being evaluated for static mapping when time optimization is the only criteria.

2.3.1 Evaluation of the Mapping Algorithms

The performance of the mapping methodology, using the algorithms described in the previous section, was evaluated by simulation, under different numbers of tasks by comparing the results with those of BFS, WFS and LNSNL [16], in terms of response time. The simulator allows to simulate the algorithms execution with emulated time (single thread using the loop tick) or elapsed time (with multiple running threads and a clock in a separate thread). The simulation process can be done in one or multiple iterations.

For the evaluation, random OpenMP graphs are generated following three different approaches. In system model 1 (SM1), the simplest, there is one single parent tasks that creates concurrent child tasks (they have

no dependences among them). This model is very similar to common applications, where there are no child tasks. In system model 2 (SM2), the graph is (potentially) very complex, as each task-part can create and have an unlimited number of child tasks. In system model 3 (SM3), each task-part has a minimum number of child tasks and a maximum number of child tasks. Note that it is assumed the full TDG to be known before the mapping process and data dependencies can be defined between task-parts. Figure 8 shows three examples for the system models with 10 tasks, where one level of nesting is used in SM1 (left), each task-part can create an unlimited number of child tasks in SM2 (center), and each task-part creates only one child task in SM3 (right).

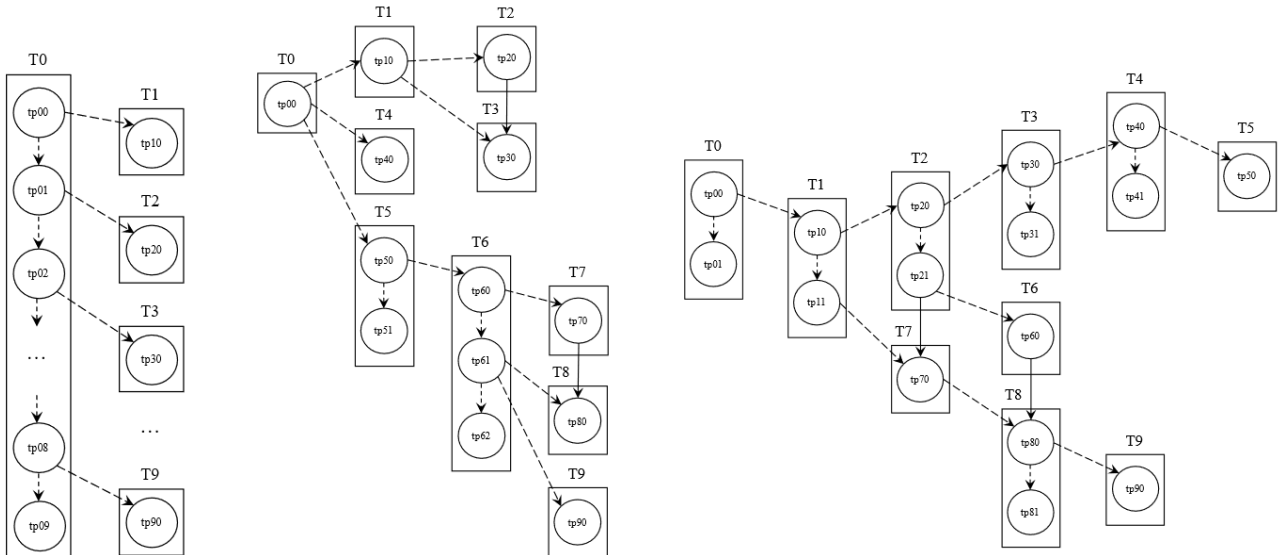


Figure 8: Examples of the system models applied in the mapping process

The simulation process is performed in two scenarios: (1) including overhead of the mapping process, working as an online mapping, and (2) without including overhead, working as an offline mechanism. In the first scenario, the overhead of the mapping process is included, so it works similar to an online mapping. But in the second scenario, the overhead of the mapping process is not included, so it works like an offline mapping. Table 3 represents the simulation parameters with their default values. The number of task-parts for each task is 1 in SM1, 3 in SM2, and 2 in SM3. Note that a random graph is generated at each iteration so that a random execution time is specified for each task-part in the graph.

Table 3: Simulation parameters

Parameter	Default value
Number of tasks	[100, 1000]
Number of task-parts {SM1, SM2, SM3}	{1, 3, 2}
Minimum execution time	5
Maximum execution time	20
M (minimum number of child tasks in SM3)	5
N (maximum number of child tasks in SM3)	10
Probability of selecting sibling tasks	0.6
Maximum number of dependencies between siblings	3
Number of threads	{4, 8}
Number of iterations (number of times the method is run)	10

Evaluation was performed generating random graphs, where the number of tasks is defined by the programmer, while the number of task-parts for each task is determined randomly (except in the first system model, where each task includes only one task-part). The parent-child relationship between tasks and the data dependency between task-parts in the whole DAG are specified randomly. The execution time of task-parts is specified randomly in multiple iterations, but the WCET for each task-part (calculated based on the maximum of the random numbers) is considered in the simulation process. Moreover, the application deadline is determined randomly and the response time for each task-part is calculated based on the task-part's execution time and the application deadline.

In each mapping algorithm, the simulation process runs until all task-parts are dispatched to and executed by threads. In the case with threading, the application's runtime is determined using a clock (which is controlled by a different thread), while in the other case, it is determined using the loop's tick. The simulation process for tied and untied tasks is conducted separately. Finally, the application response time, the idle and waiting time of threads, and the number of missed deadlines are calculated at the end of the simulation. Note that the simulations including overhead are produced using the case with threading (elapsed time), but the simulations without including overhead are produced with the case without threading (logical time).

Figure 9 provides an example of the results of the evaluation for the system model 1, considering the overhead scenario. The simulation results show that (i) WFS works significantly worse than the others for tied tasks, (ii) LNSNL works worse for untied tasks, and (iii) MTET-MET works very similar to BFS, both outperforming WFS and LNSNL in most of the cases. In fact, in the majority of the scenarios, MTET-MET outperformed the other mapping approaches.

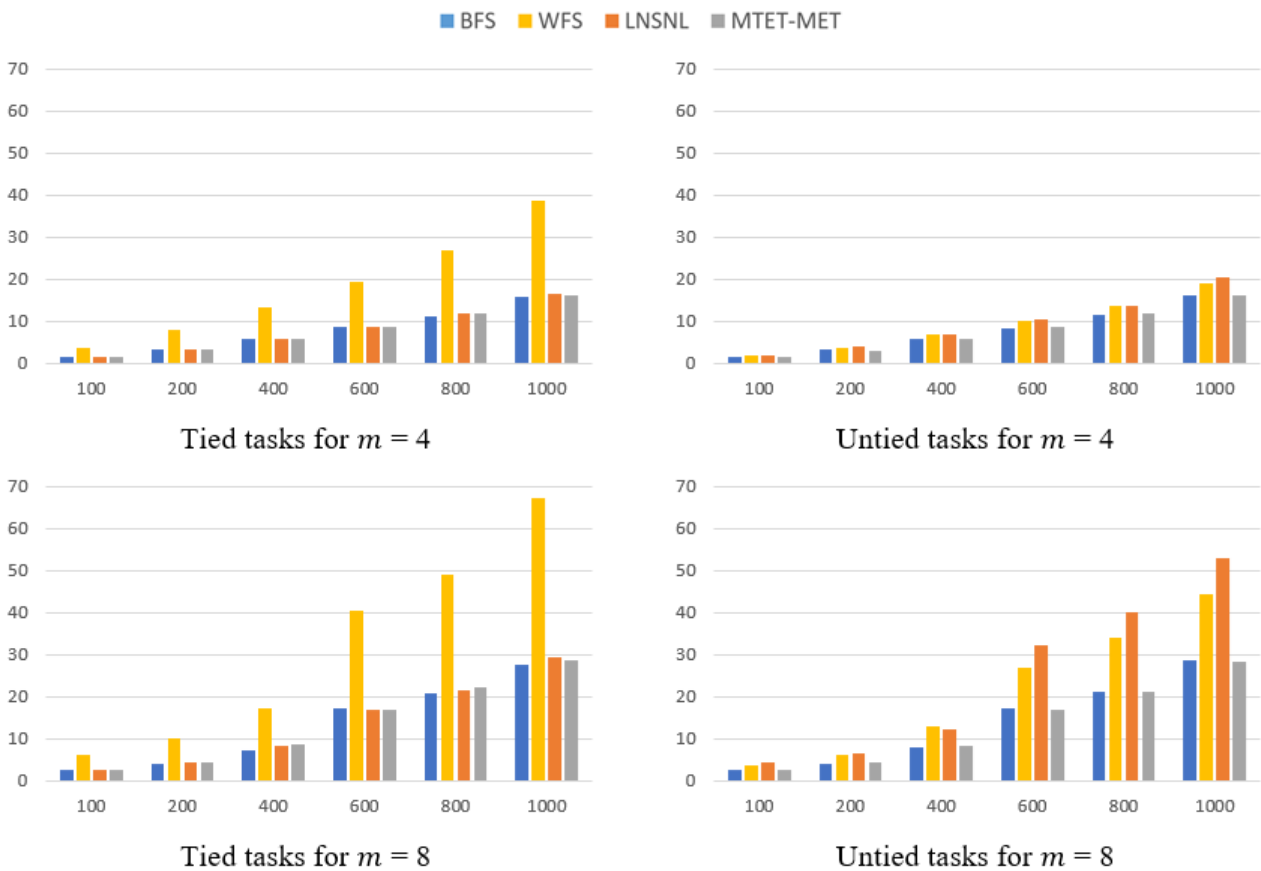


Figure 9: Evaluation in system model 1 with overhead

Table 4 details the improvement of the proposed method, compared to the others, for tied and untied tasks in terms of response time including overhead, where all the system models are considered and the number of

threads is 4 and 8. MTET-MET outperforms BFS for SM2 and SM3, while BFS presents slightly better results for SM1. MTET-MET also outperforms WFS, being even more effective for tied tasks, reducing the time up to almost 89%. Finally, compared to LNSNL, MTET-MET is better in most of the cases, showing increased performance for 8 threads, and for untied task, reaching up to 67% gain. Overall, the proposed MTET-MET mapping algorithm shows better results than the other methods in reducing response time when reproducing on-line scenario by including the overhead of the mapping algorithm.

Table 4: MTET-MET compared to BFS, WFS and LNSNL methods, including overhead

System model	Number of threads	Tied tasks			Untied tasks		
		BFS	WFS	LNSNL	BFS	WFS	LNSNL
SM1	$m = 4$	-2.52%	56.04%	-0.01%	-2.23%	21.71%	13.49%
	$m = 8$	-4.03%	55.99%	-0.19%	-3.64%	39.21%	33.66%
SM2	$m = 4$	10.03%	78.28%	22.12%	10.29%	37.25%	34.79%
	$m = 8$	20.13%	87.67%	37.29%	18.77%	43.81%	49.62%
SM3	$m = 4$	10.98%	77.99%	29.53%	13.56%	37.89%	51.42%
	$m = 8$	19.88%	88.64%	49.06%	23.19%	47.49%	67.01%
Average:		9.08%	74.10%	22.97%	9.99%	37.89%	41.67%

Table 5 represents the improvement of the new mapping method, compared to BFS, WFS and LNSNL, in terms of response time, when the overhead is not considered in the simulations (note that with static mapping there is no overhead in the execution). Compared to BFS, MTET-MET works better, except for SM1, where it is slightly worse. Compared to WFS, MTET-MET works very efficiently, especially for tied tasks, being more noticeable for SM2 and SM3 with tied tasks. Compared to LNSNL, the efficiency of MTET-MET is better, except for SM1 with tied tasks, in which case it is similar. In summary, the proposed method works efficiently, compared to the other methods, especially for untied tasks, in a scenario reproducing off-line behavior by disregarding the overhead of the mapping algorithm.

Table 5: MTET-MET compared to BFS, WFS and LNSNL methods, without overhead

System model	Number of threads	Tied tasks			Untied tasks		
		BFS	WFS	LNSNL	BFS	WFS	LNSNL
SM1	$m = 4$	0%	49.82%	0%	1.43%	11.41%	1.41%
	$m = 8$	0%	49.90%	0%	2.39%	11.79%	2.42%
SM2	$m = 4$	10.78%	74.47%	4.91%	18.87%	31.87%	4.75%
	$m = 8$	4.61%	86.40%	3.36%	11.39%	28.18%	3.35%
SM3	$m = 4$	8.92%	74.02%	2.82%	19.09%	32.11%	4.89%
	$m = 8$	6.60%	86.92%	4.72%	13.12%	29.65%	4.98%
Average:		5.15%	70.26%	2.64%	11.05%	24.17%	3.63%

3 Multi-criteria Optimization

3.1 Updates on Multi-Criteria Configuration Flow

The multi-criteria flow framework aims to profile, analyse and deliver the most fitting configuration per optimization criterion for a given application, to execute in a given platform. The optimized configuration includes both application and system parameters to be optimized.

For MS4, The pipeline of the multi-criteria configuration flow was extended and adapted to include additional components. Figure 10 depicts an overview of the multi-criteria optimization flow, abstracted to three essential components: profiling, analysis and optimization. The components were improved to better deal with applications with multiple OpenMP parallel regions, reflected as multiple TDGs in the same TDG.json file. The profiling component, besides some changes in the input configuration file, was extended with a compilation phase and data extraction from amalthea models. The analysis phase kept its functionalities by using time and energy analysis tools. The optimization loop phase was divided in two optimization branches: one optimizing the application per optimization criterion (similar to the approach described in [3]), and another flow using the multi-criteria optimization described in Section 3.2. Since the analysis format was maintained, this section focus on describing the profiling and optimization phases.

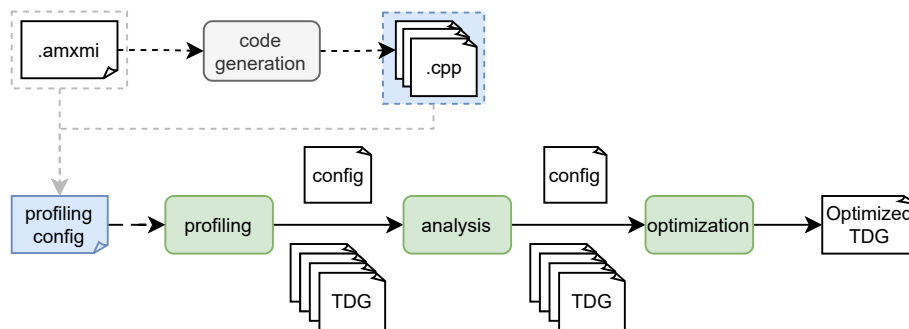


Figure 10: Overview of the multi-criteria optimization flow.

3.1.1 Profiling phase

This phase was updated mainly to better deal with applications with multiple parallel regions, to retrieve information from amalthea models, and to allow code compilation at profile time.

The profiler deals with multiple versions of the same application, i.e. different combinations of the function specializations, by executing (or simulating) the application a specified number of times (defined in the configuration), with different system configurations. In terms of structure, the multiple versions share the same TDG. The only difference among them is the specialization represented by a given node, more specifically, if the functionality is to be executed in the CPU, GPU, FPGA, or any other accelerator device. The information regarding the version of each specialized functionality in the application and the instructions to setup the target system are defined in the profiling configuration file. Figure 11 illustrates the new structure of this file, which is now divided into three sections of properties: application, platform, and optimization.

The application section includes information regarding:

- the Amalthea model from which the application was designed; and
- the Amalthea constraint that contains the end-to-end deadline and/or task event chain constraint
- the variants, i.e., the code version information for each variant.

```

1  {
2    "app": {
3      "name": "TSR",
4      "amalthea_model": "./models/PCC_Jetson.amxmi",
5      "latency_constraint": "Req_Deadline_EC_Tsr",
6      "variants": {
7        "cpu": {
8          "dir": "./cpu_tsr",
9          "build": {
10             "dynamic_mapping": "colcon_build",
11             "static_mapping": "colcon_build"
12           }
13          "dots_dir": "build/amalthea_ros2_model",
14          "run": ". ./install/setup.sh && _ros2_run_amalthea_ros2_model_TSR",
15          "iterations": 100,
16        },
17        "gpu": {
18          ...
19        }
20      }
21    },
22    "platform": {
23      "selected": "Xavier",
24      "platforms": {
25        "Xavier": {
26          "setup": "echo 'userspace' | _sudo_tee _/sys/.../cpu/cpufreq/policy0/scaling_governor",
27          "cpu": {
28            "cmd": [
29              "echo {frequency} | _sudo_tee _/sys/.../cpu/cpufreq/policy0/scaling_setspeed",
30              {"OMP_NUM_THREADS": "{threads}"}
31            ],
32            "args": {
33              "frequency": [729600, 1190400, 2265600],
34              "threads": {
35                "start": 1,
36                "stop": 16,
37                "step": 1
38              }
39            }
40          },
41          "gpu": {...},
42          "cleanup": "echo 'ondemand' | _sudo_tee _/sys/.../cpu/cpufreq/policy0/scaling_governor"
43        },
44        ...
45      }
46    },
47    "optimization": { ... }
48  }

```

Figure 11: Sample profiling configuration file with some of the basic properties.

Each variant contains information of its location, how to compile it, where the dot files will be generated, how to execute the compiled version, and how many times the executable shall be executed. The compilation is divided in two properties, dynamic and static mapping, where the former is used to compile the application using the default OpenMP scheduling algorithms, and static mapping is used to compile the application using a pre-defined mapping for each OpenMP task, for each parallel region. In the profiling phase, only the `dynamic_mapping` is used, while the `static_mapping` will be used in a later optimization phase.

For the target system (platform), the configuration specifies how is it possible to reconfigure the system with different parameters. More specifically, this section defines the commands to reconfigure the CPU (Line 27 of Figure 11), and any accelerator device available in the system, via the `cmd` attribute, where it is possible to define (an array of) multiple commands as necessary, to obtain the desired configuration. The most relevant commands are those that reconfigure the system and that affect considerably the performance of the functionalities and, subsequently, the response time of tasks, and include the following aspects:

- the frequency, as exemplified in the command of Line 29, using as argument the value of *frequency*; and

- the number of OpenMP threads (in the CPU section, Lines 29 and 30), by setting the `OMP_NUM_THREADS` environment variable, using as argument the value of `thread`.

Commands are a single element or an array of commands, that can be either a string of the command to be executed (such as in Line 29), or a `key → value` object, where each key represents an environment variable to be setted with the corresponding value (Line 30).

Commands can be parameterized through the `{param_name}` notation, which can be specified as: a scalar value, an array of values (Line 33), or a range (Line 34). The profiler is responsible for testing all possible combinations of parameter values (a configuration) and execute the program in the different setups, in the different variants.

Finally, the optimization section contains additional information to be passed to the optimization phase, when necessary. This section is dependent on the optimizer and, if empty, the optimizer should consider a default configuration. An example for this section is the specification of the algorithms for the static mapping. If none are given, then the mapping exploration will consider the all the studied algorithms (see Section 2.3).

3.1.1.1 Running the Profiler

The profiling phase is composed of one main script and depends on a set of monitoring tools included in the AMPERE ecosystem to provide measurements from executions. The main script starts by compiling each provided version with the compilation commands specified as "dynamic mapping". This will build an executable that runs using the default dynamic task to thread mapping algorithm of OpenMP [17].

The compilation process is expected to be performed using an extended version of LLVM [18] developed by the Barcelona Supercomputing Center (BSC). The features include the generation of OpenMP TDGs for user-model-defined taskgraph regions. Not only the compiler generates an execute, but also, for the purpose of AMPERE, TDGs are generated in two formats: (1) a ".dot" file containing the information of the TDG structure and corresponding amalthea task/runnable connection, with one ".dot" file per parallel region, and (2) a ".cpp" file with the source code of the TDGs structure to be used by the runtime. This structure includes several parameters related to the tasks, like the static thread parameter, which adds the possibility of statically mapping OpenMP tasks to threads.

With the information of the ".dot" files, the amalthea model, and, optionally, an amalthea constraint, the profiler generates a TDG.json file containing all the TDGs and the information retrieved from the amalthea model. Figure 12 shows an example of a TDG.json file generated based on the configuration file in Figure 11.

Each TDG is annotated with the corresponding `amalthea_task_id`, extracted from the ".dot" files. The same process is used for each node of each TDG, annotating the nodes with the corresponding `runnable_id`. The amalthea model is used to annotate each TDG with the corresponding amalthea task deadline and period, and each node is annotated with `ticks` and its specialization (`spec` property), if the runnable can in fact be specialized (ARM/CPU by omission). Finally, one of two types of constraint can be specified to add information regarding the Amalthea tasks and their relationship. The property `latency_constraint` indicates that an Amalthea constraint of type "EventChainLatencyConstraint" exists and will provide the expected end-to-end deadline of the chain of Amalthea tasks and the name of the "EventChain" constraint, which is the second type of constraint. The "EventChain" contains the sequence of Tasks, thus providing the Amalthea task dependency chain. In terms of the TDG.json file, the end-to-end deadline is annotated as a first level property, as depicted in Figure 12, line 2, and the amalthea task chain is converted into "ins" and "outs" of the related Amalthea Tasks. The decision of using "ins" and "outs" to describe task chain order was for consistency purposes, similar to what is already done for the dependencies between nodes.

As expected, the generated TDG.json file does not contain results or metrics. This file is in fact a template for the profiling phase, which will be copied and populated with results while profiling the application in different system configurations, which depends on the target platform.

```
1 {
2   "end_to_end_deadline": 50000000
3   "TSR": [
4     {
5       "taskgraph_id": "2067960123",
6       "amalthea_task_id": "classification_tsr",
7       "ins": ["detection_tsr"],
8       "outs": ["output_tsr"],
9       "constraints": {
10        "period": -1,
11        "deadline": 33000000
12      },
13      "metadata": {
14        "variant": "cpu",
15      }
16      "nodes": {
17        "0": {
18          "runnable_id": "classification1",
19          "spec": "ARM",
20          "ins": [],
21          "outs": ["1"],
22          "metrics": {
23            "ticks": 183936
24          },
25          "results": [],
26        },
27        ...
28      },
29    },
30    ...
31  ]
32 }
```

Figure 12: TDG.json file generated based on the application, the amalthea model, and the "Latency Constraint" defined in Figure 11.

Next, the script is similar to the approach being used since WS3. It iterates over the possible system configurations, which are combinations built from the CPU, and accelerator, parameters. For each system configuration, the script first executes the system configuration commands to reconfigure both CPU and accelerator (the latter only for the code variants that use the accelerator), and then iterates over each executable a certain number of runs, defined in the configuration as *iterations*.

During the profiling of each configuration, the previously generated TDG.json file is copied and annotated with profiling results regarding execution time and performance counter measurements and regarding the current configuration (e.g. *code_version*, *CPU_config*, *variant_config*).

At the end of the profiling execution, the tool generates several TDG.json files, one per possible code version and system configuration. These files, together with an intermediate configuration file, are the inputs for the analysis process, which analyses each TDG file to extract metrics, and the optimization phase, which provides the best application and system configuration based on the extracted metrics.

An intermediate configuration file is used to control the following stages of the optimization flow. This intermediate configuration is a configuration "specialized" to the profiled environment. It includes the path to the generated TDG files, and their corresponding platform configuration, and inherits most of the information present in the profiler configuration file, such as variants information, the target platform information and the optimization section.

3.1.2 Optimization Phase

The optimization phase was divided in two branches. The first branch is an adaptation of the previously designed optimization loop that comprises a recompilation of the application with a statically defined mapping,

and selects one optimized configuration per optimization criterion. The second branch uses the Multi-criteria optimization depicted in Section 3.2, which provides a overall optimized configuration for all optimization criteria.

There are some main differences between these two optimization tools. While the first branch is focused on providing an optimized configuration for each optimization criterion (i.e. one for timing and another one for energy), the second branch provides a single configuration optimized for all the criteria. The second limitation is that, currently, the first branch optimization tool only deals with CPU-based applications, not considering accelerators, such as GPUs or FPGAs.

3.1.2.1 Optimization-Per-Criterion

This first optimization branch is represented in Figure 13 and consists of 4 phases: an exploration of the task to thread static mapping algorithms, a reprofiling, a timing analysis, and finally the selection of the best configurations.

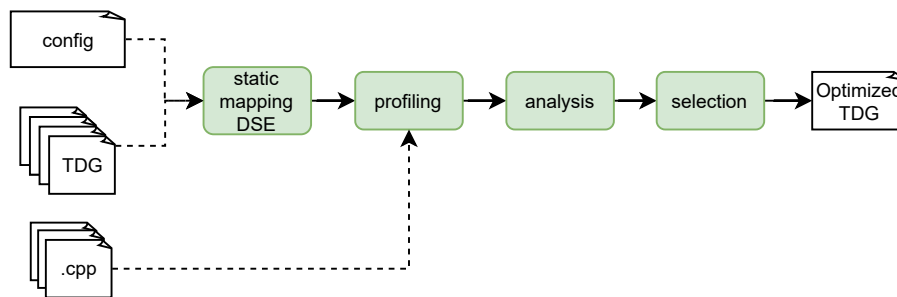


Figure 13: Optimization flow for the single criterion optimization.

The design-space exploration of static mapping is maintained similar to the version described in MS3 [3], and uses the same algorithms described in Section 2.3. As expected, the exploration will provide a static mapping for each TDG of the application, by annotating each OpenMP task in the TDG with the property `static_thread` with the corresponding thread id, depicted in the mapping provided in the previous steps. The objective of the reprofiling phase is to have measurements more specific to the static mapping defined for each TDG. Since each task is statically mapped to a thread, the obtained performance results are much more accurate, considering the expected execution of each task. The reprofiling phase uses the same profiler as previously described, but instead of using the "dynamic_mapping" compilation, the "static_mapping" compilation is now used (see Line 11 of Figure 11).

In this phase, each TDG has information about the static mapping and the system configuration in which it was executed previously. The reprofiling phase is then a loop iterating each TDG that:

1. recompiles the corresponding code variant with the provided static mapping;
2. reconfigures the system with the provided configuration;
3. measures the execution of the compiled version;
4. and redefines the TDG with the new results.

After reprofiling all the TDGs, the timing analysis and energy analysis tools are again executed, and once again obtain timing and power/energy metrics for each task and for each TDG.

The selection phase starts by first filtering the cases of TDG.json files that do not respect all the deadline constraints. If a single TDG does not respect the deadline, then that configuration (the TDG.json file) is not acceptable and so it is removed from the equation. Then, the framework will provide at the end a performance table, describing the response time (makespan) for each code version in each system configuration. Then, the selection method between all possible configurations is based on looking at the calculated makespans. The selected TDG.json file is the one providing the lowest cumulative value of makespans.

3.1.2.2 Multi-Criteria Optimization

Figure 14 depicts the work flow of this second branch. This branch is in fact a simple encapsulation for the optimization framework depicted in Section 3.2. The flow provides the optimization framework with the several TDG files, and receives as output, from the framework, a single TDG file in which the final configuration is defined. This TDG will contain information regarding:

- the static mapping per OpenMP task
- the specialization of OpenMP tasks
- the scheduling parameters per OpenMP region
- the selected CPU (and accelerator) frequency

The `converter` is responsible to take the newly annotated information and generate three files: a TDG.cpp code file, an amalthea model, and a configuration script. The TDG.cpp file will contain information regarding the static mapping defined per OpenMP task. The amalthea model is a copy of the referenced model, annotated with the scheduling parameters and runnable (OpenMP Task) specialization per amalthea task. This means that the developer can now regenerate the code from the Amalthea model, now with information regarding scheduling parameters and runnable specialization. The configuration script is generated based on the platform specification, initially defined in the profiler configuration, specializing the commands with the system configuration defined in the TDG (e.g. the CPU/accelerator frequency and number of threads).

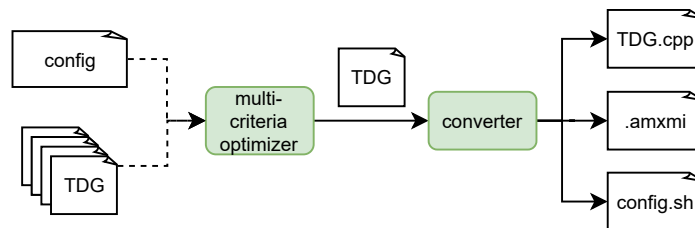


Figure 14: Optimization flow using the multi-criteria optimization framework.

3.2 Updates on Multi-criteria Optimization

The developed multi-criteria optimizer [19] aims to provide the optimal placement of tasks onto processing units (PUs) and the optimal power configuration of the hardware platform. The problem was formulated as a mixed integer quadratically-constrained programming (MIQCP) and solved with the Gurobi commercial tool [20].

The optimizer returns the system configuration in terms of:

- the placement of tasks onto the available PUs;
- for tasks for which both the software and a hardware-accelerated implementations are available, the decision about whether to use the software or the hardware implementation;
- the intermediate deadline of each task, summing up to the end-to-end one given by the timing constraints specifications;
- the operating frequency of each of the islands of the hardware platform.

Being this a complex problem, following the multi-criteria paradigm, the optimizer provides two criteria to determine the “optimal” configuration the designer can choose from depending on their needs. The first mode allows to obtain the deployment exhibiting the lowest average power consumption of the application running on the target platform, while guaranteeing the schedulability of the whole system. The second mode instead goes towards the system configuration that maximizes the robustness of the application, while keeping the power consumption within a given power budget. The robustness is defined in terms of *slack*, i.e., the difference

between the end-to-end deadline of a DAG and its response time, normalized with the deadline. Additionally, the two modes can be used in sequence: a first run of the optimizer can be carried out in the lowest power consumption mode, so that to obtain the minimum power value guaranteeing meeting the timing constraints; then, the very same problem can be solved again in the maximum robustness mode, using as power budget the value returned by the first optimization run.

The power consumption is computed using the model in [21]. The schedulability of the task set with respect to the timing constraints is assessed depending on the type of PU.

CPU test. When tasks are mapped to CPU cores, they are scheduled with the P-EDF algorithm with the intermediate deadline assigned by the solver. In order for the optimizer to determine whether a task with a given deadline is schedulable on a given CPU core, the MIQCP formulation includes constraints implementing the schedulability test. In particular, the test computes the total utilization of the core given the current mapping and checks whether it is less than 1, condition that guarantees the schedulability under P-EDF. However, given that the tasks are not independent but organized in DAGs, not all the tasks mapped to a core may run concurrently. Hence, the schedulability test is performed using the *unrelated* tasks, i.e., those tasks that are not in a dependency relationship. The test is thoroughly explained in [19].

Hardware accelerator test. The hardware accelerators (like GPUs, FPGAs) are modelled as non-preemptive FIFO queues. In order to determine whether an accelerated task is schedulable or not, it is required to compute its worst-case traversing time, which considers the worst queueing time the task may undergo before being served by the accelerator. This worst-case timing happens when all the other accelerated tasks are queued when the task arrives. More details can be found in [19].

The heuristic placement algorithms `TIF` and `bb-search` described in D2.4 [3] were only used in the preliminary phases of the development of the multi-criteria optimizer and are not included in the final release of the AMPERE project. As a matter of fact, their implementation does not provide the support for the placement of tasks over hardware accelerators.

Also, with respect to the version of the MIQCP optimizer described in D2.4 [3], we implemented the import of TDG files in the JSON format, required for the full integration in the AMPERE toolchain.

3.2.1 Validating the optimizer

The validation of the approach is threefold. A first evaluation consisted in the development of an alternative problem formulation identical to the MIQCP one but with the removal of the task intermediate deadlines from the variables, thus assigning to them fixed values. These values were obtained with a *deadline splitting* strategy, according to which the end-to-end DAG deadline was proportionally split over the tasks depending on their execution time. Such an approach led us to a simpler MILP formulation. This was used as the reference/baseline performance with respect to which our full MIQCP approach was evaluated, and its advantages assessed.

A second evaluation was carried out leveraging the simulation of the temporal behaviour of the tasks using PARTsim [21]. PARTSim¹ is a non-functional power-aware real-time systems simulator capable of simulating the schedule of tasks within a platform under various scheduling policies, along with the associated estimated power consumption. The simulator supports single-core and multi-core platforms, including systems with DVFS capabilities, with either partitioned or global fixed-priority and EDF-based schedulers. For DVFS-capable systems, the simulator can estimate both the power consumption and the thermal profile of the platform during the simulation. This is achieved through platform description files that can be automatically generated using its companion profiling tool, PARTProf [21], deployed on the target board one is willing to characterize. This tool can extrapolate a realistic model for the power and timing behavior of real-time tasks when executed under

¹More information is available at: <https://github.com/gabrielelara/PARTSim>.

DVFS on heterogeneous architectures by profiling the execution of a set of representative tasks on the target embedded platform.

Therefore, the platform major characteristics in terms of execution time and power consumption behavior, gathered from the target platform using PARTProf, can be configured both in the PARTSim simulator, and in our multi-criteria optimizer while the system is being optimized.

We used PARTSim to virtually deploy about 1700 randomly generated real-time DAG sets, each comprising one or multiple parallel independent DAGs. These scenarios have been optimized for a target ODROID-XU4 platform. The platform description provided for the simulator by the PARTProf tool provides PARTSim with the very same model used by each of our solvers to estimate the timing and power behavior of the systems being optimized. Combining this model with the static configuration provided by each of our solvers, we can accurately simulate the behavior of one or multiple DAGs as if executing on the target platform under the desired DVFS settings and task placement.

For each scenario where our optimizer found a solution, we ran the scenario in PARTsim, checking the simulated end-to-end response times and the average power consumption calculated for the simulation duration. As expected, no simulated scenario resulted in deadline misses in the simulations, and the simulated power consumption values matched the theoretical expectations, confirming the soundness of the approach.

The third evaluation involved experiments on a hardware platform using the AMPERE run-time, and it is detailed in Deliverable D4.4 [22].

We provide below a few further details about a campaign of optimization experiments we carried out to evaluate the effect of the system load on the optimal configuration found by the optimizer and to collect the run time taken by the solver to obtain the optimum value. Figure 15 shows the variation of average power consumption obtained by the MIQCP and MILP formulations as a function of the overall system load. It is evident and expected that the power increases with the load, because the more the computations, the more the computing power. Furthermore, the MIQCP is generally able to find configurations exhibiting a lower power consumption than the MILP, due to the fact that having the task intermediate deadlines as variables enlarges the solution space, potentially containing better solutions.

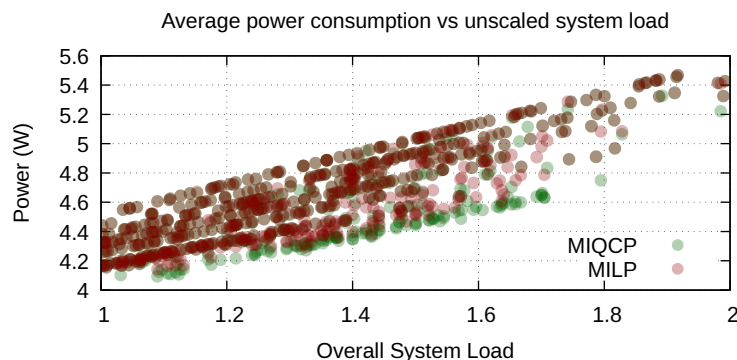


Figure 15: Optimal power values as a function of the unscaled system load (X axis).

Figure 16 depicts the power consumption values obtained by both the MIQCP and the MILP and the run time taken by the corresponding solver to produce that solution. The MILP generally requires less execution time because the solution space to be explored is smaller than that of the MICQP.

More details about the performed simulations can be found in [19].

Figure 16 also shows that there are applications for which the MIQCP optimizer requires more than 24h to find the optimal configuration. For such very complex cases, depending on how much time is allocated to the design stage of the development process, this run time might not be acceptable. Hence, the AMPERE toolchain makes available the heuristic placement algorithms described in Section 2.3.

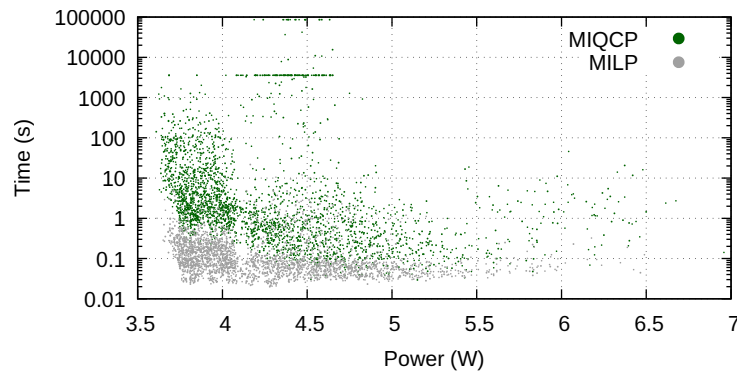


Figure 16: Solving time (Y axis) obtained by the two optimizers (different data series) vs the achieved optimized average power (X axis).

3.2.2 Optimizer extension to support OpenMP

The model presented in [19] holds for tasks organized in DAGs. We extended the optimizer to support OpenMP tasks as used in the AMPERE project. Basically, each DAG represents a task and its nodes are runnables, executed in parallel but still respecting the data dependencies. Each task constitutes an OpenMP parallel region, which is assumed not to suffer from interference made by other OpenMP parallel regions possibly present in the system, which is possible using `SCHED_DEADLINE`, for example (see below). Runnables in OpenMP are served by *worker threads* in a FIFO fashion. The worker threads are in turn organized in OpenMP regions, meaning that a thread of a region only serves runnables of that region.

The model extension on which the optimizer is based mandates that for every OpenMP parallel region, there exists a worker thread for each CPU core. The optimizer is expected to provide the static mapping between runnables and worker threads, which essentially means pinning a runnable to be served on a specific CPU core. Furthermore, to guarantee the isolation between worker threads and between parallel regions, each worker thread is assigned to a `SCHED_DEADLINE` reservation. Hence, the optimizer produces as output the parameters of each reservation: the budget, the period and the deadline.

OpenMP runnables can still be mapped on the accelerators, but in case more than one OpenMP task is mapped to the same accelerator, then we need to account for cross-interferences among tasks from different parallel regions, which is supported anyway by our optimizer.

3.2.3 Extension to support task chains

The use cases of the AMPERE project are characterized by the presence of several tasks exchanging data, organized in chains, of which the first is periodic. Some of the other tasks in the chain may not be periodic, but activated whenever their input data is ready.

The model in [19] only considers periodic tasks; hence, an adaptation was required. Since a data-driven task is activated only when the data produced by a periodic task is available, the extension we developed *merges* the data-driven task with its periodic activator, thus creating a bigger periodic task. The operation can be repeated iteratively for all the subsequent data-driven tasks. Such an approach is enabled by the fact that at least the first task of a chain is periodic.

3.3 Evaluation on PCC Use Case

The developed optimizer was tested on the PCC use case to obtain the optimal system configuration for the NVIDIA Jetson AGX Xavier board, made of 8 CPU cores and a GPU, and for the Xilinx UltraScale+ ZCU102. In

what follows, we describe the optimization for the NVIDIA platform. Experimental runs for both platforms are reported in Deliverable D4.4 [22].

The PCC model is composed of several tasks exchanging data, organized in a chain. Each of these tasks executes a set of runnables with OpenMP, so they are parallelized. Each task has its own deadline, acting as an end-to-end deadline for its runnables. The whole task chain has an end-to-end deadline as well, which refers to the complete processing of data of the PCC use case.

The model was provided to the optimizer in the form of a JSON file, produced by the profiling and analysis tools in Figure 10, which in turn performed several executions of the code auto-generated by the SLG code generator, starting from the AMALTHEA model. The JSON file used as input to the optimizer was containing the measurements of the execution times of each task for each PU type (CPU or GPU) for various available frequencies, enriched with the performance counter measurements, that were mapped to power consumption figures by the energy modeling tool (see Section 3.1).

Looking at the model, most of the tasks involved in the chain have just one or two runnables, so the OpenMP parallelization was applied only to the task named `classification_tsr` made of 10 runnables. In order to deal with the use case, the extension described in Section 3.2.2 was used. The results of the optimization are described in the next paragraphs.

3.3.1 Minimum power optimization

The optimizer was run in minimum power mode with the PCC use case. The resulting system configuration is showed in Figure 17. Each node is a runnable, for which the figure reports the execution time, the intermediate deadline and the corresponding requested computational bandwidth. Runnables are grouped into tasks, highlighted with different colours. The number of tasks in the picture is smaller than that of the use case because of the task merge operation described in 3.2.3. The dependency between nodes are represented by arrows.

The picture shows the PUs of the hardware platform grouped by island. For each island, the operating frequency corresponding to the requested optimal system configuration is reported. The placement of runnables onto PUs is graphically showed by the placement of the corresponding node into the rectangle representing the PU. For each PU, its total reserved computational bandwidth as a consequence of the mapping is reported. Also, the picture describes the pinning of OpenMP runnables onto worker threads. Each worker thread is represented by a coloured rectangle placed inside a PU rectangle, meaning that the worker thread runs on that specific PU. Inside each worker thread rectangle, there are the runnables pinned to that specific worker thread. For each worker thread, the ID of the corresponding OpenMP parallel region and the parameters of its SCHED_DEADLINE reservation are reported.

As it is evident from the figure, all the runnables were mapped to CPU cores, and the estimated average power consumption is 1.586W. The reason behind this choice taken by the optimizer is the high power consumption experienced when activating the GPU. Indeed, mapping all the runnables onto CPU cores guarantees the lowest power consumption. Also, both the end-to-end and task deadlines are likely to be not so tight. The schedulability is guaranteed using only the slower software implementation of the tasks; hence, even if the GPU tasks take less time to execute w.r.t. the CPU ones, they are not needed. Being the optimization metric of this experiment the minimum power consumption, the solver is discouraged to use the faster hardware-accelerated tasks to keep the power consumption low.

3.3.2 Maximum robustness optimization

Given the considerations of Section 3.3.1, the solver does not leverage the GPU tasks because of the target of the optimization, i.e., obtaining the lowest power consumption. However, having faster hardware-accelerated implementations of the very same tasks may be of help to obtain a more robust system, at the cost of a slightly increased additional power consumption. Indeed, the minimum power configuration exhibits a nearly 0%

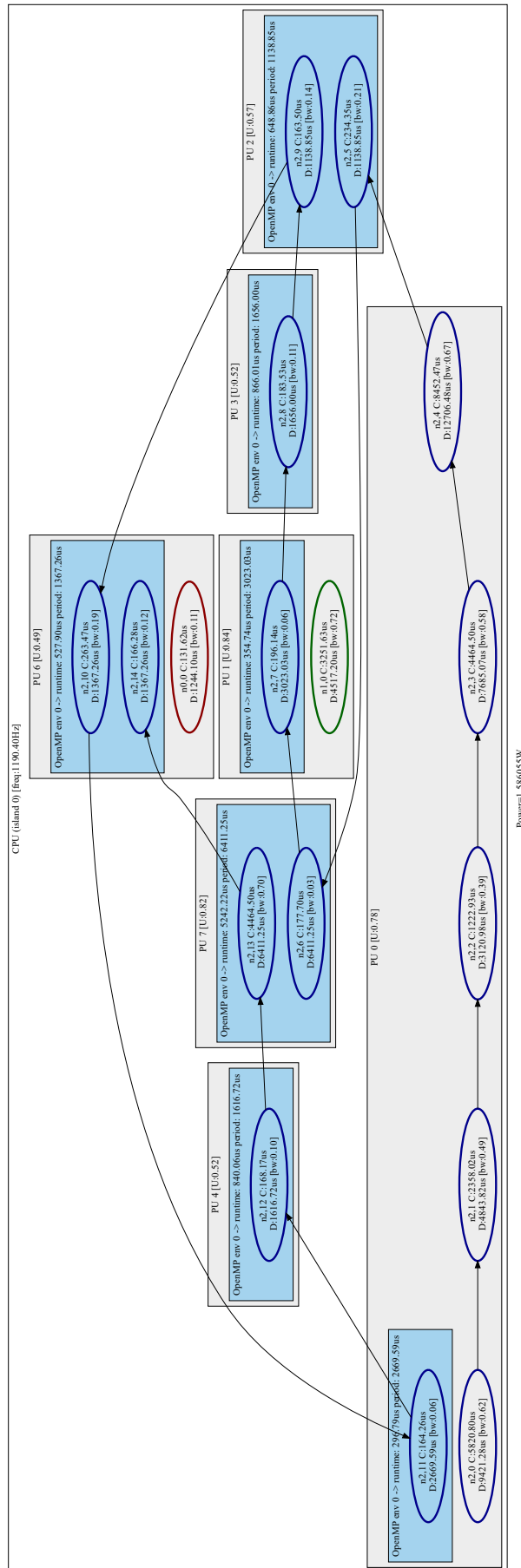


Figure 17: Optimal PCC system configuration to achieve the minimum average power consumption.

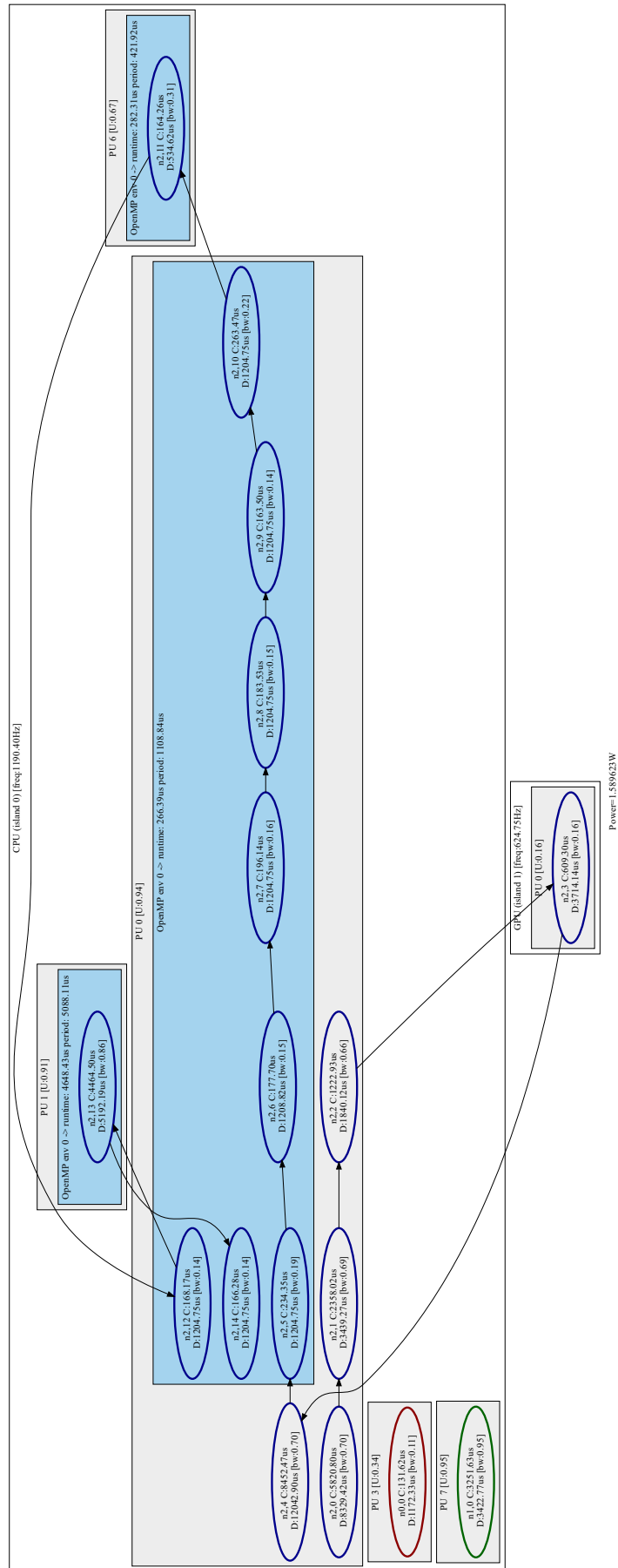


Figure 18: Optimal PCC system configuration to achieve the maximum robustness.

slack. Hence, we run the optimizer in the maximum robustness mode, so as to investigate further interesting trade-offs between power consumption and robustness of the resulting design.

We took as power budget that returned by the optimization in Section 3.3.1. In order to encourage the optimizer to deploy hardware-accelerated tasks, we decided to increase the power budget by just 5%, potentially paving the way for a more robust configuration. The result is showed in Figure 18. The optimizer mapped one runnable on the GPU, as its faster implementation is required to achieve a 3% slack. The estimated power consumption of such a configuration is 1.589W, only 0.22% bigger than the minimum power of the first optimization run. Thus, the increase can be simply considered as a measurement noise. Nevertheless, we gain in terms of enhanced robustness of the model w.r.t. unmodelled/unforeseen spikes in execution time estimations.

3.4 Evaluation on ODAS Use Case

We considered the railway use case of the AMPERE project, named ODAS, with the aim of obtaining the optimal system configuration for the NVIDIA Jetson AGX Xavier board.

The ODAS model is composed of six independent tasks made of several runnables, most of which are replicated to guarantee a certain degree of resiliency mandated by the use case. As an example, the UKF component contains almost 250 runnables. Replicas can be executed in parallel; hence, we considered parallelizing their execution with OpenMP, with the additional constraint that the optimizer will not place the replicas of the same runnable on the same PU.

A representation of the model fed to the optimizer is depicted in Figure 19, where each task is represented as an independent periodic DAG. It is clear to see that the application is characterized by a massive runnable replication for resiliency, causing a gigantic number of dependencies between runnables represented by arrows in the picture.

Because of the huge size of the model and the complexity of the optimizer mode handling the isolation between OpenMP parallel regions, we disabled the per-region reservation mechanism in order to find the optimal placement. Even with this simplification, it was not possible to optimize the model, due to the excessively high number of variables and constraints to be generated when preparing the optimization LP program.

A possible workaround for this use-case, is the one to transform the model into a reduced-size one, by recurring to aggregation of topologically compatible elements. This way, we obtain a model that contains the same work to be done, with some tiny runnables grouped into bigger ones, keeping the same timing, topological/dependency and reliability constraints as in the original model. Such an approach makes sense, because in the case of the ODAS use-case, we have about 60 runnables in the UKF DAG of the model, representing as many Kalman filters that need to track many possible subjects moving through the scene. These computations need to be carried out reliably, thus multiplying by 3x the number of nodes in the model to optimize, considering the replicated nodes. However, the number of CPUs available on the board is only 8 (plus the GPU), therefore it makes sense to aggregate several of these Kalman filter operations, when having no reciprocal anti-affinity constraints, into a single sequential runnable. For example, even with a reduction of a factor of 4, we would end-up with $60 \cdot 3/4 = 45$ bigger runnables (equivalent to the 180 original tiny ones) that can still be spread quite well across the available 8 CPUs, and are way more manageable from an optimization perspective. A preliminary feature performing such model reduction/transformation has been incorporated into our MIQCP-based multi-criteria optimization tool, obtaining for example the model in Figure 20.

This transformed reduced-size model was optimized for minimum power in about 8 hours of computations, obtaining the optimized placement reported in Figure 21.

It is worth to note that the optimizer used solely the CPU cores of the platform, leaving the GPU unused (visible in the bottom-right corner of Figure 21), for three main reasons: the high workload of the application, challenging the overall schedulability; the presence of numerous replicas, constraining the optimizer to map replicated runnables to different PUs; and finally the higher power consumption involved when deploying kernels on the GPU, compared to the execution of equivalent code on the CPU(s) of the board. The CPU frequency

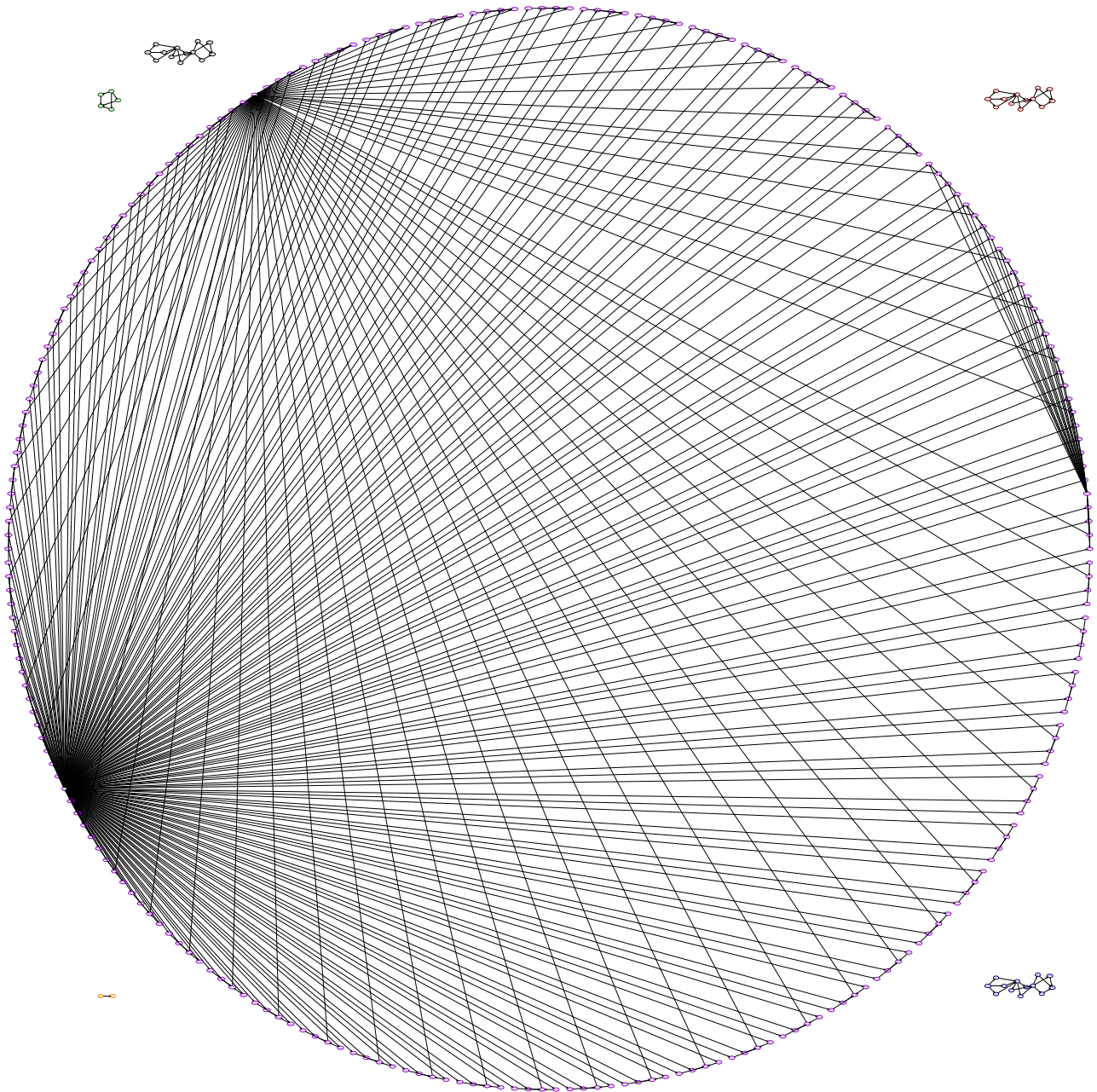


Figure 19: ODAS model fed to the optimizer.

has been set to the intermediate one, 1190.4 MHz. This optimized configuration results into an estimated average power consumption of 3.51W. Executing repeatedly the resulting configuration on the NVIDIA Jetson AGX platform using the tools described in D4.4 [22] results in no missed deadlines. Table 6 shows average and maximum registered response times for each task when executed on the target platform.

3.5 Evaluation on ODAS Use Case Using Mapping Heuristics

As an alternative to the aggregation solution, we also evaluated the ODAS use case with an approach that explores OpenMP task-to-thread mappings using heuristics, suitable for the cases where the multi-criteria optimizer is not able to scale. This approach is able to provide two optimized configurations, one for response time and another for energy, based on the first optimization branch of section 3.1.

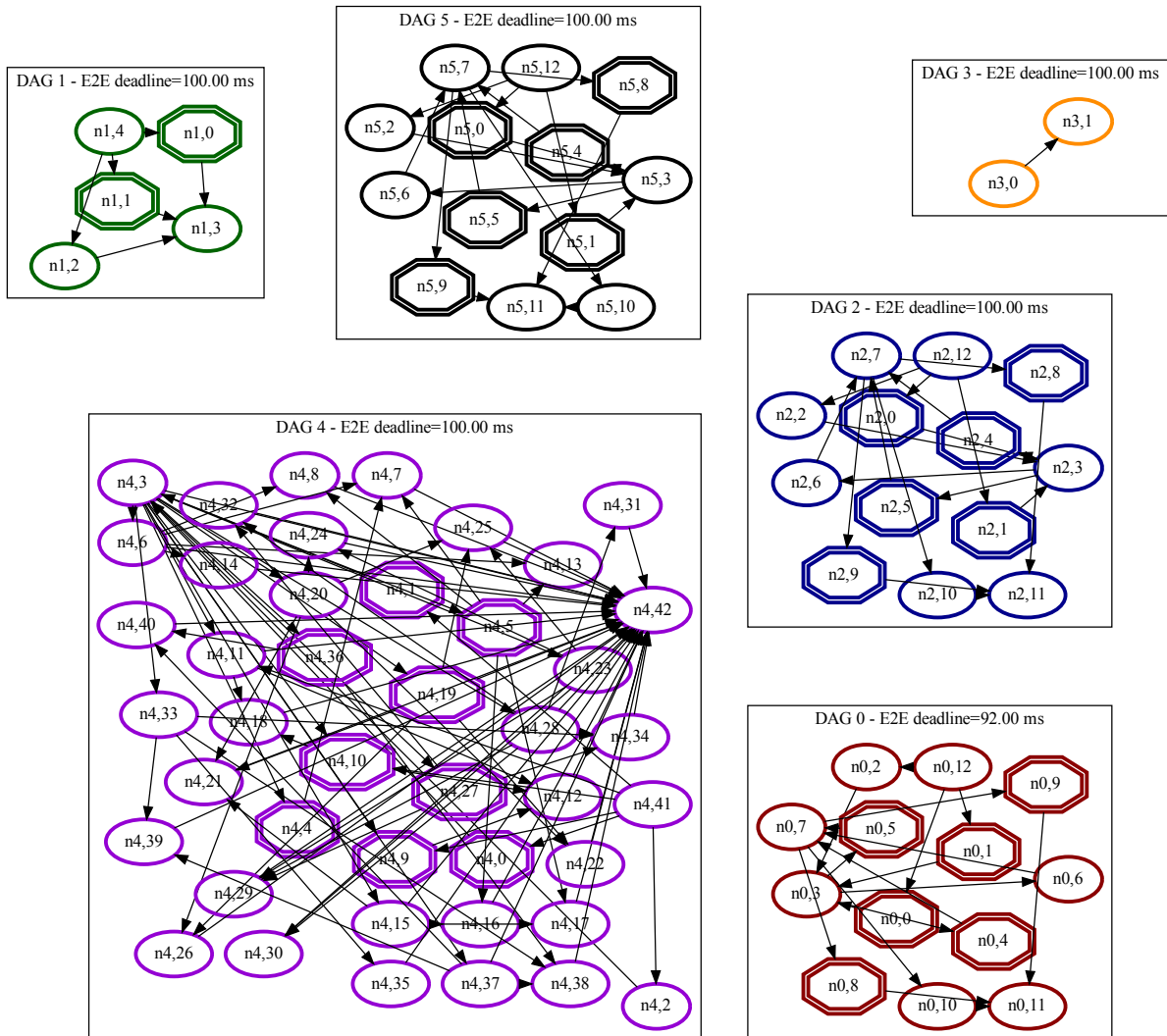


Figure 20: Transformed ODAS model as reduced by the optimizer via aggregation of small runnables (and still equivalent to the full model in Figure 19), before starting the actual MIQCP-based optimization.

The ODAS use case is divided into six Amalthea tasks: RADAR Preprocessing, LiDAR Preprocessing, OffLoaded CAMERA Preprocessing, Sensor Fusion, UKFs and CCM. The tasks do not contain many exploitable parallelization, but they do use replication. Figure 22a shows the TDG of the LiDAR Preprocessing task. In this example we can see that the parallelizable code is in fact the replicas of specific runnables, and a "synchronization" node exist between replications. Tasks RADAR Preprocessing and Sensor Fusion contain a TDG with a very similar structure, while the TDGs of CCM and Offloaded CAMERA Preprocessing are in fact very small (between two to four runnables in these tasks). The only task taking advantage of parallelism is the UKFs task, which contains 243 nodes, and the main TDG structure of this task is represented in Figure 22b. This task uses 60 UKFs, running in parallel, with each UKF having three replicas.

This description is important to understand the results of the mapping exploration. We executed the multi-criteria optimization flow using four representative mapping algorithms:

- MTET-MET: Minimum Total Execution Time allocation heuristic with a Minimum Execution Time dispatching heuristic;
- MTET-FIFO: Minimum Total Execution Time allocation heuristic with a First-In-First-Out queue;

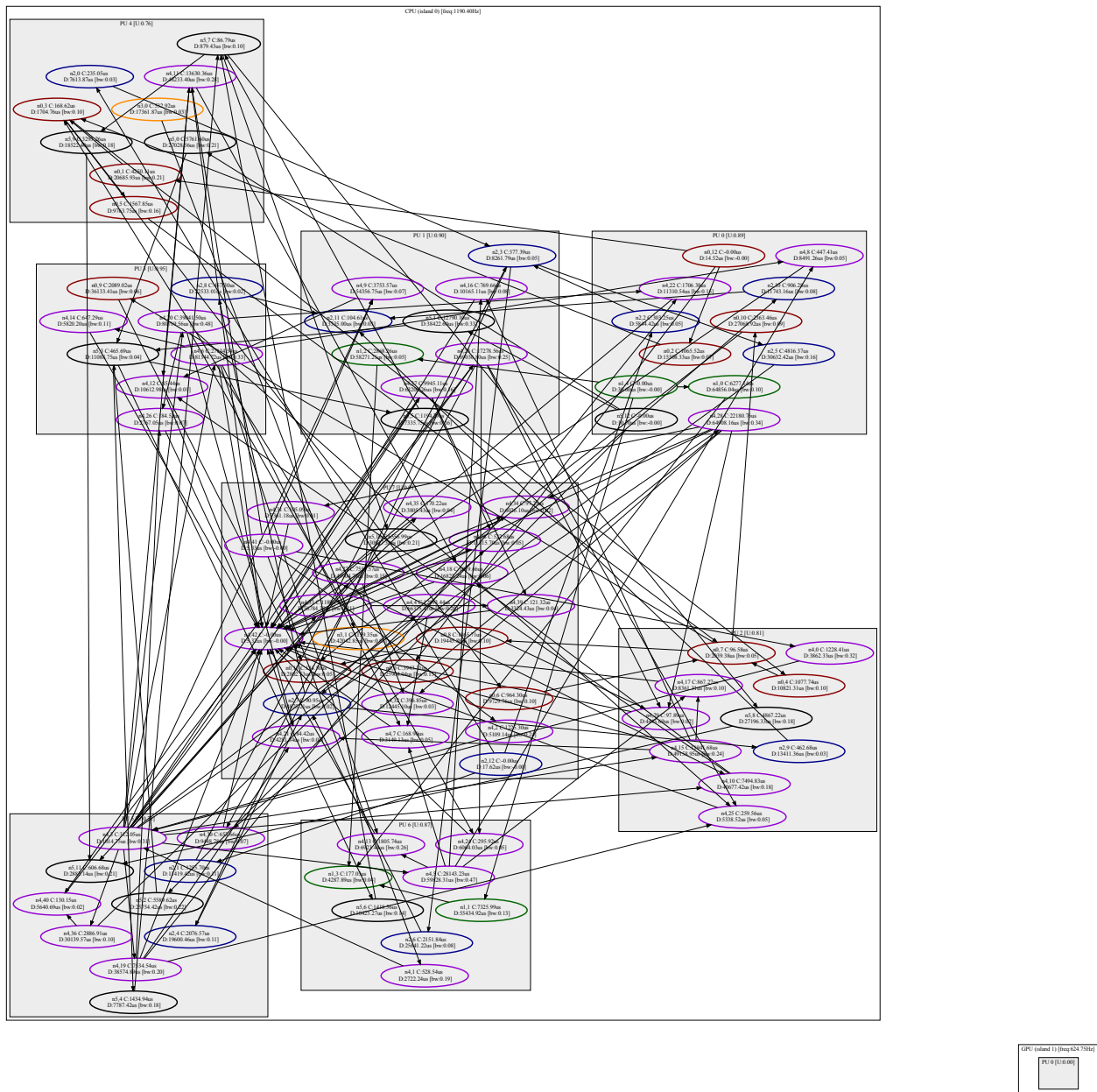


Figure 21: Optimal placement found by the MIQCP-based optimizer for the transformed reduced-size ODAS model in Figure 20.

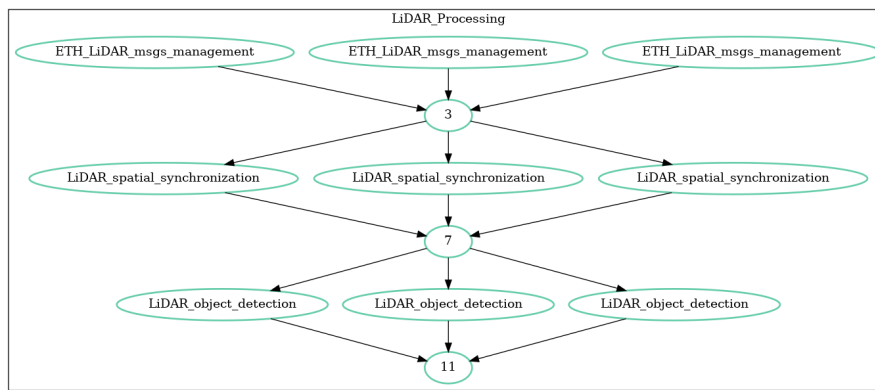
- BFS-FIFO: traditional BFS algorithm of LLVM’s runtime OpenMP;
- SEQR: sequential algorithm, only replicas in different threads.

The two latter algorithms are used for comparison purposes, one to compare with the original dynamic approach (BFS) and the other to compare with a sequential execution, where only replicas are in fact executed in different threads.

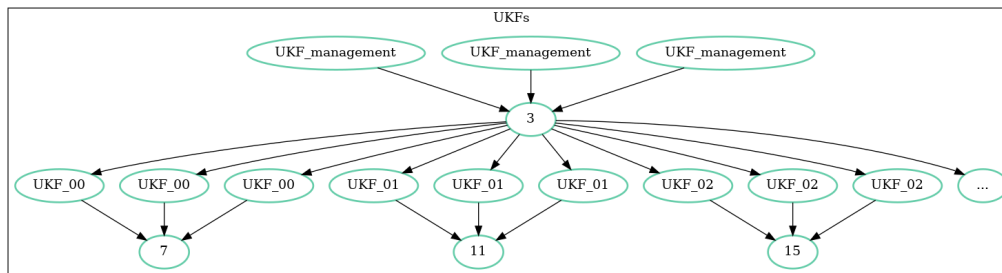
The exploration of the mapping algorithms only had impact in the UKFs Amalthea task, while the others did not take advantage of the heuristic-based mapping algorithms. This is mainly due to the other tasks having almost no parallel executions. Figure 23 shows the results for the UKFs task and for the LiDAR Processing task, where the latter is representative to the behavior of the other tasks. Observe that the LiDAR Processing shows no potential parallelism, and this can be seen when comparing any mapping algorithm to the sequential

Task	Deadline [ms]	Response Time [ms]	
		Average	Maximum
0	92	9.738	40.714
1	100	7.278	48.500
2	100	8.338	35.165
3	100	1.932	8.493
4	100	63.452	79.271
5	100	21.529	48.970

Table 6: Average and maximum response times per task in the ODAS use-case when executed on the NVIDIA Jetson AGX platform.



(a) LiDAR Preprocessing, each runnable with tree replicas



(b) UKFs (using 60 UKFs, each one with three replicas)

Figure 22: Two representative TDGs of Amalthea tasks modelled in ODAS use case.

representation of the code (where only replicas are executed in different threads). Henceforth, the mapping optimization is focused essentially in the UKFs task, which shows variability in Figure 23 between the algorithms.

The multi-criteria optimization was done with the exploration of mapping algorithms and with the exploration of different system configurations. In this case, the exploration was done with different CPU frequencies, more specifically: 729.60MHz, 1.190GHz and 2.265GHz. Figure 24 provides the results of the mapping exploration for the different frequencies, over the different Amalthea tasks, where the WCET was used for the mapping exploration. Considering that we are actually specifying a static task-to-thread mapping, we will have more guarantees on the predictable execution time. The results of the Amalthea Tasks are stack to show the complete end-to-end execution time/energy consumption accumulated from all the tasks. Figure 24a shows the results for execution time. Here we can see improvements on performance when increasing the frequencies. From 729.60MHz to 1.190GHz we have an increase of performance of 2.19x, and increasing to 2.265GHz we further improve with more 3.02x (compared to 1.190GHz).

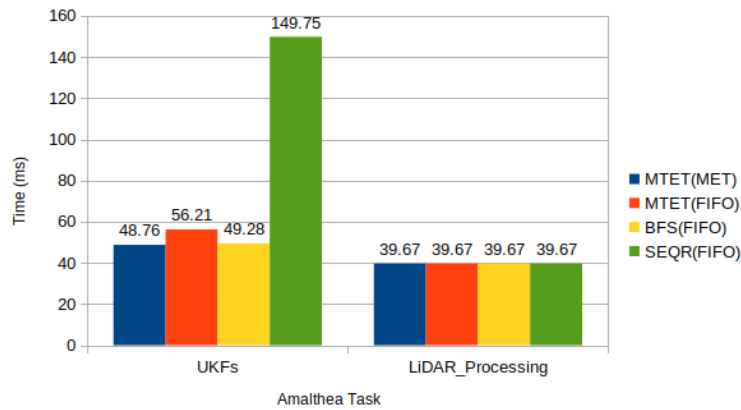


Figure 23: Mapping exploration results for two representatives tasks: "UKFs" and "LiDAR Preprocessing", when executing in a CPU frequency of 1190.40MHz, with eight OpenMP threads.

In Figure 24b we see improvements regarding the use of a CPU frequency at 1.190GHZ, but we start expending more energy when considering higher frequencies. The lowest frequency shows that, despite being expected to consume less energy, the tasks to be performed takes longer to execute, expending more energy to complete the tasks. When increasing the frequency from 729.60MHz to 1.190GHZ we improve energy consumption by 2.12x, but when we increase to 2.265GHz we have a performance loss of 0.73x.

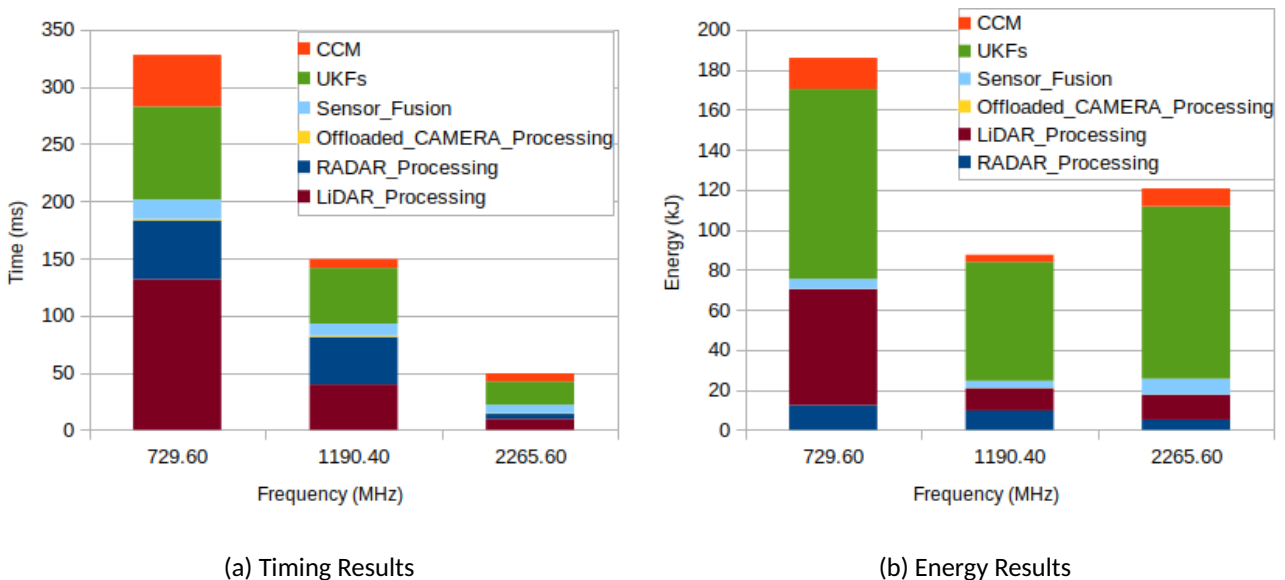


Figure 24: Timing and energy performance results from the Multi-criteria optimization flow, for different CPU frequencies, using eight OpenMP threads.

Since the optimizer selects one configuration for each optimization criterion, for this use case the optimizer selected specific static task-to-threads mappings (one for each task) and the following system configurations:

- For time predictability: a CPU frequency of 2.265GHz;
- For lower energy consumption: a CPU frequency of 1.190GHz.

4 Conclusions

This document presented the updated analysis and tools for multi-criteria optimization for the AMPERE framework, together with their evaluation on the project use cases.

The document described the evolutions of the mechanisms identified in Deliverable D3.3 [4] for analysis of energy-efficiency, predictable execution, and software resiliency techniques, as well as the final framework and approach for the multi-criteria optimization described in Deliverable D2.4 [3], and integrated into the AMPERE ecosystem within WP6 [2].

The document then presents the evaluation of the multi-criteria optimization framework in the use cases of WP1 [5].

5 Acronyms and Abbreviations

API	Application Programming Interface
CCM	Collision Checker Module
CMOS	Complementary Metal-Oxide Semiconductor
COTS	Commercial Off-The-Shelf
CPU	Central Processing Unit
CTA	Compute Thread Array
CUPTI	CUDA Profiling Tools Interface
D	Deliverable
DAG	Direct Acyclic Graph
DPM	Dynamic Power Management
DPR	Dynamic Partial Reconfiguration
DR	Dynamic Regulator
DSML	Domain Specific Modeling Language
DVFS	Dynamic Voltage and Frequency Scaling
FPGA	Field-Programmable Gate Array
GPU	Graphics Processing Unit
HDL	Hardware Description Language
HPC	High-Performance Computing
I ² C	Inter Integrated Circuit
ISA	Instruction Set Architecture
LLC	Last Level Cache
LLS	Linear Least Squares
LUT	Lookup Table
MAPE	Mean Absolute Percentage Error
MDE	Model-Driven Engineering
MET	Minimum Execution Time
MILP	Mixed Integer Linear Programming
MNTP	Minimum Number of Task Parts
MRIT	Most Recent Idle Time
MRT	Maximum Response Time
MS	Milestone
MTET	Minimum Total Execution Time
MTRT	Maximum Total Response Time
NNLS	Non-Negative Least Squares
NT	Next Thread
ODAS	Obstacle Detection Avoidance System
OS	Operating System
PCC	Pearson Correlation Coefficient
PMC	Performance Monitoring Counter

PMU	Performance Monitoring Unit
PPM	Parallel Programming Model
RR	Reconfigurable Region
SFI	Software Fault Injection
SIMD	Single Instruction Multiple Data
SLG	Synthetic Load Generator
SM	Streaming Multiprocessor
SoC	System-on-a-Chip
T	Task
TDG	Task Dependency Graph
WCET	Worst Case Execution Time
WP	Work Package

6 References

- [1] AMPERE, “Grant Agreement,” 2018.
- [2] —, “Deliverable D6.4, Final Release of the Ampere Ecosystem,” June 2023.
- [3] —, “Deliverable D2.4, Multi-criteria optimization model transformation,” September 2022.
- [4] —, “Deliverable D3.3, Energy optimisation framework, predictable execution models and analysis, and Software resilient techniques,” September 2022.
- [5] —, “Deliverable D1.1, System models requirement and use case selection,” 2020.
- [6] —, “Deliverable D2.3, Programming model extensions and the multi-criteria performance-aware component,” September 2022.
- [7] —, “Deliverable D4.3, Integrated run-time energy support, and predictability, segregation and resilience mechanisms,” September 2021.
- [8] S. Mazzola, T. Benz, B. Forsberg, and L. Benini, “A data-driven approach to lightweight dvfs-aware counter-based power modeling for heterogeneous platforms,” in *International Conference on Embedded Computer Systems*. Springer, 2022, pp. 346–361.
- [9] AMPERE, “Deliverable D3.1, Multi-criteria optimization requirements,” September 2020.
- [10] —, “Deliverable D3.2, Single-criterion energy optimisation framework, predictable execution models and software resilient techniques,” July 2021.
- [11] NVIDIA Corporation, “Jetson AGX Xavier developer kit,” 2018. [Online]. Available: <https://developer.nvidia.com/embedded/jetson-agx-xavier-developer-kit>
- [12] AMPERE, “Deliverable D5.1, Reference parallel heterogeneous hardware selection,” 2020.
- [13] M. S. Gharajeh, S. Royuela, L. M. Pinho, T. Carvalho, and E. Quiñones, “Heuristic-based task-to-thread mapping in multi-core processors,” in *2022 IEEE 27th International Conference on Emerging Technologies and Factory Automation (ETFA)*. IEEE, 2022, pp. 1–4.
- [14] C. Lattner and V. Adve, “Llvm: A compilation framework for lifelong program analysis & transformation,” in *Proceedings of the International symposium on code generation and optimization*, 2004.
- [15] T. Carvalho, L. M. Pinho, M. Samadi, S. Royuela, A. Munera, and E. Quiñones, “Framework for the analysis and configuration of real-time openmp applications,” in *2023 IEEE International Conference on Industrial Informatics*. IEEE, 2023.
- [16] A. Melani, M. Serrano, M. Bertogna, I. Cerutti, E. Quinones, and G. Buttazzo, “A static scheduling approach to enable safety-critical openmp applications,” in *Proceedings of the 22nd Asia and South Pacific Design Automation Conference*, 2017.
- [17] A. Marongiu, G. Tagliavini, and E. Quiñones, “Openmp runtime,” in *High Performance Embedded Computing*. River Publishers, 2022, pp. 145–172.
- [18] B. S. Center, “Bsc extended llvm 16.0,” url=<http://gitlab.bsc.es/ampere-sw/wp2/llvm>, 2023.
- [19] T. Cucinotta, A. Amory, G. Ara, F. Paladino, and M. D. Natale, “Multi-criteria optimization of real-time DAGs on heterogeneous platforms under p-EDF,” *ACM Transactions on Embedded Computing Systems*, Apr. 2023. [Online]. Available: <https://doi.org/10.1145%2F3592609>
- [20] Gurobi Optimization, LLC, “Gurobi.” [Online]. Available: <https://www.gurobi.com/>
- [21] G. Ara, T. Cucinotta, and A. Mascitti, “Simulating Execution Time and Power Consumption of Real-Time Tasks on Embedded Platforms,” in *Proceedings of the 37th ACM/SIGAPP Symposium on Applied Computing*, ser. SAC '22. New York, NY, USA: Association for Computing Machinery, 2022, pp. 491–500. [Online]. Available: <https://doi.org/10.1145/3477314.3507030>
- [22] AMPERE, “Deliverable D4.4, D4.4 Evaluation of run-times,” June 2023.