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

# D3.2 Single-Criterion Energy-Optimization Framework, Predictable Execution Models, and Software Resilient Techniques

Version 1.0

## Documentation Information

| | |
|---|---|
| **Contract Number** | 871669 |
| **Project Website** | www.ampere-euproject.eu |
| **Contratual Deadline** | 30.03.2021 |
| **Dissemination Level** | [PU] |
| **Nature** | R |
| **Author** | Björn Forsberg and Thomas Benz (ETHZ) |
| **Contributors** | Alessandro Biondi, Alexandre Amory, Biruk Seyoum, Francesco Restuccia, Gabriele Ara and Tommaso Cucinotta (SSSA)<br>Tiago De Carvalho and Miguel Pinho (ISEP)<br>Sara Royuela (BSC)<br>Viola Sorrentino (TRT)<br>Gianluca Mando (THALIT)<br>Michael Pressler (BOS) |
| **Reviewer** | Alexandre Amory (SSSA) |
| **Keywords** | Single-Criterion Optimization, Energy, Timing Predictability, Resilient Software, DVFS, FPGA, reconfigurable |

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

# Change Log

| Version | Description Change |
|---|---|
| **V0.1** | Initial version |
| **V0.2** | Addressed comments after review |
| **V1.0** | Release version |

# Table of Contents

# 1. Executive Summary

This Deliverable summarizes the progress of Work Package 3 up until Milestone 2, and is a report describing the energy-efficiency approaches, the predictable execution models and software resilient solutions for parallel execution, without considering the other constraints.

The energy-efficiency approaches are based on the energy-models defined in Task 3.1, incorporating the energy consumption of the major components present in selected parallel heterogeneous architectures, as well as the available power management knobs. It describes and evaluates the ability of these techniques to estimate their impact on the overall energy-performance trade-off achievable on the target architecture. We show how the techniques are successfully used to determine the most efficient operation mode for energy-efficiency, at low-overhead by model-based evaluation of profiling data, as well as strategies for optimization that utilize dynamic voltage-frequency scaling and optimize for the use of the most efficient components in the system, with a special focus on floating-point operations.

The predictable execution models present techniques for joint consideration of process allocation, both at the host and accelerator, and effects of memory and on-chip network resources. We describe how memory access and communication costs have impact on the time-predictability of parallel applications, and identify the execution models to consider, with a focus on Logical Execution Time and communication middleware such as ROS. We present the ongoing development of the relevant timing analysis approach, based on real-time schedulability analysis techniques to estimate upper bounds on response times and derive meaningful statistics on the response-time distribution. We study the execution behavior of FPGA accelerators for Deep Neural Networks to the purpose of performing timing analysis. We also address design optimizations of PGA design to handle reconfigurable accelerators under timing constraints.

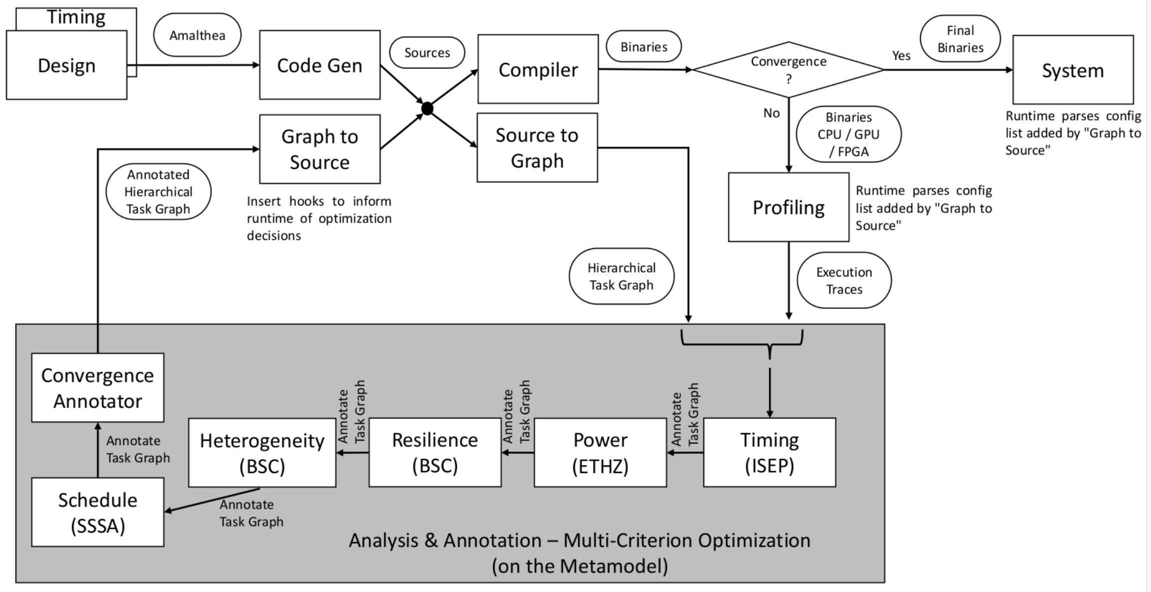
The software resiliency solutions provide modeling and programmer's techniques to protect the most vulnerable components of the selected parallel architectures, from the application to architecture, resulting into two orthogonal and coordinated software protection solutions that are integrated at all layers of the AMPERE system stack. One of the presented solutions is fully integrated in the synthesis tool as a specific optimization component, while the other is a programmer-in-the-loop approach for increased flexibility. Together they provide two complementary approaches for fault-tolerance, which is the focus of this milestone.

All goals set out to be achieved by the end of Milestone 2 have been achieved, and the alignment of the different analysis and optimization components provide an excellent starting point for the coming work in Milestone 3, in which the techniques presented in this report are integrated to construct the multi-criteria optimization framework.

# 2. Introduction

This report describes the single-criterion optimization for energy-efficiency, predictable execution, and software resiliency techniques for the AMPERE framework. Due to the single-criterion optimization scope, these optimization techniques are presented in isolation, but will be integrated and co-evaluated as part of the next milestone (MS3). Several steps have already been taken in preparation of this, and will be presented in this deliverable. The tasks within Work Package 3 interact closely with components developed in Work Packages 1, 2, 4, and 5. As part of Milestone 2, significant effort has been investigated into aligning the work done in the different work packages to simplify later integration, and aid the design also in the single-criterion optimization phase. An overview of the interactions of WP3 (shown in dark gray) is presented in Figure 1.

Figure 1: An overview of the interaction of WP3 (gray) with the other components within AMPERE.


Timing
Design
Amalthea
Code Gen
Sources
Compiler
Binaries
Convergence ?
Yes
Final Binaries
System
Graph to Source
Source to Graph
No
Binaries CPU / GPU / FPGA
Runtime parses config list added by "Graph to Source"
Annotated Hierarchical Task Graph
Insert hooks to inform runtime of optimization decisions
Profiling
Runtime parses config list added by "Graph to Source"
Hierarchical Task Graph
Execution Traces
Convergence Annotator
Annotate Task Graph
Heterogeneity (BSC)
Resilience (BSC)
Power (ETHZ)
Timing (ISEP)
Schedule (SSSA)
Analysis & Annotation – Multi-Criterion Optimization (on the Metamodel)

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 heterogenouity. The DSML and the requirement annotations for DSML in the context of AMPERE are further described in D1.3. The primary DSML used in AMPERE is AMALTHEA[1], which is representative also for AUTOSAR.

Once the system has been modeled in (or converted to) AMALTHEA, the code generator described in D2.2 generates the corresponding source code, including annotations for the OpenMP parallel programming model (PPM), describing the dependencies and parallelism exposed in the modeled system. This is represented by the *Code Gen* step in the figure. The source code is passed to the OpenMP compiler, as described in D2.2 for compilation into binaries. Importantly, the OpenMP compiler has been extended to not only produce the binary images themselves, but also structured information about the system, which is used for the optimization

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

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), as described in D6.2. The TDG will recurr in the following sections of this deliverable, linking the contributions together.

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. This information is described in detail in D6.2, but includes the properties (e.g., non-functional requirements) annotated in the DSML towards which the system should be optimized. Additionally, the TDG contains profiling information from the generated system, through the execution of the Extrae [1] profiling infrastructure, described in D6.3. 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 optimization phase, as developed in WP3 and reported on in this deliverable, is highlighted in the gray box in Figure 1. It consists of multiple components developed in parallel: the timing analysis and optimization, resiliency optimization techniques, energy optimization, and heterogeneous scheduling.

As part of Milestone 2, which this deliverable reports on, the AMPERE optimization phase is developed as a single-criterion optimization framework. This means that each optimization component is developed and tested in isolation, to ensure parallel development. In the next milestone, MS3, the optimization phase will be transformed into a multi-criteria optimization phase, in which all components are co-operating to ensure that the system fulfills *all* requirements modeled in the DSML. In light of this, the remainder of this deliverable presents each optimization component in isolation. However, as part of MS2 significant effort has already been invested in the planning for future integration, as mainly outlined by D6.2, and as this deliverable aims also to provide insights on how the independently developed optimization components fit together. Furthermore, this deliverable outlines the different considerations and techniques used to achieve the respective goals of each component. Fundamentally, this is achieved through the targeting of the common interfaces, such as the TDG, as described in D6.2. Note that due to the single-criterion optimization of this milestone, the ordering of the optimization components in the grey box in Figure 1 is not yet determined, but will be explored during the integration to the multi-critera optimization planned for Milestone 3.

Returning to Figure 1, once the optimization phase, highlighted in the gray box, has completed, the optimization phase is either finished, i.e., all functional and non-functional requirements are guaranteed to be upheld, or another round of optimization is required – in particular, the AMPERE optimization relies on an optimization loop as reported in D3.1, and additional profiling information may be required to complete the optimization stage, either by verifying that the offline determined configuration of the system is correct also during online execution, or to collect additional information required by one or more optimization components. This is achieved using the *convergence* outcome of the optimization stage, as shown in the figure, which exits the optimization loop once all optimization components have successfully optimized the respective NFRs. At that point, the binary system as compiled from the sources corresponding to the TDG is final. If additional information is required, e.g., a new profiling run, instead another iteration of the optimization loop is triggered.

As the TDG is a common data structure between the previous work packages and the optimization pipeline of WP3, it also enables a second important feature, at the end of the optimization phase. The encoded information in the TDG is fed back to 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. This is highlighted as the Annotated TDG in the figure.

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 is highlighted as the Graph to Source step, and provides the mechanism to encode optimization outcomes, such as DVFS configuration, scheduling orders, and redundancy aspects into the system. As shown in the top-right of the figure, this information is parsed by the AMPERE runtime components, developed in WP4, at the start-up of the system. 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.

The remainder of this deliverable describes the single-criterion optimization components in detail.

- Chapter 3 addresses Task 3.4, presenting techniques for modular redundancy for critical components, and coordinated solutions involving different layers of the AMPERE system stack. It presents new software protection techniques for fault tolerance.
- Chapter 4 addresses Task 3.2, and the optimization of energy consumption in the AMPERE system, as well as strategies to utilize the available power management knobs to optimize overall energy-performance for major components in the system.
- Chapter 5 addresses Task 3.3, investigating the timing effects of host, accelerator, and memory system, and communication cost impact on time-predictability. It presents the ROS middleware and the Logical Execution Time model for consideration in the project. It presents relevant real-time analysis techniques to estimate upper bounds on the response times, as well as resource reservation paradigms of the FPGA fabric.

# 3. Resilient Software Techniques

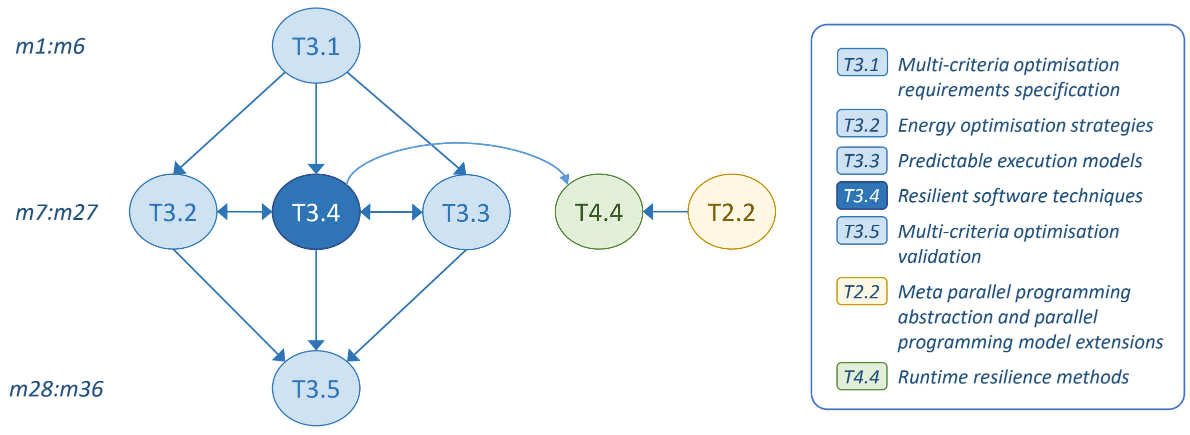
*Dependability* is the property of a system that reflects the user's degree of trust in that system, i.e., the confidence of the user that it will operate as expected and that it will not fail in normal use. Different metrics are considered for measuring the dependability of a system, including the following:

1. *availability*, or the probability that the system will be up and running when it needs to be used;
2. *reliability*, or the probability that the system will conform to specification during a specified period of time;
3. *safety*, a measure of how likely the system will cause damage to people or its environment;
4. *integrity*, the absence of improper system alterations; and
5. *maintainability*, or the capacity of the system to undergo modifications and repairs.

As described in the DoA [2], and based on the current trend towards many-core heterogeneous platforms, AMPERE aims at enhancing the *robustness* of the system, and thus focuses in particular on improving *reliability* and *availability*.

This section is based on the studies presented in *D3.1, Multi-criteria optimisation requirements* [3] with regard to software and hardware resilient methods, and it includes the work performed in *Task 3.4, Resilient software techniques* during Phase 2 of the project (m7-m15). For this period, Task 3.4 focuses only on fault tolerance, being the target at MS2 a first version of software resilient solutions not considering other non-functional constraints. Figure 2 presents a graphic description of the synergies of Task 3.4 and other tasks in the project.

Figure 2: Synergies between Task 3.4 and other tasks in the project.


m1:m6
T3.1
m7:m27
T3.2
T3.4
T3.3
T4.4
T2.2
m28:m36
T3.5
T3.1 Multi-criteria optimisation requirements specification
T3.2 Energy optimisation strategies
T3.3 Predictable execution models
T3.4 Resilient software techniques
T3.5 Multi-criteria optimisation validation
T2.2 Meta parallel programming abstraction and parallel programming model extensions
T4.4 Runtime resilience methods

The reminder of the section introduces first the software resilient solutions implemented (Section 3.1), and then a study of the interaction of the described techniques with critical aspects of the overall system, including scheduling, orchestration and proactive reaction to faults (Section 3.2).

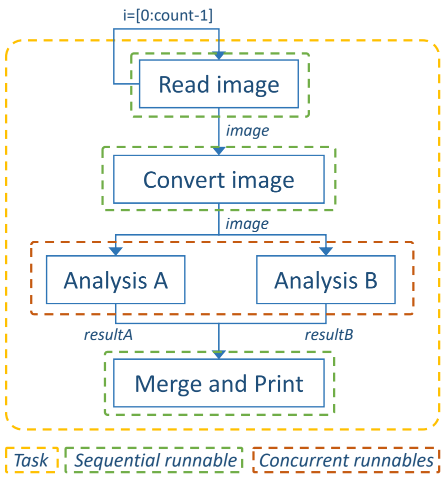
## 3.1. Software Resilient Solutions

Correctness is of great significance in real-time systems, especially if they are applied in automotive or related industries, where results might influence people's safety. Even cosmic radiation can influence systems, which makes it impossible to guarantee a correct result in some cases. Therefore, in the most critical parts of the system, where errors are not tolerable, there might be a need to double-check the result that is being produced. This can be done by repeating the same work multiple times, and then comparing the results, a.k.a. *redundancy* or *replication*.

Redundancy affects different layers of the AMPERE's software stack. First, the DSML (i.e., AMALTHEA) shall expose a new feature to enable user's defining which parts of the model must be replicated, and which are the results to be considered for result checking. Then, the code synthesis tool (i.e., APP4MC SLG) must be enhanced to understand these new capabilities, so it can transform the DSML features into features of the parallel programming model. Consequently, the parallel programming framework (i.e., OpenMP), including the model, the compiler, and the runtime system, must be enhanced as well (as already noted in D3.1 [3]) to include the new redundancy feature.

To illustrate the modifications implemented targeting software resilience, Figure 3 shows the workflow of a simple application enriched with information on how the application can be modeled with AMALTHEA tasks and runnables. The application runs $count$ times a pipeline composed of four steps: (1) read an image; (2) convert the image to a suitable format; (3) two different concurrent activities process the image to produce results; and (4) merge and print the results. The figure also shows which parts are suitable for runnable granularity (green line for sequential runnables, and orange line for concurrent runnables, i.e., runnables not causing race conditions) or task granularity (yellow line), considering the AMALTHEA model.

Figure 3: Workflow of a sample application.



The AMALTHEA model conveniently includes *custom properties* that allow enhancing the model in a generic way. Their reason to be is that AMALTHEA is constantly evolving, and many properties might not be yet available, as it is our case. So we include a new custom property, called *redundancy*, to define which runnables are to be replicated. The property includes an integer value defining the number of replicas to create.

The automatic code generator [4], provided by the AMPERE partner BOSCH, is a plugin to the APP4MC project that currently transforms AMALTHEA models into sequential C programs. We have incremented this tool to accept the new custom properties that define parallelism (as described in D1.3 [5]), and also transform the *redundancy* custom property into a new clause attached to the OpenMP `task` directive[1]. Figure 4 shows a graphic description of the features at the AMALTHEA and OpenMP level, and the role of the code synthesizer.

Figure 4: High-level description of the transformations for software resilience.

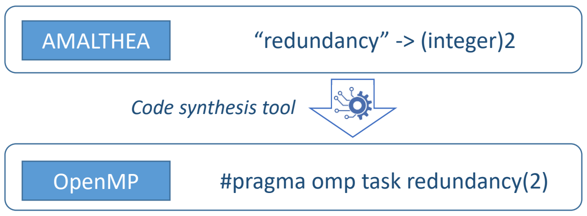


Figure 6a shows the OpenMP code generated automatically from the application shown in Figure 3. Furthermore, Figure 6b shows the Task Dependency Graph (TDG) that represents the execution flow of the OpenMP tasks. There, $T3$ corresponds to the $AnalysisA$ task (or runnable in AMALTHEA), and it is duplicated together with the data dependencies, as indicated in the `task` directive. The duplication has an impact on the non-functional requirements considered in the project, including time, energy and performance. For this reason, the TDG is enriched with this information, and then used by the analysis tools of the project for considering it in the analysis (the interface for using the TDG across the different tools in AMPERE, including compiler and the diverse analysis tools, is defined in D6.2 [7]).

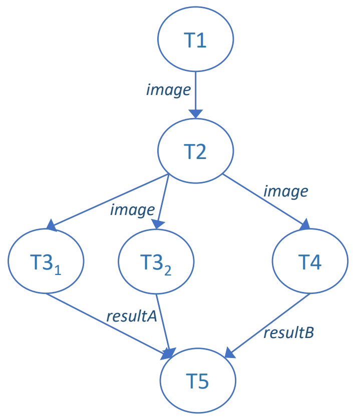
[1] Details about the transformations implemented in the AMALTHEA code synthesis tool targeting performance and redundancy are presented in D2.2 [6]

Figure 5: Automatic transformations generated from the application in Figure 3.

(a) OpenMP code generated with the AMALTHEA synthesis tool.

```
 #pragma omp parallel
 #pragma omp single
 {
   #pragma omp task depend(out:Image)
// T1
   run_read_image("");

   #pragma omp task depend(inout:Image)
// T2
   run_convert_image("");

   #pragma omp task depend(in:Image) \
         depend(out:ResultsA) redundancy(2)
// T3
   run_analysisA("");

   #pragma omp task depend(in:Image) \
         depend(out:ResultsB)
// T4
   run_analysisB("");

   #pragma omp task depend(in:ResultsA,ResultsB)
// T5
   run_merge_results("");
 }
```

(b) Task Dependency Graph generated by the OpenMP compiler/runtime.



The information described in the OpenMP directives and clauses is later processed by the compiler and the runtime systems. The current implementation works as follows:

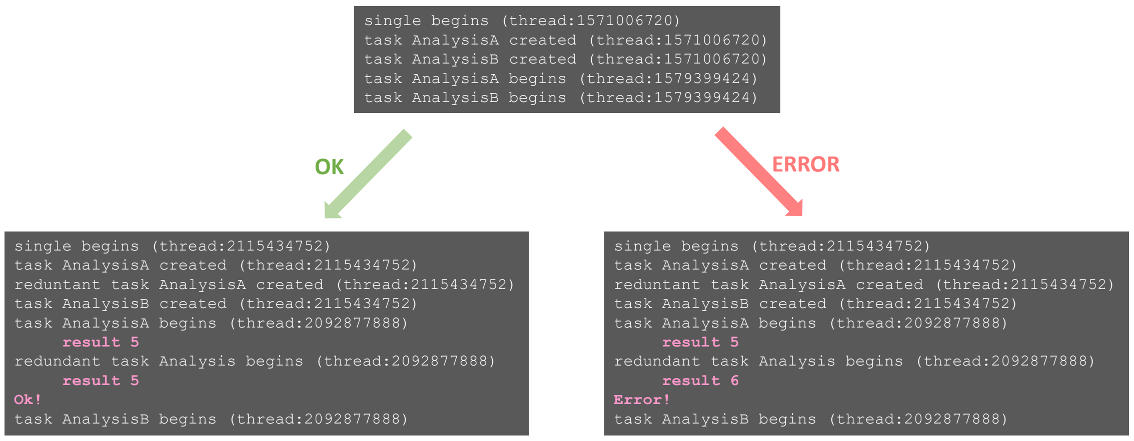
1. The compiler transforms the `redundancy` clause into a new parameter added to the runtime call that creates the corresponding task. This implementation uses the Mercurium [8] source-to-source compiler and the GCC libgomp [9] runtime library.
2. The runtime system interprets the new parameter and creates additional tasks (the number is defined by the value passed as parameter) to validate the correctness of the work carried out. This process is transparent to the user, except if an error is detected in the computation of a redundant task, in which case either the user, or other parts of the system, will be notified.

Figure 7 shows an example of the report currently provided by the runtime. In the left, the report shows a correct execution with no errors detected with the redundant tasks, and in the right, the report when an error has been detected.

## 3.2. Proactive Orchestration

The strategy investigated by THALIT/UNISI is in line and complementary to the one carried out by BSC. It aims to build fault tolerance from the application layer, which means that the programmer is responsible for deciding which aspects of the algorithm need to be protected, making a distinction between programmer responsibility

Figure 7: Sample report produced by the runtime executing the application in Figure 3 with *AnalysisA* replicated.



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

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

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

and automatic fault tolerance. The main part of the effort was spent trying to figure out ways for limiting the programmer effort and the interaction between application and fault tolerance code as to minimize the effects of this strategy on code productivity, maintainability, correctness testing. The THALIT/UNISI approach deal with the single instance of an application execution, aiming to enforce some predicative conditions on state variables. The whole procedure is quite general: it is composed of modular observing code built around a predicate, therefore a condition over certain state variables of the algorithm to be observed, which determines when its behavior is within the expected bounds.

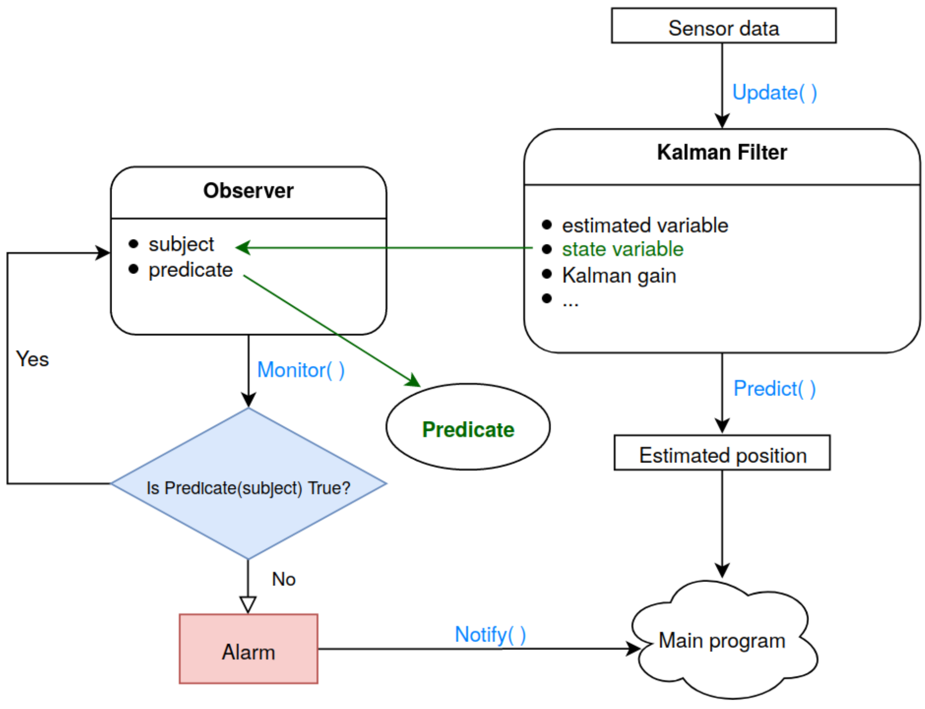
UNISI has studied an approach, in agreement with THALIT, for allowing the programmer to specify an observer code on some state variables of the algorithm. This approach aims to improve the resilience of the application by focusing on early detection of symptoms that may lead to system faults. The final purpose is to develop an online run-time technique which can detect errors in software behavior that could lead computed values to diverge from those of an error-free program execution. This can be done using an external piece of software capable of observing the main program and then raise a flag if there is a divergence in the software-state evolution compared to the normal one. In this manner, it is possible to provide the protection required by critical applications with a low cost (in terms of space, design complexity and power consumption) and without slowing them down, enabling high performance in a safer environment. In fact, one of the requirements of this mechanism is to limit as much as possible the interference between strictly application code and this observer based control code. In this way, by maintaining these two concepts as much decoupled as possible, we allow the programmer to specify some predicates over the state variables, which are observed during the software runtime, as to understand if the monitored predicates are within the expected bounds or if some of them are starting to diverge.

The main idea is to develop an evolution of a common C++ design pattern called Observer. It is a behavioral design pattern that lets the programmer to define a subscription mechanism to notify multiple objects about any events that happen to the object they are observing. Starting from this simple but instructive template, the goal is to produce an observer capable of monitoring the behavior of a specific software component, being at the same time as independent as possible with respect to the observed object. One of the main advantages related to this approach concerns the possibility of having an observer code which is external respect the main code and with minimal coupling with it. In fact, this approach will not increase the complexity of the observed code, therefore limiting the interaction with the observer only to those components strictly necessary for the purpose of sharing the state of the interested variables. This also guarantees a further advantage, namely that of the generality of the observer: proceeding along this path it will be possible to observe different types of variables without requiring any modification to the observer code. Another relevant aspect concerns the way in which the correct behavior or not of the observed variable will be verified. To this end, the observer code needs some reference values to compare with the observed variables, for determining their correct value.

In fact, the observer will monitor if the observed variables keep satisfying the predicate which indicates a correct behavior. Conversely, when the predicate becomes false there is an evidence of a possible incorrect execution. Such predicates are a function of the specific case and observed algorithm. For this purpose, a specific predicate will be associated to each instance of the observer code, with the aim of using it to monitor the behavior of the state of the algorithm. Each predicate is passed as a function parameter to any specific observer code.

The proposed software resilience method was applied within the THALIT use-case of the AMPERE project, in order to monitor the behavior of a Kalman Filter algorithm. The first experiments done so far have proved to be promising, although further refinement will be needed, with the observer which is able to monitor the Kalman filter behavior and notify the user when the predicate fails. Figure 8 shows a block diagram on the operation of the Observer.

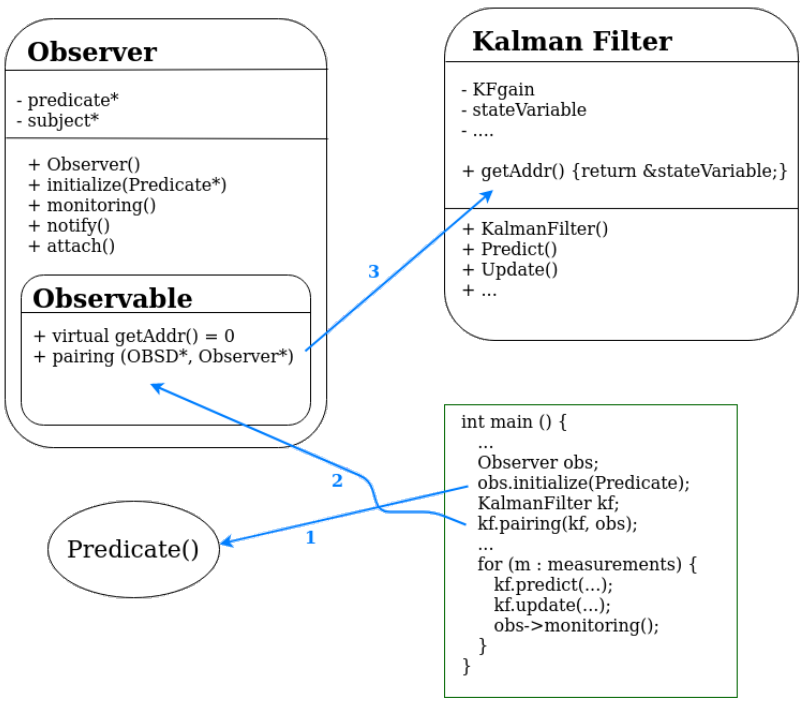
Figure 8: Block diagram on the operation of the Observer module



In such figure, the two rounded rectangles represent the two classes *Observer* and *Kalman Filter*, the black rectangles represent the input and output variables for the Kalman filter class, and the circle stands for the *Predicate* function. In addition, the blue labels represent the different functions and the two green arrows are used to indicate the state variable and predicate that are read as input of the *Observer* class. As can be seen, the Kalman filter reads the sensor data in input and produces as output an estimated position which is then used by the main program. During this procedure, obviously, its internal state can change and this change is captured by the *Observer*, which monitors the behavior of the state variable. For each cycle of the Kalman filter, using the Predicate function, the Observer is able to check whether the state variable is still valid (blue diamond block), otherwise it is possible to trigger an alarm to notify the main program about this unexpected behavior.

A more detailed workflow explanation is shown in Figure 9, where the state of interest is represented by a single variable for simplicity, and the predicate is a threshold on such variable. The KalmanFilter class, which represents the application code, derives from Observable class and it must implement the *getAddr()* function to return the address of the state needed by the observer's predicate. Lastly, in the main function (green box), the whole procedure related to the initialization and use of the Observer class is reported. In particular, after the procedure for initializing and coupling the Observer with the Kalman filter, each time a new input value arrives it is sent to the Kalman filter class to be processed. Then, the Observer is called for monitoring the variable of interest. In this way the Observer checks the new value at very update.

Figure 9: Execution workflow of the Observer module



```
int main () {
  ...
  Observer obs;
  obs.initialize(Predicate);
  KalmanFilter kf;
  kf.pairing(kf, obs);
  ...
  for (m : measurements) {
    kf.predict(...);
    kf.update(...);
    obs->monitoring();
  }
}
```

## 3.3. Summary

The two mechanisms proposed by BSC and THALIT/UNISI have some common aspects but they are also complementary, each one with its own pros and cons. So, dependently on the specific case, the agnostic replication assisted by the run-time execution proposed by BSC could be assisted by the observer solution proposed by THALIT/UNISI, with the cost of having the programmer a little bit more included in the loop. Even if at the first sight one could think that these techniques achieve the same objective by using different approaches, it is not completely true and it may have sense to combine the two. In fact, the observer approach proposed by THALIT/UNISI monitors the software, and it could identify the cases in which the execution starts to diverge from the expected one, but not sufficiently to be yet triggered by wrong outputs. This is a sort of gray area, which cannot be managed using replication techniques, where this approach allows some different options instead of simply reject the outcome of that process, for instance raising a flag as initial alert but keeping-on to use the calculated results. In such cases, instead of seen the function as a black-box, with some inputs and outputs, the observer may allow to monitor also some internal mechanisms of it, providing a bit more of flexibility.

Another possibility can be to use both these techniques together, with an orthogonal approach: it could be possible to replicate the algorithm execution including the observer onboard, i.e. replicating the observer along with the different copies of the function. Even if this process results in a simple copy-paste of the code, it could however provide some useful information since it might catch some software or transient errors which may happen in different places of memories or different units on which the several copies of the code are being executed. Furthermore, another advantage that could be obtained by using both strategies together, is to make more efficient the control of the outputs produced by the replicated versions of the software. In fact, comparing different outputs of some replicated functions could be difficult or time consuming. This checking procedure, which is needed by the BSC replication approach, could be simplified if the THALIT/UNISI observer will be capable of providing some metrics in order to determine if the execution has been correct or not. The key advantage is represented by the fact that these metrics measured over some observed state variables would have a smaller size than the final output, which make the comparison procedure easier. Lastly, another improvement is related to the final majority voting phase: in the aforementioned cases in which a soft-error is corrupting the execution of a certain copy, the observer can detect it and raise a flag so as to alert the redundancy manager that one of the copies is not working properly and, consequently, this can be taken into account in the majority voting phase. Therefore, the final objective is to integrate the two approaches in an orthogonal way as to compensate for any weak points of each other and exploit the strengths of the two techniques.

In conclusion, THALIT/UNISI and BSC approaches can be used in isolation and, possibly, also in an integrated fashion with minimal coupling, as to adapt to the various parts of a complex system in order to put down ad-hoc strategies for fault-tolerance in software. In fact, the two different approaches could be seen as a sort of "tool set" for the programmer, for providing the possibility to use one, the other or both together, depending on the needs required for a correct assessment of the software execution. In our view, this would provide a significant prospective for the whole AMPERE project.

# 4. Energy Optimisation Strategies

Task 3.2 is responsible for the development of efficient resource management strategies for optimizing the energy efficiency of the system, and the target at the presented MS2 is the evaluation of energy optimization aspects in the single-criterion optimization phase. Based on this, the presentation can be divided into two clear sub-goals. The first is to provide a deep understanding of the energy profile of different tasks, such that their optimal energy configuration given specific constraints can be identified. Such constraints may be inherent to the energy requirements of the modeled application, or follow from functional and non-functional requirements that are not strictly related to energy, e.g., real-time execution requirements. The second sub-goal is to use this information to optimize the energy efficiency of the system.

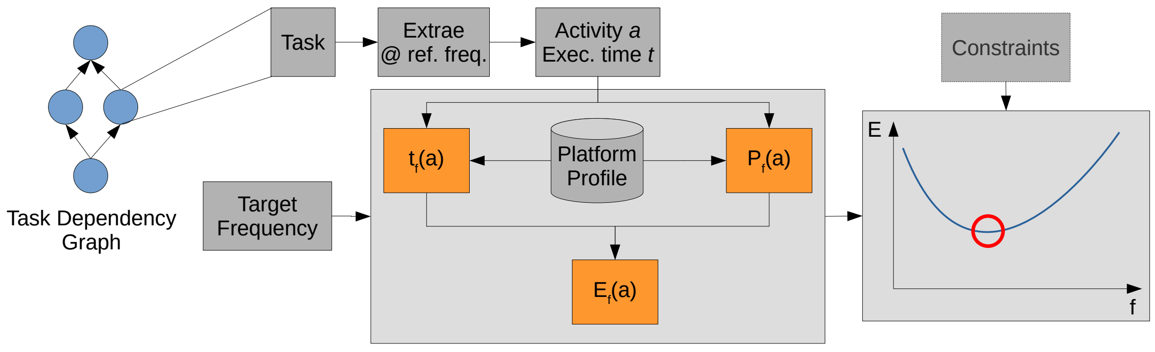
There are two main types of strategies that can be used to improve energy-efficiency; non-intrusive and intrusive methods. Non-intrusive methods do not require any changes to the *runnable* code within tasks, primarily by utilizing hardware techniques. The main non-intrusive technique available for controlling energy-efficiency is Dynamic Voltage Frequency Scaling (DVFS). By altering the operating voltage and frequency, the overall energy spent to complete a task can be impacted. DVFS can be statically decided during offline scheduling, but can be made further effective by dynamically rescheduling the system at runtime, by redistributing energy budgets among tasks to further optimize the energy usage. Energy efficiency optimization through intrusive methods involves changes to the code of runnables in the system, e.g., through reduced precision computing, which requires less switching activity to perform the computation.

As part of MS2, Task 3.2 primarily presents results in the front-end, related to the understanding of the energy characteristics of tasks, as well as in non-intrusive static DVFS optimization. This is motivated by the single-criterion aspect of MS2, in which energy is optimized in isolation from other non-functional requirements.

## 4.1. Energy Analysis Front-End

An overview of the front-end for the energy optimization framework is shown in Figure 10. The input to the energy optimization framework comes from the Task Dependency Graph (TDG), shown in the left of the figure, and described in detail in D6.2 [7]. For each task in the TDG, the energy budget as described in D1.3 is forwarded (as described in D2.2), together with additional functional and non-functional information about the task. Together with this, the profiling data for each task is provided, as generated by the Extrae tool, as described in D6.3. Using this profiling information, the energy optimization front-end produces an energy profile for each task.

Figure 10: An overview of the energy estimation flow in WP3.



To reduce the amount of profiling runs that are required, the Extrae profiling is only performed once at a reference frequency $f_{ref}$. From this reference profile, the energy front-end extracts the performance counter activity $a$ and the execution time $t$ from the profiled execution of the task. This is used to estimate the energy profile of the task at the remaining DVFS operating points of the system. The primary benefit of this approach is that the abstraction layer of the domain-specific modeling language remains intact, even when introducing complex time-frequency dependencies into the model. In the single-core, single-frequency case – i.e., the starting point before the AMPERE project – it is sufficient to specify a single execution time for each task. This execution time scales non-linearly with voltage/frequency and without this estimation framework, modeling complexity significantly increases as each task must have individually defined execution times for each frequency.

Given by the model or extracted from the Extrae traces, the inputs to the energy front-end are thus the reference execution time $t$ and the activity $a$, both of which are given at a single reference frequency $f_{ref}$. The front-end uses three components to produce an energy estimate for all frequencies $f \in F_{DVFS}$, where $F_{DVFS}$ is the set of discrete frequencies that the system can run at. To produce the energy estimate for $f \in F_{DVFS}$ it is given as *target frequency* $f_{target}$ to the front-end, as shown in the left of Figure 10. Internally, the front-end is constructed by three main components, the time-scaling model $t_f(a)$, the power-scaling model $P_f(a)$ and the energy-scaling model $E_f(a)$. As energy is the time-power product, the $t$ and $P$ models account for the effects of DVFS in each dimension. Intuitively, one can imagine a *time-power graph* (time on X axis) in which $t$ scales the switching activity $a$ horizontally in time, and $P$ scales the power usage of the switching activity $a$ vertically. Both models use pre-trained correlation databases that are part of the Platform Model.

The time-scaling model $t_f(a)$ is a set of linear models, one for each frequency $f$, on the form $t_f(a) = t_{reflen} \times (A \times f_{target} \times \frac{B}{a} + C) + D$, where $t_{reflen}$ is the excecution time measured at the reference frequency as part of profiling, and $f_{target}$ is the target frequency given as input to the analysis and for which the estimate is to be generated. $A$, $B$, $C$, $D$ are constants trained by fitting the model to the time scaling effects of a representative set of benchmarks, to account for different memory and compute patterns. This training is done ahead of time, and the weights are given as part of the platform model. This model serves as an initial and sufficiently accurate model for the single-criterion optimization phase, but will receive further attention during the multi-criteria optimization phase (as part of the next milestone) to improve accuracy also under co-operative time estimation from all relevant optimization criteria. At that point timing impact is not limited only to DVFS but also e.g., memory interference, which may also affect the definition of the time model. Further time estimation aspects are discussed in Section 5.

The power-scaling model $P_f(a)$ is fundamentally a wrapper around the power model $p_f(a)$ used in the online energy monitor, as described in D4.2, and is on the form $P_f(a) = p_f(a) \times A + B$. As outlined in D4.2, different counters are representative for the power usage at different frequencies $f$, and thus the composition of $a$ may not be the same for the reference frequency $f_{ref}$ and the target frequency $f_{target}$[1]. To address this, the Extrae profiling input contains traces for all counters that are relevant for any frequency, such that data for the expected counters can be passed to $p_f(a)$. However, as these counters are recorded at a different frequency, they need to be corrected to make the counters at $f_{ref}$, $a_{f_{ref}}$, representative for $a_{f_{target}}$ at the target frequency. This is achieved by fitting the model's $A$ and $B$ parameters using representative benchmarks. As with the time scaling model, this is done ahead of time and the information is stored in the platform model. Note that the model used does not contain any reference to $f_{target}$, as AMPERE will provide an individual $p_f(a)$ for each frequency $f$. Fundamentally, the chosen approach is to scale the output of $p_f(a)$ rather than scaling the input $a$. This approach simplifies the models, as compared to correcting each individual counter value in $a$, which would require approximately 50 different translations to be trained. The latter approach is prone to overfitting, which makes the former (and used) approach of scaling the output much more robust, providing better energy estimates over diverse sets of tasks. The parameters $A$ and $B$ used to achieve this also have intuitively understandable semantics, as $A$ provides the increase in power attributable to the switching activity being performed at a different voltage level, while $B$ provides an offset for the increase in idle- power under the same circumstances.

In the current implementation of the $t$ and $P$ models, the $a$ component is recorded over the entire task execution time, or using the terminology from D4.2, the entire task consists of a single *frame* – i.e., the time during which performance counters are registered. As part of our future work, we plan to apply the models

[1]See D4.2 for information on performance counter-frequency mapping

to individual frames of the tasks (which is an additional feature provided by Extrae), to better account for individual scaling factors of memory and compute bound segments of the code. This will improve the estimates for tasks which frequently switch between memory and compute bound execution styles. The online monitor presented in D4.2, benefiting from direct access to the performance monitoring unit (PMU), already uses this approach with frame sizes of $100$ ms. Finally, as shown in Figure 10, the energy usage is computed by $E_f(a)$ as the time-power product of the previous models.

By executing the front-end once for each $f_{target}$, an energy-frequency model is generated, as shown in the right of Figure 10. This shows how, for each task, the energy consumption changes with each DVFS operating point $f_{target}$. This output can then be used for the energy optimization strategies presented in the next section.

Fundamentally, the analysis front-end could be implemented without the scaling models, only incorporating the activity $a$ and execution time $t$ provided by Extrae, and using the power model of D4.2 as-is. This would however require that $f_{target}$ is always set to $f_{ref}$, the frequency at which the profiling took place, and would therefore require re-profiling of each individual DVFS operating point for each task in the TDG. The time spent in such a setup could therefore be approximated as $T_{profiling} = N \times |\tau| \times \sum_{f \in F_{DVFS}} t_f$, where $|\tau|$ is the number of tasks in the TDG, and $t_f$ is the execution time for the task at frequency $f$. The size of $|\tau|$ for the use-cases of WP1 are for reference expected to be between 1 and 21, although models can be arbitrarily complex. To get a representative profile for the task, it is usually necessary to repeat the measurement several times, giving the factor $N$. As this number grows rapidly with increasing complexity of the model, the presented scaling model allows us to decrease the profiling time approximately by a factor $|F_{DVFS}|$. On the platforms selected for the AMPERE project, $|F_{DVFS}| \approx 30$. The energy estimate recieved is verified before system deployment – through the *convergence* evaluation as shown in Figure 1 – and only if there is significant deviation, another profiling run is triggered. Thus, a more accurate model improves performance of the analysis phase, but does not affect the offline guarantees provided once the optimization phase has reached convergence.

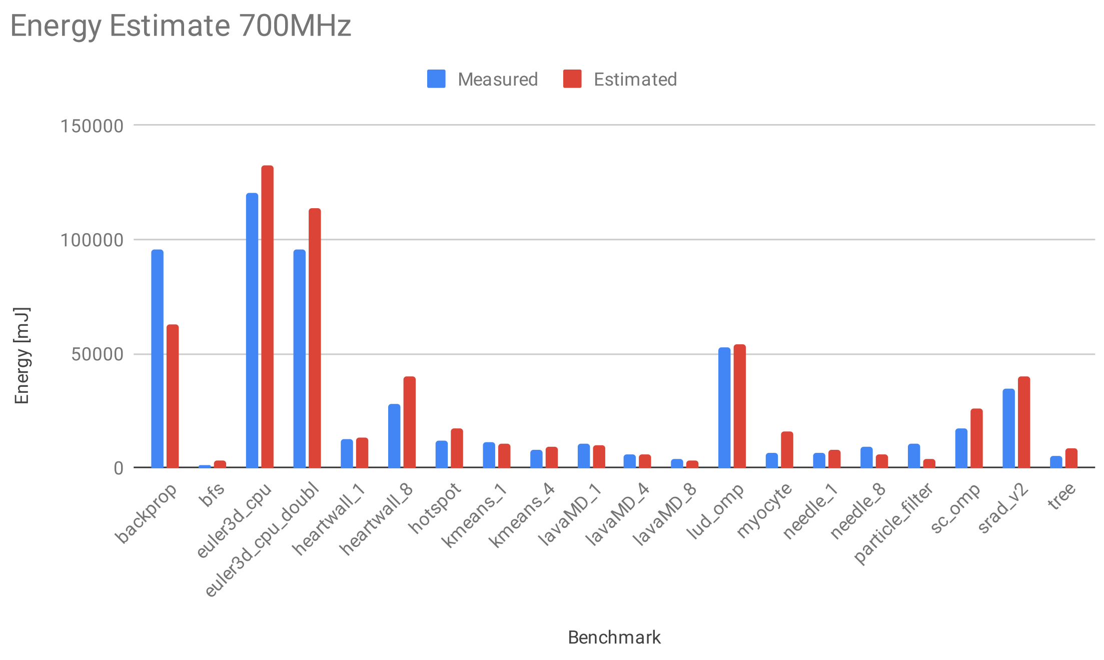
### 4.1.1. Energy Analysis Performance Evaluation

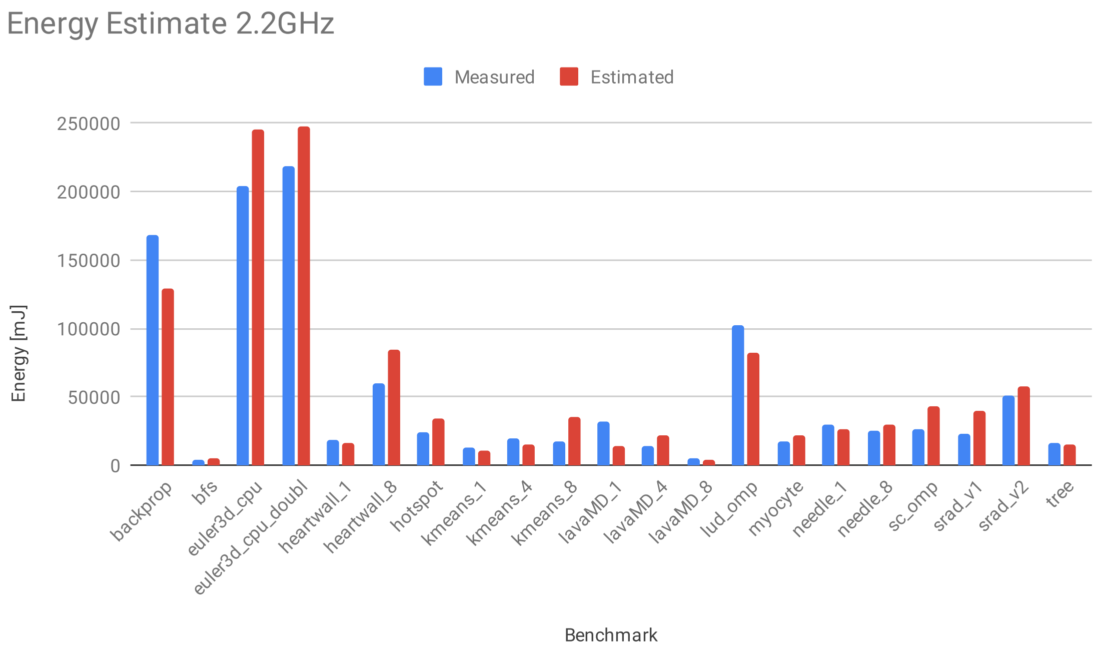
As the quality of the optimization strategies (to be presented in the next section) depends on the performance of the energy analysis done in the front-end, we provide here an evaluation of the approach to demonstrate its effectiveness. For this evaluation we use the Rodinia benchmark suite [10], which incorporates a large number of benchmarks with significantly different memory and computational patterns. Each benchmark corresponds to a single runnable, each as an individual task in the TDG. We perform the experiment on the NVIDIA Xavier platform selected for the project, profiling at $f_{ref} = 1.2GHz$, and evaluate the accuracy of the estimated energy values at $f_{target} = 700MHz$ and $f_{target} = 2.2GHz$, presenting the results in Figure 11.

The main pattern that can be seen, both for 700 MHz (scaling down from $f_{ref}$) and 2.2 GHz (scaling up from $f_{ref}$), is that the model is prone to over-estimation. While an exact model would be best, over-estimation is preferable to under-estimation, as this provides a safety margin for the system. The most significant part of the mismatch between the energy estimate and the measured numbers is the time model $t_f(a)$, which scales the execution time of the benchmark.

Due to the current implementation using only a single *frame* for the entire benchmark, the $t$ model has a hard time to estimate the impact of DVFS on the execution time, as the activity $a$ is spread out over the entire benchmark. This obfuscates discrete regions of compute and memory boundedness, which scale differently under DVFS. Scaling the core frequency during a memory bound region does not significantly alter the execution time, only fewer cycles pass while waiting for the memory latency (which has its own clock domain), which is not affected. In a perfect compute bound region on the other hand, DVFS has a more linear impact on energy. This effect is particularly clear in some benchmarks, e.g., *backprop*, which spends a large amount of its execution time initializing data, which is a memory bound operation. At the end, accounting for approximately 10% of the execution time, the benchmark becomes significantly compute bound. When considering a single activity *frame* for the entire benchmark, the activity $a$ is spread out over the entire benchmark, leading to an incorrect time scaling. Due to this effect, the *backprop* benchmark has its energy usage significantly

Figure 11: The energy values provided by the model as compared to the measured values for benchmarks from the Rodinia benchmark suite.





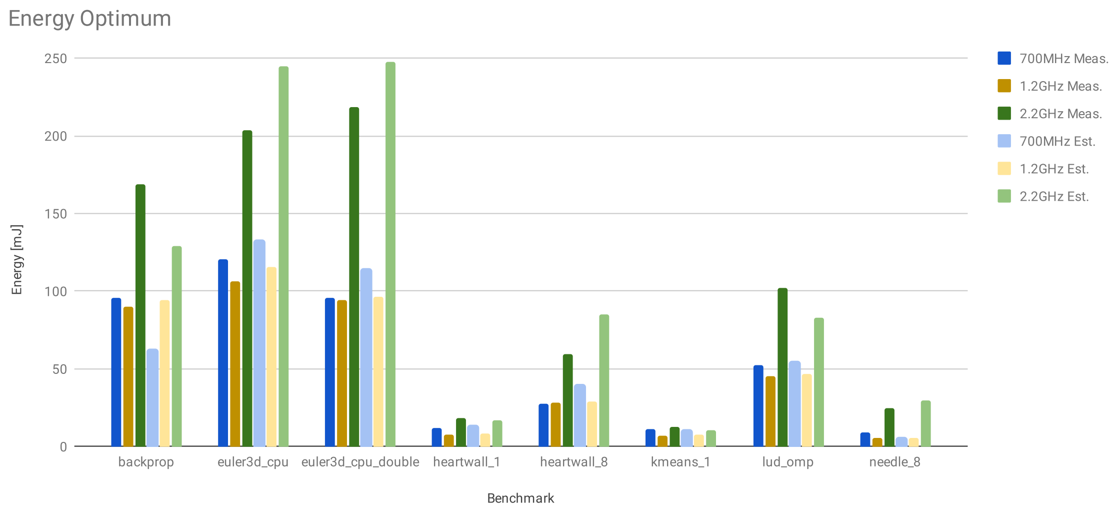
under-estimated. As outlined previously, the impact of this effect will be limited as part of ongoing development, as Extrae can be configured to provide performance counter activity at fixed intervals, allowing us to attribute activity to discrete *frames* and scale them independently. Furthermore, additional timing effects are investigated within the project (e.g., through inter-task interference), and as these aspects are combined in the multi-criteria optimization phase, additional development (or full integration) on the model is expected.

Another effect that is present in the NVIDIA Xavier platform is the custom hardware-managed clock and power gating, over which software has no influence. This hardware feature is activated for frequencies below 2.0 GHz, at which the hardware is actively trying to conserve power, but completely deactivated at higher frequencies, allowing the system to run at full throttle. An important effect of this is that the reference frequency used in the experiment $f_{ref} = 1.2$ GHz is executing in the hardware power managed mode, while only one of the target frequencies ($f_{target} = 700$ MHz) is. The other target frequency ($f_{target} = 2.2$ GHz) is running in the un-throttled mode. This difference impacts the recorded performance counter data $a$, and manifests in that the 700 MHz estimator performs significantly better than the 2.2 GHz counterpart. Also in this case, the memory and compute boundedness of the benchmarks play a role, as the un-throttled mode mainly affects the memory and SoC power domains, and as such memory bound benchmarks, e.g., *lud_omp*, see a larger scew in their estimated energy usages. It may be necessary to, on the NVIDIA Xavier platform, use two reference frequencies $f_{ref}$ to address this, if the accuracy is not sufficient in practice. By using one reference frequency in the power managed mode, e.g., $f_{ref}^{managed} = 1.0$ GHz, and one in the full-throttle mode, e.g., $f_{ref}^{throttle} = 2.0$ GHz, the number of profiling runs would still be significantly reduced, but may provide significantly better analysis data for the energy resource management strategies.

The energy-estimate is important for the validation of non-functional energy requirements, however, an additional important aspect is how well the energy estimates can be used to select the most energy-efficient operating point, by generating the energy-frequency curve as illustrated to the right in Figure 10. A good estimation of the optimal frequency is key to ensure that the energy optimization strategies can optimize the system from an energy standpoint.

To evaluate this, Figure 12 shows the shape of the energy efficiency curve as approximated by the three frequencies used in the previous evaluation, $700$ MHz, $1.2$ GHz, and $2.2$ GHz. In the interest of readability, a representative subset of the benchmarks from the previous evaluation are shown, each with six bars. The three left-most bars present the measured energy usage for the benchmark at each of the evaluated frequencies, while the three right-most bars correspondingly present the estimated energy usage. The expected behavior is that the estimated energy numbers match the measured ones, and importantly, that the most efficient measured energy-efficiency can be identified also from the estimates. Only when this is the case it is possible for the energy resource management optimizer presented in the next section to make an informed and correct decision when optimizing for energy efficiency.

Figure 12: The estimation of the energy efficiency curves for Rodinia benchmarks, at three different frequencies.



The figure clearly shows that the estimated energy usage provides a good proxy for the measured values, without the cost of reprofiling. For most benchmarks, the relative height of each bar matches very well between the measured and estimated values – with a constant offset that matches the over-estimations discussed in connection to the previous plots. Such constant offsets do not affect the possibility to identify the most energy-efficient operating point $f_{target}$, as the relative differences between the operating points persist.

A clear deviation is shown in the *backprop* benchmark, which was identified already in the previous evaluation as having an incorrect $t$ scaling factor associated with it. In Figure 12, the overly linear scaling factor, from the single-frame $a$, can be clearly distinguished from the slightly U-shaped measured numbers. As outlined earlier in this section, we expect this benchmark to perform better once the multi-*frame* consideration of $a$ is introduced to the model, with the interval-based sampling from Extrae. As also outlined, the possibility of re-profiling until convergence (validation), implies that this mismatch will not lead to an incorrect system design, but the performance benefit of avoiding re-profilation is partially lost until the improvements to the $t$ model have been implemented. In this particular case, the practical impact of the estimator incorrectly predicting the 700 MHz operation point as the most efficient is negligible, as the energy-efficiency at 700 MHz and 1.2 GHz (the actual optimum) is very small, but should not be ignored.

These results for the generation of the frequency-energy curve, as in the right of Figure 10, are promising, and the limitations are addressable in the continuation of Task 3.2 during the next milestone. From the presented techniques and the resulting curve, the necessary input for the energy resource management strategies are provided.

### 4.1.2. Capturing Applications Timing and Power Behavior under DVFS

This section presents a profiling suite useful to jointly characterize the timing and power profiles of a number of applications under DVFS. Note that this tool is required to gather a comprehensive data set needed for schedulability analysis and platform optimization, and that it is planned to be integrated with the Extrae tool mentioned above. This will be useful to populate part of the AMALTHEA Trace Data Base (ATDB) mentioned in Deliverable D1.3 [5], that will end-up populating the AMALTHEA model with timing and energy annotations, needed for the platform optimization phase.

The proposed profiling suite can be used for the twofold goal of collecting meaningful data about real-time tasks execution on the target machine and later analyze said data in a second moment offline. First, the suite is deployed on the designated embedded platform, on which it performs a set of automated profiling runs, collecting data coming from various sources (e.g., task execution time, CPU frequency, platform power consumption, CPU temperature, CPU counter values, and so forth). After collecting results from all the runs, another software component, typically running on a general-purpose machine, can then post-process the collected data, calculating statistics and converting them into a suitable format to be used in other software tools. We will henceforth refer to these two components of the profiling suite as the *embedded* and *host* component respectively. Both components can be easily installed and configured on Linux hosts and embedded devices running Linux[2].

The *embedded* component can perform automatically several profiling runs on the designated platform; each run consists of running a specific workload/application on the target machine and collecting key information that can characterize the exectution of that workload in several working conditions. It is especially useful to characterize the behavior of each individual application on the target platform, since different workloads may exhibit diverse behaviors both with respect to DVFS [11] and to multi-core scalability, both of which are key elements that contribute to the efficient execution of applications on embedded platforms. Indeed, using our tools on a diverse set of embedded platforms (including a Xilinx Zynq UltraScale+ ZCU102, which is one of the target platforms of the AMPERE project), we can show how diverse these behavior can be, both with respect to the scalability of execution time and power consumption. Figure 13 shows some of the results of our evaluations to emphasize this concept.

Users can easily customize the set of workloads that are profiled on each platform by editing a couple of configuration files. This allows for gathering data that more tightly represents workloads of interest. If applications to be profiiled need to operate on sets of data, required input files are randomly generated at the beginning

[2]At the moment, we successfully deployed the *embedded* component on two Linux distributions, Ubuntu and PetaLinux.

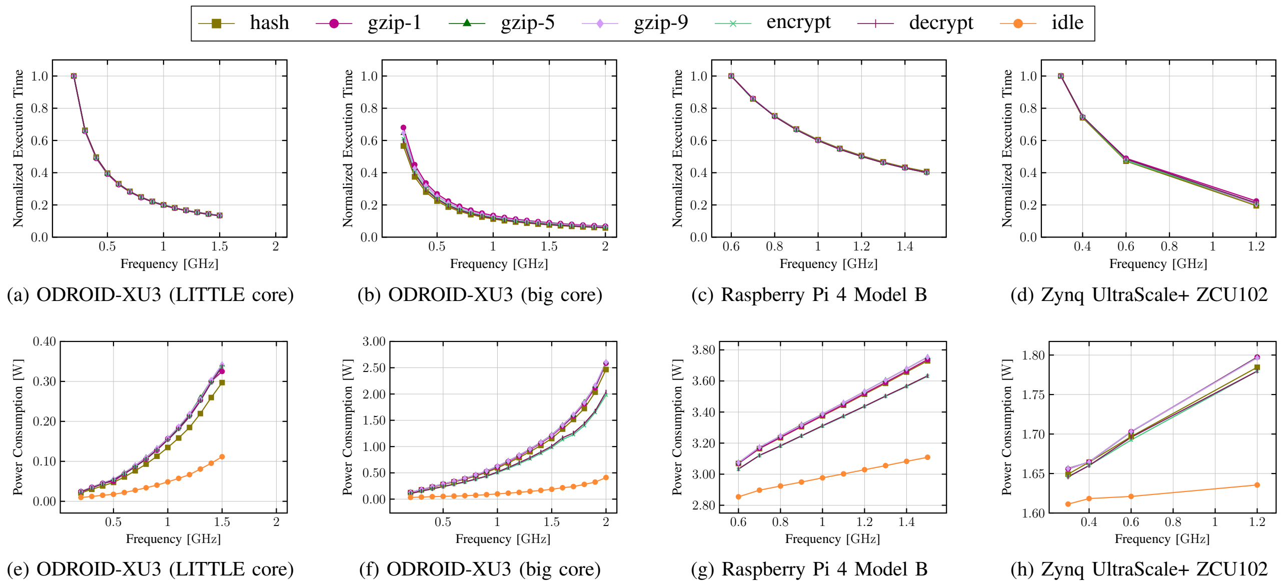


Figure 13: Variation of tasks execution time — (a), (b), (c), (d) — and power consumption — (e), (f), (g), (h) — when varying operating CPU frequency on various embedded platforms and core types, for a diverse set of workloads. All execution times are normalized with respect to the longest execution time for each workload type, usually registered for the smallest frequency of the least powerful core on each platform.

of each experiment and stored in a *ramfs* partition, limiting interactions with disks/SD card devices to the bare minimum (storage access is not simulated).

The profiling of a platform is performed by iterating all possible system configurations, varying working conditions by changing:

- CPU type, if the target platform is a single-ISA heterogeneous platform, providing islands with different kinds of CPU (e.g., ARM big.LITTLE or DynamIQ);
- CPU Operating Performance Point (OPP), i.e., frequency and voltage;
- the type of workload to be run;
- the number of concurrent instances of the selected workload to run in parallel (from none up to one per CPU).

Profiling multiple tasks on different cores simultaneously is particularly relevant because it allows for a more accurate representation of the system's behavior when multiple tasks are running concurrently. Each run can be repeated automatically for a configurable number of times to improve the statistical relevance of the collected results.

To collect data during each task execution, a special application is run concurrently to the profiled tasks. Preferably, the data collection application runs on a separate frequency island; on platforms that only have one frequency island, preference goes to cores not involved in the current profiling run, if any. The software periodically samples a set of key metrics; the set of supported metrics varies from platform to platform because different devices may expose different kinds of sensors for the same metric. In general, metrics collected during each run include power consumption, the temperature of the CPU, and actual CPU frequency, which may differ from the one selected by the profiling tool due to thermal protection mechanisms.

The design of the data collection application is modular and easily extensible to expand its support to more platforms in the future. Platforms that do not ship with internal sensors for power consumption or other useful metrics (e.g., CPU temperature) can also be profiled using external power meters attached to the embedded platform itself. To collect information about the internal behavior of each task, `perf` is typically used, but the suite supports other tools, including `PMCTrack`[3].

After completing the first phase, all data samples and logs are collected to be post-processed by the *host* application component. The *host* application calculates statistics on the collected data. Output format of this phase is currently comprised of a set of CSV format tables.

Finally, for the milestone MS3, this work will support Extrae for profiling, will be integrated to the AMALTHEA model, and will extend the support for the Xilinx Zynq UltraScale+ ZCU102 FPGA board.

## 4.2. Energy Resource Management Strategies

The main resource management strategy explored as part of MS2 is the use of statically assigned DVFS operating points to tasks. This approach is the most applicable to single-criterion energy optimization because it is non-intrusive, which preserves the separation-of-concern property of model-driven design (MDE) approaches. Thanks to the frequency-energy graph that is implicitly generated by the analysis phase, the single-criterion optimization for DVFS operating points is straight-forward: As there are no other criteria to be optimized for, the optimal energy efficiency is achieved by configuring the task to execute at the frequency at which the energy usage is lowest. This is illustrated in the right of Figure 10, where the most energy-efficient point on the curve has been selected.

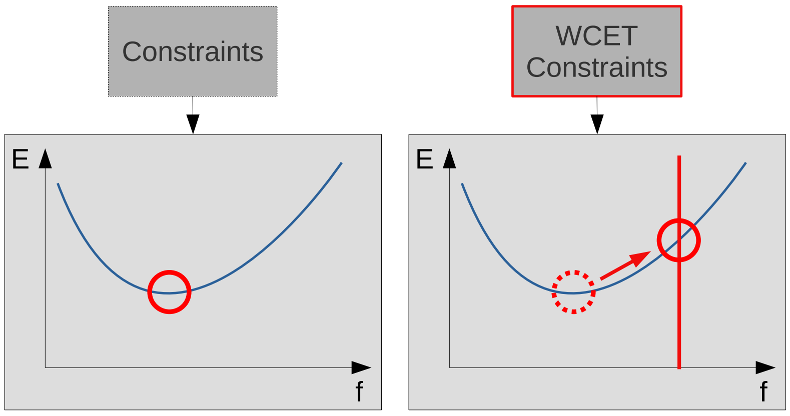
This process becomes more interesting once additional criteria are introduced in the optimization phase, as the DVFS operation point with the minimum energy usage may not be the best trade-off with other optimization criteria. Using real-time properties of tasks as an illustrative example, selecting the most energy-efficient DVFS operation point may set the system frequency too low for tasks to finish their execution within their period (or before their deadline). To address this, the resource management optimizer of the energy component includes the additional *constraints* parameter, also shown in the right of Figure 10, which can be used to limit the range from which DVFS operating points $f_{target}$ are selected. In the real-time example, the optimization may take place with a *WCET constraints*, that enforces the resource manager to consider all frequencies $f$ for which $t_f(a)$ exceeds the given value as non-selectable. This is illustrated in Figure 14, where the un-constrained (or single-criterion) optimizer in the left part selects the DVFS operating point with the lowest energy usage. In the right part of the figure, the red line represents the lowest possible frequency that fulfills the constraints on $t_f(a)$ (as given by the *WCET constraint*), and the optimizer instead selects the DVFS operating point that fulfills both requirements. As such, it provides the best trade-off between energy efficiency and external optimization criteria. In essence, these constraings provide the mechanisms through which the effects of other optimization criterion (e.g., timing, resliency) can be evaluated, which is an important aspect of the coming MS3.

To further extend this strategy, as part of MS3 we will explore the use of online slack reclamation mechanisms to improve the energy efficiency based on runtime data collected by the energy monitor presented in D4.2. Slack reclamation is a technique that at runtime monitors the constraint that caused a non-optimal DVFS operation point to be selected. As these constraints are based on conservative estimates, e.g., a worst-case execution time, the average execution time is likely to be significantly shorter. If so, the constraint that caused the non-optimal DVFS operation point to be selected no longer applies, and additional energy efficiency can be achieved by dynamically changing the voltage/frequency to optimize for energy-efficiency. As this requires closer integration of scheduling, timing, and energy-efficient components this task could not be started as part of the single-criterion optimization phase in MS2, but Task 3.2 still has the required time allocated as part of future milestones.

Lastly, another technique that could potentially improve the energy-efficiency of the system is the use of reduced precision arithmetic operations, e.g., by switching from 64-bit data types to 32-bit or even smaller (down to 8-bits). Narrower data types require fewer hardware resources, reduce the switching activity, and thus the energy usage. As part of MS2, ETHZ has explored the use of such techniques on the RISC-V research platform as adopted in the AMPERE project.

[3]PMCTrack website `https://pmctrack-linux.github.io/`.

Figure 14: In the single-criterion optimization phase (left) the energy optimizer is free to select the DVFS operating point with the best energy-efficiency for a task. However, in the multi-criteria optimization phase, additional *constraints* may be imposed and a different operation point selected.



The main idea of reducing the precision is to enable SIMD-style execution of several narrower data formats at once, using the wider architectural registers. For example, two 16-bit floats can be packed into a 32-bit register, and computed on simultaneously using a single instruction. The work utilizes and implements instruction extensions that are in line with the RISC-V standardization proposal for digital signal processing[4] (DSP), which supports SIMD operations for reduced precision floating point operations. The ETHZ-developed RISC-V platform used in the project features a host processor and an accelerator of a parameterized number of simple RISC-V cores [12]. The accelerator cores have access to a tightly coupled data memory (TCDM), acting as a L1 scratchpad, providing single-cycle data access. The system further features a larger 15-cycle latency L2 cache.

ETHZ has on this RISC-V platform explored the effects of different configurations for floating point units (FPU), including support for different floating point formats, SIMD vectors, and FPU pipelining. Additionally, the effects of sharing FPUs among multiple cores, to explore the energy-performance trade-off. In our experiments, we explore primarily 16- and 32-bit formats, the former of which can be executed in SIMD style by packing them into a single register. We connect a variable number of FPUs to a variable number of cores as auxiliary processing units, and arbitrate requests from different cores to each FPU using a round-robin policy. The key insight here is that for most workloads the density of FPU instructions is smaller than 50%, meaning that a dedicated FPU for each core may be under-utilized, and unnecessarily drawing power while not used. Similarly, the use of narrower instructions have the aforementioned promise of reducing the overall energy needed for arithmetic operations.

The three-dimensional design space explored consists of an accelerator with either $C = 8$ or $C = 16$ cores, either with $\frac{C}{1}$, $\frac{C}{2}$, or $\frac{C}{4}$ FPUs, and each configuration is evaluated either with 32-bit floating point, or SIMD 16-bit floats. The performance-energy trade-off is explored with every configuration running at $100$ MHz, deployed on the UltraScale+ programmable logic. The results show that for all configurations the $\frac{C}{4}$ configurations use the least amount of energy, but interestingly, the second best configuration is $\frac{C}{1}$, while the middle configuration $\frac{C}{2}$ uses most energy. This is because lowering the amount of FPUs decreases the number of components that have to be powered, while increasing the amount leads to each component being idle more of the time. Due to the more complex interconnect and arbitration logic for shared FPUs, the $\frac{C}{2}$ is not able to lower the number of powered components enough to offset the additional power for the arbitration. However, overall the energy-efficiency, i.e., the operations per second per watt, increases as the number of FPUs increase. This can be attributed to the reduced number of stall cycles, due to FPU sharing, which are determinental for energy-efficiency.

For reduced precision SIMD operation, the theoretical speedup that can be achieved is $2\times$, as two 16-bit opera-

[4]RISC-V Extension *P*, https://github.com/riscv/riscv-p-spec/blob/master/P-ext-proposal.pdf.

tions are computed at the same speed as one 32-bit operation. For code with little control flow, such as convolutions, fast Fourier transform, and matrix multiplication, this speedup is also achieved in practice, although more complex code may experience lower or negligible speedups. The improvement in energy-efficiency largely correlates with the experienced speedup, as the switching activity for the operation of 2 $\times$ 16-bit SIMD and 1 $\times$ 32-bit scalars are similar, while the total number of operations is reduced.

While the number of FPUs in the system can not be affected for the host processors or the GPU, it may be used on the FPGA. The usage of reduced precision or transprecision computing in AMPERE is yet to be determined – while automatic tools exist [13] for such transformations, it may be a task better suited for manual implementation by domain experts in the runnables themselves. In such a scenario it would not require extensions to the modeling language or the AMPERE toolchain. However, it remains an effective way to improve energy-efficiency in applications that are robust to reduction in precision, e.g., neural networks, and can be further explored within the scope of the project.

## 4.3. Summary

In this section the analysis and the single-criterion optimization for energy-efficiency have been presented. In line with the introductory notes in Section 2, the Task Dependency Graph provides the structure, and Extrae the profiling information of the system, together providing the necessary input to optimize the system for energy-efficiency at a fine granularity. The model-based estimation of the energy-frequency curve provides a low-overhead technique to generate the necessary data on which the energy-efficiency of the system can be optimized. Furthermore, the use of constraint mechanisms provides an interface for other optimizations to impact the solution space identified by the energy component, to enable multi-criteria optimization. The constraint mechanisms, it is also enables monitoring of the energy efficiency of the system and runtime optimization, if the constraint is not present at runtime. This technique corresponds to the predicate technique for observation presented for the resilience techniques. Finally, Section 4.1.2 presents a complementary timing/energy charaterization approach that still needs to be integrated into the AMPERE ecosystem.

# 5. Predictable Execution Models

This section presents several techniques and methodologies to provide predictable execution of tasks on heterogeneous and parallel systems, contributing to Task 3.4. It begins by addressing the issue of memory interference, followed by the general and profiling-based techniques used to provide timing information for tasks in the system. The Logical Execution Time paradigm is then presented as an overarching framework which provides a structured approach to addressing these two subjects. Next, techniques for probabilistic analysis of DNN workloads on FPGA are presented. Finally, a tool called DART optimizes the hardware partitioning of real-time tasks offloaded to an FPGA.

## 5.1. Predicting and Controlling Memory Interference in Integration Platforms

We are currently witnessing an evolution in the automotive electronics architecture space owing to multiple factors, the most prominent ones being the need for higher levels of automation, electrification and increased connectivity. One of the key enabling technologies in realizing these goals is the availability of high performance computing platforms that can host ever more data- and compute-intensive functions. The trend is to move away from distributed designs, where each function is hosted on its own dedicated microcontroller-based Electronic Control Unit (ECU), towards a more centralized design where multiple functions are consolidated onto powerful multi-core and multi-processor system-on-chip based domain/zone control units and eventually centralized vehicle computers. Applications developed by multiple vendors with varying Quality of service (QoS) requirements, different criticalities and varying trust zones, will be co-hosted on such common integration platforms. A key challenge for system designers is to design composable mechanisms on such integration platforms that ensure freedom from interference among different applications and allow diverse applications to co-exist, adapt to dynamic workloads and meet their timing requirements while also utilizing the system resources effectively.

Building such mixed-criticality integration platforms imposes new challenges for a system designer, given that these powerful platforms typically feature multiple processing elements, sharing some hardware resources like the interconnect and the main memory. As a consequence, when applications are co-deployed, interference through shared resources leads to undesired application performance coupling [14] and as a result, non-negligible context-dependent execution-time variability. Composable mechanisms like reservation-based scheduling [15] or the usage of hypervisors have been proposed to provide temporal isolation in terms of core usage. However, especially for memory-intensive applications, interference through the memory subsystem is a primary source of variability as well as of performance bottlenecks. For instance, the work in [14] has shown that co-hosted applications can suffer from an increase in the average (sequential) read access latency by up to 8x as compared to standalone execution due to memory interference.

In AMPERE, Bosch is currently working on mechanisms to a) upfront predict the run-time of co-running independent applications in multi-core microprocessors and b) dynamically regulate memory accesses of individual cores to provide temporal isolation and reduce memory interference during runtime.

The base for the prediction are performance counter values extracted from applications executed in isolation. For prediction we use machine learning based methods to predict the performance impact of multiple applications executing in parallel. To control the interference during runtime we work on a novel approach for memory interference isolation that uses a feedback control to dynamically regulate memory accesses of individual cores in a multi-core microprocessor system based on the saturation (utilization) level at the memory controller. Our mechanism directly regulates the source of interference by leveraging the information regarding the memory utilization, acquired from existing hardware performance counters provided by modern COTS memory controllers. Details of mechansim to control memory interface will be presented in the next version of this deliverable.

### 5.1.1. Interference-Aware Runtime Prediction

One of the most notable causes of interference is the main memory subsystem, which results in significant degradation in application performance and response time. Consequently, early run-time prediction of co-running independent applications becomes challenging in multi-core processors. Currently available techniques for run-time prediction like traditional cycle-accurate simulation is significantly slow, and analytical models are not accurate. In contrast, existing machine-learning based approaches [16], [17], [18], [19] and [20] do not take interference into account.

We use a machine learning-based approach to address this challenge by training a model to correlate performance data for a set of benchmark applications between the standalone and interference scenarios. After that, the trained model is used to predict the performance of newer applications in interference scenarios. We take advantage of the hardware performance counters present inside the platform and incorporate them as features into our model. In our proposed framework, samples obtained from the standalone and interference scenarios do not satisfy simple one-to-one correspondence. To address this, we develop a simple yet effective sample alignment algorithm. In addition, we systematically identify the subset of features that have the highest positive impact on the model performance. The efficiency of our approach is validated by predicting the average run-time of a new application and mean absolute prediction error of the model in interference scenario while executing applications in standalone only.

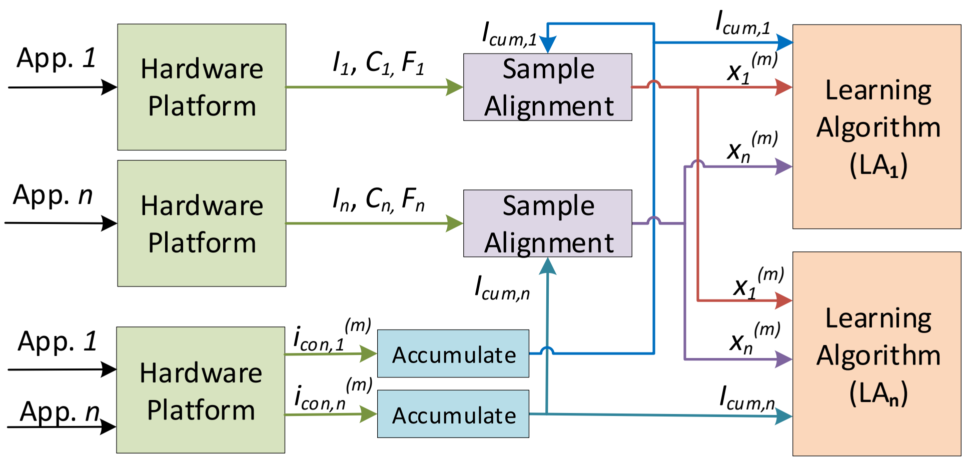


Figure 15: Training Phase

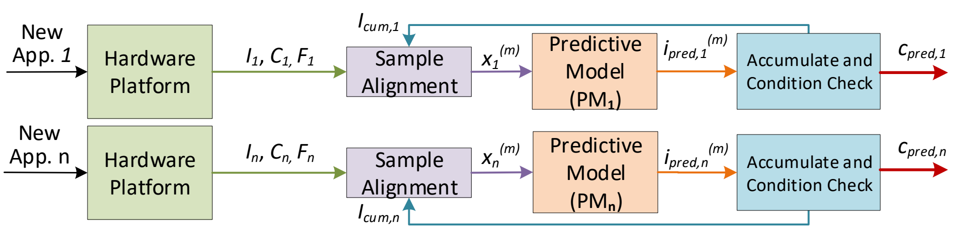


Figure 16: Prediction Phase

#### *5.1.1.1. Proposed Approach*

An overview of the proposed approach is shown in Figure 15 and Figure 16. The learning-based formulation of the run-time prediction of co-running applications problem consists of two phases: a training phase and a

Table 1: Symbol table.

| Symbols | Descriptions |
|---|---|
| $n$ | Application executed on core $n$ |
| $K_n$ | Total number of samples in app on core $n$ |
| $I_n$ | Trace of instruction per time unit for app on core $n$ |
| $C_n$ | Cycles that were required to execute $I_n$ |
| $F_n$ | Recorded features during execution of $I_n$ |
| $x_n$ | Aligned features |
| $i_{con,n}^{(m)}$ | Recorded $m$-th sample in the trace of instruction per time unit for app on core $n$ in case of interference with other apps |
| $i_{pred,n}^{(m)}$ | Predicted $m$-th sample in the trace of instruction per time unit for app on core $n$ in case of interference with other apps |
| $c_{pred,n}$ | Predicted cycles that were required to execute $i_{pred,n}^{(m)}$ |
| $I_{cum,n}$ | Accumulated value |

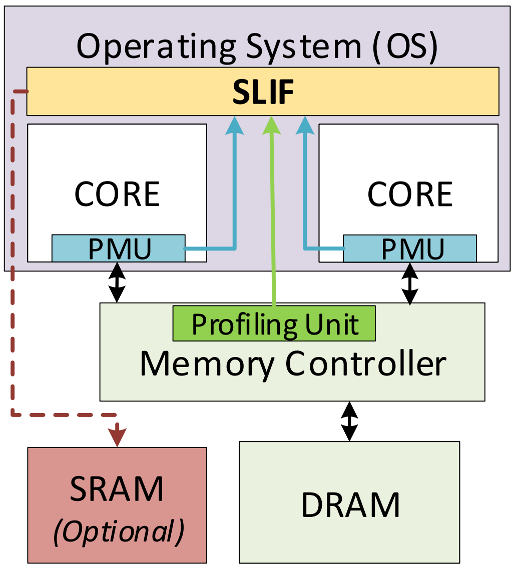


Figure 17: System overview.

prediction phase. During the training phase, applications are profiled to extract features using System Level Instrumentation Framework (SLIF) shown in Figure 17. In this scenario, features are instructions $I$, cpu-cycles $C$ and other hardware performance events $F$ of an application. These features are collected during a fixed time period $t$ from $n$ number of cores inside the hardware platform.

The profiling of applications is carried out in two different scenarios: standalone and interference. At first, all the training applications are executed in standalone and their respective features are collected. In the second scenario, the same training applications are then co-run in pairs to collect instructions $i_{con}$ in interference having $L$ number of total samples. This $i_{con}$ is then used to obtain sample aligned feature vector $X$ and learn an interference-aware run-time predictive model.

Finally, in the predictive phase, $n$ arbitrary applications are executed in the standalone scenario and profiled periodically to collect instructions $I$, cpu-cycles $C$ and other hardware performance events $F$. Features are sample aligned and fed into the model to predict the run-time of the applications in the interference scenario.

**Algorithm 1:** Sample Alignment and Interpolation functions

```
1 Function Sample_Alignment($I_n$, $C_n$, $F_n$, $i_{x,n}$, t):
2     $c_{new,n}$ = Interpolate($C_n$, $I_n$, $i_{x,n}$)
3     $i_{new,n}$ = Interpolate($I_n$, $C_n$, $c_{new,n}$ + $t$) $x_n$ = Interpolate($F_n$, $I_n$, $i_{new,n}$) - Interpolate($F_n$, $I_n$, $i_{x,n}$)
4     return $x_n$
5 Function Interpolate($T_n$, $R_n$, $q_n$):
6     Where $R_n \leftarrow [r_n^{(1)}$ ... $r_n^{(K_n)}]$ and $T_n \leftarrow [t_n^{(1)}$ ... $t_n^{(K_n)}]$
7     Find largest $\alpha$ for which $r_n^{(1)} + \ldots + r_n^{(\alpha)} \leq q_n$
8     $\gamma_n = \frac{r_n - r_n^{(\alpha)}}{r_n^{(\alpha+1)} - r_n^{(\alpha)}}$
9     $\omega_n = t_n^{(\alpha)} \times (1 - \gamma_n) + t_n^{(\alpha+1)} \times \gamma_n$ return $\omega_n$
```

### 5.1.1.2. *Sample Alignment*

Applications tend to experience delays in run-time when co-running predominantly due to interference at memory controller. Therefore, the samples of features $I$, $C$ and $F$ that are collected in standalone scenarios and feature $i_{con}$ collected in interference scenarios do not have one-to-one correspondence. This is a key challenge for sampling-based machine learning approaches as $I$, $C$ and $F$ are the inputs to the learning algorithm whereas $i_{con}$ is the output to be learned during training-phase. Furthermore, it is also important to keep track of the pace of each application progress during the prediction phase as each application in interference can have difference pace of application progress.

To solve these problems, we propose an efficient sample alignment algorithm which relies on the principle that total number of instructions remains the same irrespective of execution scenario (standalone or inference). The algorithm uses $i_{con}$ during the training-phase ($i_{pred}$ during the prediction-phase) as a feedback to keep track of the current section of the executing application in interference and re-adjust to the section of executing code which corresponds to the same number of instructions in standalone. However, the alignment will be very coarse grain if the adjustment is made at discrete level (sample-level). Therefore, we use linear interpolation (line 7 to 13 in Algorithm 1) to have fine grain alignment of features.

The interpolation function first takes an instantaneous feature value $q$ to find the largest sample number $\alpha$ in feature vector $R$ for which $r^{(1)}+\ldots+r^{(\alpha)} \leq q$. Secondly, the ratio $\gamma$ is calculated that tells the factor by which $q$ lies between the two adjacent samples that is $r^{(\alpha)}$ and $r^{(\alpha+1)}$. Finally, the value for other feature vector $T$ is calculated based on $\alpha$ and $\gamma$.

The sample alignment function (line 1 to 6 in Algorithm 1) in prediction phase first calculates the cpu-cycle $c_{new}$ of the application in standalone that corresponds to the section of code for which the number of instructions executed is $i_{pred}$ in interference. Since an applications can theoretically progress at max by period $t$ in interference scenario compared to standalone scenario. We use $c_{new} + t$ time to find the section of code for the next prediction represented by $i_{new}$. Finally, the feature vector $x$ that corresponds to the section of code between $i_{pred}$ and $i_{new}$ are computed which later are fed in the prediction model.

### 5.1.1.3. *Interference-Aware Run-time Predictor*

Algorithm 2 sketches the working of Interference-Aware Run-time Predictor in training and prediction phases. The instructions $I_n$, cpu-cycles $C_n$ and other hardware performance events $F_n$ per time $t$ unit are obtained by sampling the applications on $n$ cores in standalone scenario, each having $Kn$ samples. The instructions in interference $I_{con,n}$ per time $t$ unit, obtained by co-running the same applications in pair, is the quantity to be learned during the training-phase and therefore is required in the training-phase only.

In the training-phase, we start by initializing $I_{cum,n} = 0$. $I_{cum,n}$ basically accumulates, $i_{con,n}^{(m)}$ which is used

**Algorithm 2:** Interference-Aware Run-time Predictor

```
1  Training Phase:   input : $I_1$, ..., $I_n$ = [$i_n^{(1)}$ ... $i_n^{(K_n)}$],
              $C_1$, ..., $C_n$ = [$c_n^{(1)}$ ... $c_n^{(K_n)}$],
              $F_1$, ..., $F_n$ = [$f_n^{(1)}$ ... $f_n^{(K_n)}$],
              $I_{con,1}$, ..., $I_{con,n}$ = [$i_{con,n}^{(1)}$ ... $i_{con,n}^{(L_n)}$],
              $t$
2  foreach App $n$ do
3      $I_{cum,n}$ = 0
4  for $m$ from 1 to max($L_1$,...,$L_n$) do
5      foreach App $n$ do
6          $I_{cum,n}$ = $I_{cum,n}$ + $i_{con,n}^{(m)}$ if $m \leq Ln$ then
7              $x_n^{(m)}$ = Sample_Alignment($I_n$, $C_n$, $F_n$, $I_{cum,n}$, t)
8          else
9              $x_n^{(m)}$ = 0
10     foreach App $n$ do
11         $LA_n$([$x_1^{(m)}$ ... $x_n^{(m)}$], $i_{con,n}^{(m)}$)

12 Prediction Phase:   input : $I_1$, ..., $I_n$ = [$i_n^{(1)}$ ... $i_n^{(K_n)}$],
              $C_1$, ..., $C_n$ = [$c_n^{(1)}$ ... $c_n^{(K_n)}$],
              $F_1$, ..., $F_n$ = [$f_n^{(1)}$ ... $f_n^{(K_n)}$],
              $t$
   output: $c_{pred,n}$
13 foreach App $n$ do
14     $I_{cum,n}$ = 0 $runFlag_n$ = true
15 m = 1 $isNotDone$ = 1 $appNotFinish$ = $n$ while $isNotDone$ do
16     foreach App $n$ do
17         $x_n^{(m)}$ = Sample_Alignment($I_n$, $C_n$, $F_n$, $I_{cum,n}$, t)
18     foreach App $n$ do
19         $i_{pred,n}^{(m)}$ = $PM_j$([$x_1^{(m)}$ ... $x_n^{(m)}$])
20         $I_{cum,n}$ = $I_{cum,n}$ + $i_{pred,n}^{(m)}$
21         if $I_{cum,n} \geq i_n^{(1)} + \ldots + i_n^{(K_n)}$ and $runFlag_n$ then
22             $c_{pred,n}$ = $m \times t$ $runFlag_n$ = false $appNotFinish$ = $appNotFinish$ - 1
23         if $appNotFinish == 0$ then
24             $isNotDone$ = 0
25     $m$ = $m$ + 1;
```

as reference for sample alignment. The sample alignment $x_n^{(m)}$ is performed for each application as long as the total samples $L_n$ of $I_{con,n}$ have been aligned. Since not all application have equal number of samples and run-time in interference scenario, $x_n^{(m)}$ is set to zero in case an application have finished executing (line 10 in Algorithm 2). All the computed [$x_1^{(m)}$ ... $x_n^{(m)}$] and $i_{con,n}^{(m)}$ are then fed in the learning algorithm (*LA*) to generate a predictive model (*PM*) that learns the relationship between [$x_1^{(m)}$ ... $x_n^{(m)}$] and $i_{con,n}^{(m)}$.

### 5.1.1.4. *Evaluation Setup*

Table 2: SB-VDS Benchmark characteristics in isolation.

| Applications | Avg. IPC | Avg. Bandwidth |
|---|---|---|
| multi_ncut | 0.88 | 450 MB/s |
| disparity | 0.50 | 441 MB/s |
| tracking | 0.59 | 406 MB/s |
| mser | 0.67 | 328 MB/s |
| sift | 0.69 | 126 MB/s |
| stitch | 0.90 | 124 MB/s |

We evaluate our approach on the NXP S32V234 [21] embedded platform. The SoC features 4 ARM Cortex A53 [22] CPUs, organized into 2 clusters each having 2 cores. Each core has its own private L1 data and instruction cache whereas the 2 cores within a cluster share a unified L2 cache. We use two cores from two distinct clusters to perform our analysis. We sample the system using SLIF, which is implemented in Linux version 4.19 as a loadable kernel module, at a fix period of 200,000 cpu-cycles.

A subset of benchmarks in the San Diego Vision Benchmark Suite (SD-VBS) [23] are used to gain insight into the platform and evaluate the proposed approach. The input dataset for the benchmark applications is available in nine different sizes. Since we are interested in applications that are DRAM-bound, we use the ones with the largest input data size (named *FullHD*).

The characteristics of each benchmark included in our evaluations are summarized in Figure 2. Benchmarks are listed in decreasing order of average memory bandwidth usage when each benchmark runs in isolation on the evaluation platform. Notice that the benchmarks cover a wide range of memory bandwidth usage, ranging from 13MB/s (*texture_synthesis*) up to 450MB/s (*multi_ncut*). We have not included the *texture_synthesis* benchmarks in our evaluation given its low bandwidth requirement and therefore insignificant impact on execution time due to memory contention.

Unless otherwise noted, we use *multicut*, *mser*, *stitch* and *sift* applications for training-set, and *disparity* and *tracking* for test-set in all the experiments.

### 5.1.1.5. *Feature Engineering*

The evaluation platform, comprising of ARM Cortex A53 cores [22], has 58 measurable hardware performance events, but provides only six 32-bit hardware performance counters and a dedicated 64-bit cycle counter. This complicates matters because we can only read 7 of the total 58 hardware performance events at any given time. This limitation is overcome by re-running the identical benchmark application, each time reading a different set of six hardware performance events. Using SLIF and taking an average over 50 iterations leads in minimal variation ($0.5\%$) in observed values. As a result, the subset of measured hardware performance event values can be considered to correspond to each other as if they were read at the same instance.

Apart from hardware performance events inside the cores, the memory controller exposes a set of memory-mapped performance counters that report: (1) the number of DDR cycles elapsed; (2) the number of busy DDR cycles; (3) the total number of memory accesses in terms of read and (4) write; the total number of bytes transferred in (5) read and (6) write transactions.

We measured a total of 128 characteristics from the system, 64 from each core. However, incorporating all available features into the machine learning algorithm does not necessarily generate the best analytical model. In fact, the prediction error can become large in case of severe multicollinearity as it increases the variance of the regression coefficients, making them unstable. This issue of multicollinearity may also arise in our circumstance because different features theoretically have an impact on each other. One example of such a case is number of branches executed is related to total number of instructions executed, which on turn have relation to L1 instruction cache access.

The reduction of features is accomplished in two steps. Firstly, features reporting a value of $0$ are eliminated as their contribution to the analytical model is insignificant. We discover 26 hardware performance events with zero numeric values for all of our applications, thus they are eliminated from the feature set.

Secondly, we computed the correlation matrix using Spearman rank-order correlations of the remaining features between each other, which is a standard practice to find the level of dependency and association among all features. On top of this computed correlation matrix, we performed hierarchical clustering and manually selected a threshold by visual inspection of the dendrogram (a branch diagram that represents categories) to group our features into clusters and keep a single feature from each cluster.

For example L2 cache access, DDR busy cycles and non-cacheable external memory requests are clustered closely together. By visual inspection, we chose a threshold of 0.4 which results in selection of roughly one feature per group and reducing the overall feature set to 71.

Next, we find the subset of 28 most relevant features out of 71 with 2.8% mean absolute error by using *permutation importance* [24] technique with the Ridge Linear Regression [25] method. *Permutation importance* calculates the increase in the models prediction error after permuting the feature on the training set. A feature is "important" if changing its values raises the model error, because the model depended on the feature for the prediction.

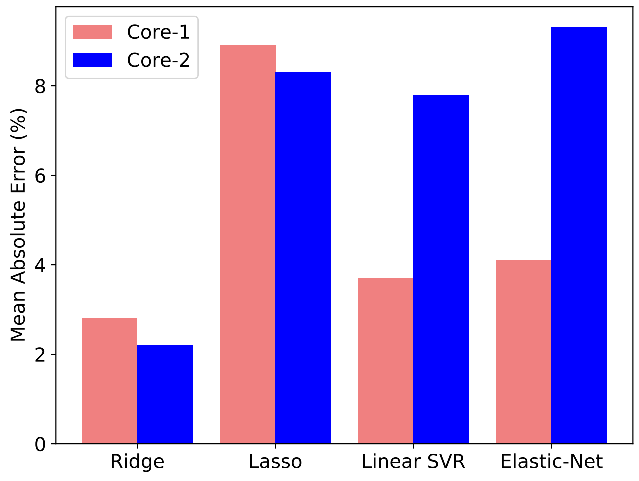


Figure 18: Comparison of mean absolute error for different machine learning models

Table 3: Comparison of measured and predicted total run-time of applications using the Ridge Linear Regression

| **Training Set** | **Test Set** | | **Standalone (ms)** | | **Interference (ms)** | | **Predicted (ms)** | |
|---|---|---|---|---|---|---|---|---|
| | *Core 1* | *Core 2* | *Core 1* | *Core 2* | *Core 1* | *Core 2* | *Core 1* | *Core 2* |
| *multicut*, *mser*, *stitch*, *sift* | *disparity* | *tracking* | 4666 | 4878 | 5000 | 5000 | 4818 (-3.6%) | 4994 (-0.1%) |
| *stitch*, *sift*, *disparity*, *tracking* | *multicut* | *mser* | 4912 | 4898 | 5000 | 5000 | 4981 (-0.4%) | 4943 (-1.1%) |
| *disparity*, *tracking*, *multicut*, *mser* | *stitch* | *sift* | 4952 | 4985 | 5000 | 5000 | 4986 (-0.3%) | 4997 (-0.1%) |

### 5.1.2. Evaluation Results and Analysis

We evaluate four different linear regression models that are Ridge [25], Lasso [26], Elastic-Net [27] and Linear Support Vector Regression (SVR) [28]. Linear SVR is basically SVR with linear kernel. Figure 18 shows mean absolute prediction error of both the cores separately for these models. Ridge method have the minimum mean absolute error of 2.8% and 2.2% for core-1 and core-2 respectively.

Table 3 shows the summary of various combination of training-set and test-set, and the corresponding run-time of the test-set in standalone scenario and interference scenario using the Ridge Linear Regression [25].

Furthermore, Table 3 also shows the predicted run-time from Table 2. When the *disparity* and *tracking* applications are executed for 4,666 ms and 4,878 ms respectively in standalone, their run-time is increased to 5,000 ms during co-execution. By using the *permutation importance* technique along with the Ridge Linear Regression and the Interference-Aware Run-time Predictor, we are able to predict a run-time with the average percentage error of -3.6% and 0.1% respectively. It can be observed that the mean absolute error of core-1 (*diparity* application) in Figure 18 is 2.8% which is less than than the average error for the same setup shown in Table 3. The reason for higher average error can be accounted for the fact that Figure 18 show prediction error of the *predictive model* only, whereas Table 3 highlights the error for the complete algorithm. Even though the average error of the *predictive model* averages out to almost zero, the sample alignment function can introduce additional errors. This can happen as predicted instructions $i_{pred}$ are used to decide for the next set of features to be fed into the *predictive model*. The next sample is selected on an assumption that the program will pace by the instructions equal to the period $t$ which is not always true and thus can cause additional errors.

## 5.2. Resource Allocation and Timing Constraints

For timing-analysis in the absence of interference, the proposed approach for timing analysis is divided in three phases: runtime performance trace extraction, timing and resource allocation analysis and model annotation.

The first phase focuses on extracting runtime information from actual program execution. The most common method to determine program timing is by measurements, usually known as Measurement-Based Timing Analysis (MBTA). The approach is to use the actual hardware as the model for analysis. The code is deployed and executed in the hardware, providing different inputs, and the actual execution is measured usually by instrumenting the source code at different points. As outlined in previous sections, to obtain statistical validity, multiple executions of the code must be done, for the same set of inputs, to capture variations in execution time.

Compared with static analysis, profiling measurements have the advantage of being performed on the actual hardware, which avoids the need to construct a hardware model and, hence, reducing the overall cost of deriving the estimates. Another advantage is the access to multiple performance counters from the hardware [29, 30], from which metrics can be derived and can complement the performance measurements.

The state of the art of profiling tools count with several approaches that use measurement-based techniques and determine the execution time of those paths by executing the application on the target hardware platform (or by cycle-accurate simulators) to collect execution traces. These traces are a sequence of time-stamped values that show which parts of the application has been executed. These tools produce performance metrics for each part of the executed code and, by using the performance data and knowledge of the code structure, they allow to estimate the worst-case execution time of the program. Example of tools include Rapitime [31] in the commercial domain, the UpScale Analyzer in the research domain [32] and the Extrae package from the Barcelona Supercomputing Center [33].

In our approach we conduct the analysis at the `Runnable` level. We also divide this analysis in two parts: offline and runtime analysis. The former is applied when we use profiling information, i.e., performance traces are obtained from runtime execution and are stored in a file/database for later processing. The latter, described in deliverable D4.2 [34] is focused on analyzing the performance at runtime (ideally with low overhead) and subsequently adapt the execution at runtime, e.g. by rescheduling tasks.

For the offline analysis, Extrae is used to monitor and to extract performance traces from OpenMP tasks. The target code is instrumented with Extrae API instructions, which is able to extract performance traces from the program execution at various levels of a program. For instance, it is possible to instrument the application with custom events to measure a specific code parcel. More importantly, and since our target applications are OpenMP-based, it can automatically extract performance counters information for OpenMP-related features even without changing the source code. This allows the analysis to be performed at the OpenMP task level and it provides a fine-grain time analysis for a given `Runnable`.

After the code is compiled and executed - or simulated - in a target hardware, runtime traces are obtained as part of the output of the Extrae tool. The output format of Extrae is the Paraver format, a csv-like format where each line represent an event in the program execution. The output file of Extrae is defined as exemplified in Figure 19, where the first line represents the format of the trace. Each event is represented by the record type `2`, it contains information about the cpu and thread in which the code parcel executed, and the timestamp of the event. Then, a list of `events` are listed in a key-value format, which includes the event code that triggered the trace and a list of zero or more performance counters. The example of Figure 19 uses a custom event (71830003) with the value 1 to indicate the beginning of a specific code parcel to be measured and the same custom event with value 0 to indicate the end of that code parcel. This means that in the example, the same code parcel is executed twice. The execution time is given by the difference between the final and initial events, while the performance counter value is obtained in the final event since Extrae resets the counters at the initial event.

```
record:cpu:app:task:thread:event:value (:event:value)*
2:3:1:1:3:5890126:71830003:1:42000050:0:42000059:0:42000000:0:42000002:0
2:3:1:1:3:6088987:71830003:0:42000050:3291:42000059:17775:42000000:223:42000002:826
2:3:1:1:3:6089479:71830003:1:42000050:1528:42000059:1255:42000000:13:42000002:11
2:3:1:1:3:6089775:71830003:0:42000050:1522:42000059:1021:42000000:8:42000002:4
```

Figure 19: Excerpt of a paraver output file generated by the Extrae tool. Three performance counters exist in the example: total instructions (event 4200050), total cycles (event 4200059) and number of L1 and L2 cache misses (4200000 and 4200002 respectively))

Timing analysis is performed with the performance traces obtained by profiling application executions instrumented with Extrae directives. This analysis is based on WCET for each `Runnable`. Currently we are performing this analysis only considering `Runnables` with fully sequential code. This means that only `Tasks` have concurrency with `Runnables` potentially executing concurrently, but the code of a `Runnable` does not contain OpenMP directives. The WCET, in this case, will be the maximum time spent by a `Runnable`. The approach, however, is already being developed taking into account `Runnables` with OpenMP directives. Here, the WCET scenario have to take into account, for each Runnable execution, the sum of all the performances of each OpenMP task executed by the `Runnable`, where the OpenMP task performance is the sum of all task parts pertaining to the task. We consider the sum of all performances because, despite the OpenMP task might be executing concurrently, the worst case scenario would be to execute all the tasks sequentially.

We are currently using a basic analysis over the results, which includes the calculation of common metrics, such as average, standard deviation, minimum and maximum values. These metrics are calculated per performance counter.

As the analysis is performed for timing and resource usage, our approach extracts information from the following performance counters, available in most common hardwares:

- Total instructions retired;
- Total of clock cycles;
- Number of L1 cache accesses/misses;
- Number of L2 cache accesses/misses;
- Number of last level cache accesses/misses.

There are other performance counters to be considered in the analysis. However, these are still being studied to observe their relevance in the approach. The considered performance counters include:

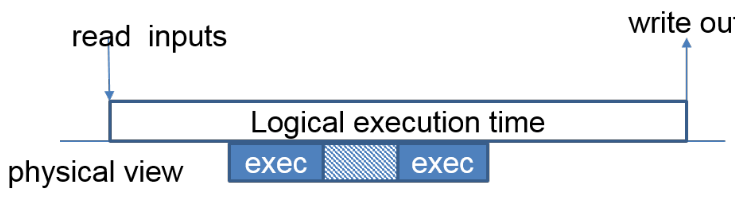
- Branch prediction accesses/misses;
- Instruction TLB load accesses/misses;
- Data TLB load/store accesses/misses.

The results of the analysis can then be stored in the model of the application and later used for model adaptation or program generation reasoning. The model annotation with timing analysis is detailed in deliverable D1.3 [5].

## 5.3. The Logical Execution Time Paradigm

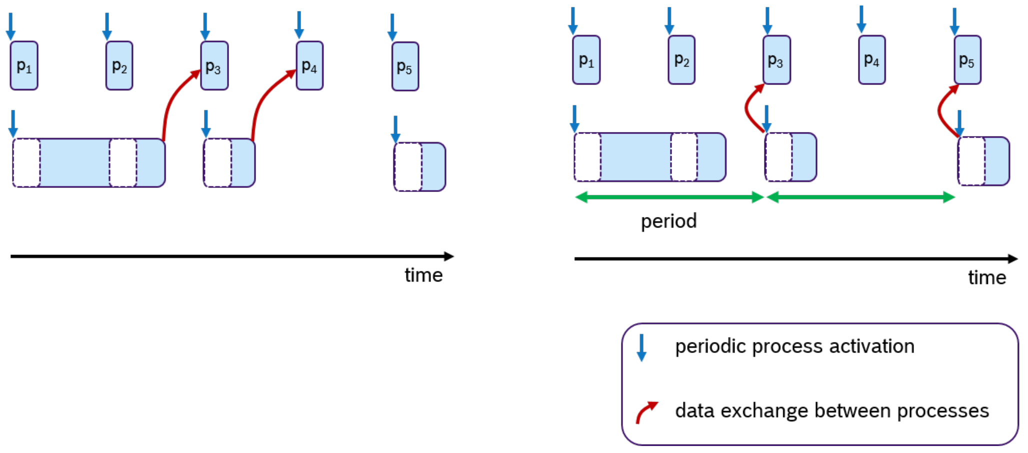
Logical Execution Time (LET) is a structured approach for reasoning about dependent tasks and their temporal interactions. It was introduced with the time-triggered programming language Giotto [35]. It is a real-time programming concept which ensures temporal determinism by decoupling computation and communication. The problem with an unconstrained communication method, i.e, allowing tasks to read and write arbitrarily is non-determinism due to “execution jitter”. The result is highly dependent on possible interferences of other tasks executing within a tasks activation interval (say from its release to the end of its period). The effects of this jitter becomes more prominent in event chains, leading to large variations in end-to-end delays. The LET model is robust against these jitters by enforcing strict communication rules. With the LET model, tasks always read data at the beginning of the *activation* interval and write data at the end of the activation interval as depicted in Figure 20.

Figure 20: LET: The observed output is independent of the time a task executes in its LET interval



Using LET, the observable temporal behavior of a task is independent from its physical execution. That is irrespective of the exact time a task executes within its execution interval, the result will be always available only at the end of its activation interval. LET also ensures portability, i.e, the same behavior of the tasks when migrated to another hardware (core), integrability on addition of newer software and interoperability, which is verified by deterministic communication. An example is shown in Figure 21. Two tasks with different periods are exchanging data. The timepoint when this data is written is at the end of the processing time. In the default case (left side), the process p3 and p4 receive the update. At the right side of the figure, the same scenario is shown with LET semantics. Here, the data is communicated only at period boundaries. In this case, the lower process communicates at the end of the period, so that always process p3 and p5 receive the new data.

Figure 21: Data communication without and with Logical Execution Time paradigm



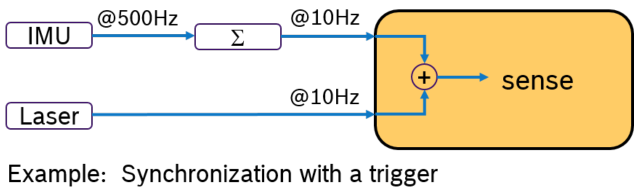
With LET, the end-to-end latencies in case of synchronous stimuli is always equal the sum of the periods of the tasks involved in the chain. However, with asynchronous stimuli it may happen that each task in the effect chain executes as early as possible in its activation window but the data arrives just after it begins execution (meaning it is operating on an older value of the data). Thus the newer data is consumed only one time period later. The same scenario could occur with every pair of tasks in the chain. Eventually, the worst case latency in the case of such asynchronous arrivals is *twice the sum of the periods of all the tasks in the chain*.

LET thus leads to longer latencies in event chains. But on the other hand, with LET, there is no need for complex synchronization mechanisms to handle race conditions or priority inversions, given its well-defined semantics.

With the shift from classic real-time systems on micro-controllers towards micro-processor-based integration platforms handling communication on shared variables across the whole system is no longer a viable option. To ease decouple the software development of the underlying hardware platform and to reuse existing software and their provided data in the development of new functionality middleware architectures like ROS and AUTOSAR Adaptive support decoupling the computation from communication. To use LET like semantics in these systems we need to tackle the execution semantics of the underlying middleware.

For ROS2, the first steps towards a determinstic activation and LET-like execution semantics are described in [36]. The *rclc* [37] executor for ROS2 allows different trigger conditions based on the availability of input data, e.g. the process can be activated when all data inputs are available. The executor also allows to make local copies of all input data when triggered. Since the synchronization of outputs is not yet implemented, LET-like semantics could be achieved with high priority gate keepers as depicted in Figure 22. The $\sum$ element would only forward the data every 50 interations to ensure that the right data age is forwarded to the sense component.

Figure 22: Synchronization of input data



As part of MS2, LET semantics have been investigated to ease the timing analysis, and the interference and timing analysis techniques presented in the earlier parts of this section provide the mandatory support for the necessary timing control. It would be in general possible to extend the LET concept to ROS2 with the presented executor. However, it is not certain if the AMALTHEA SLG should be extended to support LET. This could have limited benefit as the use-cases are based on data-driven activation patterns to a large extent. Instead the main focus in the coming milestone will be on extending the deadline management support provided by the techniques presented in the previous section, supporting LET-like execution to the level that is mandated by the use-cases.

## 5.4. Analysis Techniques to support DNN workloads

FPGA-based platforms represent an attractive alternative for realizing time-predictable embedded computing systems with hardware acceleration. They allow deploying energy-efficient accelerators with competitive performance [38] with respect to GPUs. FPGA accelerators are also more suitable for timing predictability because they are often characterized by a very regular, clock-level behavior and allow for an explicit control of the memory traffic they generate, being the hardware design to be deployed on FPGA under the full control of the designer. Even if commercial FPGA accelerators are typically distributed as closed-source modules, the direct access to the hardware design allows precisely profiling and monitoring the bus traffic they generate, hence achieving an accurate characterization and supervision of their execution behavior that would be simply impossible with other platforms. We focused our efforts on FPGA-based acceleration of DNNs by means of the Xilinx DPU accelerator, which to the best of our records is the most mature solution of this kind on the market at the time of writing. An extensive profiling campaign focused on the Xilinx Zynq Ultrascale+ platform has been conducted to study the execution behavior of the DPU by means of a DPU-specific FPGA profiler we developed. Based on the profiling, we will design an execution model for the DPU and a response-time analysis based on such model. The proposed experimental profiling is based on state-of-the-art DNNs for Advanced Driver Assistance Systems (ADAS).

## 5.4.1. Background and System Architecture

### *5.4.1.1. FPGA SoC architecture*

A typical FPGA SoC architecture combines a *Processing System* (PS), including one or more processors (generally ARM-based), with a *Field-Programmable Gate Array* (FPGA) fabric in a single device. The processors in PS execute software tasks (SW-tasks). The FPGA fabric can be programmed to host custom hardware devices such as *hardware accelerators*. Figure 23 illustrates a typical FPGA SoC architecture. Typically, computations are controlled by SW-tasks, which can in turn activate the hardware accelerators when required. The hardware accelerators and the processors can communicate through a shared off-chip DRAM memory or an On-Chip Memory (OCM). The DRAM memory is accessed by a DRAM memory controller, embedded in PS, and shared between the PS and the FPGA subsystems. This is crucial to enable high-performance, asynchronous data communication among hardware accelerators and processors. The communications between the PS and the FPGA subsystems are allowed by two interfaces: the FPGA-PS interface and the PS-FPGA interface. The FPGA-PS interface exports a set of high-throughput ports allowing the access of the hardware accelerators to the devices in PS (e.g., DRAM memory controller, OCM, peripherals). Conversely, the PS-FPGA interface exports a set of ports leveraged by the processors to access and manage the hardware accelerators. The data movement relies on the AMBA AXI bus, which is the de-facto standard for communications in modern SoCs [39]. The bus traffic within the PS subsystem (e.g., originated from the processors or the devices in the FPGA fabric and directed to the DRAM controller and other peripherals in PS) is managed by a multi-level AXI-based PS interconnect.

### *5.4.1.2. Frameworks for DNN acceleration on FPGA SoCs*

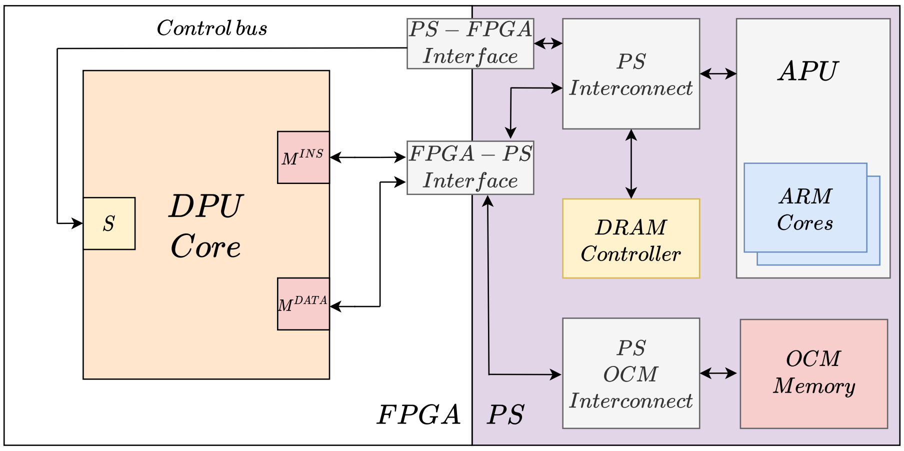
To date, most of the development and deployment efforts for DNN models rely on powerful and energy-intensive GPU-based systems. The parameters of such DNN models, such as activations, weights, and biases are typically represented as 16-bit or 32-bit floating-point data. Due to the intrinsic differences in the architecture of GPU and FPGA platforms, most of the common neural networks deployed for execution on GPUs are not compatible out-of-the-box with FPGA SoC platforms. Indeed, while in GPU-based systems neural networks can be executed through calls to parallel computation APIs (e.g., the CUDA API in NVIDIA platforms), FPGA SoC platforms require a specific *hardware accelerator* deployed into the FPGA fabric for the execution of DNN models. The academia and the industry proposed several frameworks to cope with FPGA-based acceleration of DNNs, which combine conversion tools and specialized hardware accelerators for the deployment of DNNs on FPGA SoC platforms [40, 41, 42, 43]. Among such a variety of frameworks, to the best of our records the most mature one is the Vitis AI framework [43] by Xilinx.

**The Vitis AI framework** Vitis AI provides a collection of tools, libraries, and hardware accelerator IP cores for the conversion and execution of GPU-like floating-point DNN models upon Xilinx FPGA SoC platforms. Vitis AI supports DNN models deployed through the most mainstream neural network frameworks, such as Caffe, Tensorflow, and Pytorch. The execution of the DNN layers in Vitis AI relies on the *Deep learning Processing Unit* (DPU) core. The DPU is a hardware accelerator to be deployed into the FPGA fabric and optimized for the execution of convolutional DNNs. A brief description of the DPU core is provided in Section 5.4.1.4. The DPU engine is configured by a control software application running in the PS of the FPGA SoC platform. Control software applications can be developed by leveraging the libraries provided by the Vitis AI framework.

### 5.4.1.3. *DNN working flow with Xilinx Vitis AI*

As for any framework for FPGA SoC platforms, a DNN model must undertake some process in Vitis AI before being ready to be executed on the target platform. In particular, in Vitis AI this process involves a quantization phase and a compilation phase. Such phases are performed once offline, typically on a powerful workstation. The output of such steps is a set of instructions and data for the DPU core.

Figure 23: The sample architecture of a Xilinx FPGA SoC platform deploying the DPU core in the FPGA fabric.



**Preparation and quantization**

The input to the Vitis AI framework is a pre-trained (GPU-like) floating-point neural network model. As of today, the Vitis AI framework supports only convolutional DNN models operating on images. As a first step, the data structure of the DNN model must be made compatible with the features of the DPU core. This means that the whole DNN data structures must be converted from floating-point to 8-bit fixed-point data. This process is called *quantization* and is performed by the Vitis AI quantization tool. Quantization is supported by a calibration phase, which makes use of a subset of the training images to minimize accuracy losses [43]. The output of the quantization process is the quantized DNN model. The quantized DNN model is then provided to the Vitis AI compiler tool that parses the quantized DNN model and creates an intermediate representation of the DNN. Such an intermediate representation is then mapped to a sequence of instructions for the DPU. The Vitis AI tool places the generated instructions into a `.xmodel` file. At this point, the Vitis AI software application that is going to run on the target FPGA SoC platform can be built. Its development relies on the Vitis AI software libraries, which contains a set of APIs for the whole operation of the DPU core. The application is then cross-compiled for the target platforms using the Xilinx cross-compilation tools.

**Running on the target FPGA SoC platform**

Vitis AI applications are distributed as Vitis AI images, which are based on a Petalinux[1] image target for the FPGA SoC platform under analysis. Once launched, a Vitis AI application starts the configuration phase. This phase includes the preparation of the memory buffers in the central DRAM memory for the execution of the DPU. The application loads in DRAM memory the instructions and the weights provided by the `.xmodel` file. The DPU core is then configured by the Vitis AI software application. Once the configuration is done, the Vitis AI software application triggers the DPU to start the execution. At this point, the DPU is autonomous in the execution of the DNN model. The Vitis AI software task is suspended until the execution of the DPU completes (the DPU will notify the processors once the execution is done by means of an interrupt signal). The DPU

[1]Petalinux is a Linux distribution based on Yocto and targeted to run on Zynq Ultrascale+ platforms.

fetches the instructions, the weights, and the input image to be processed by the DNN from the DRAM buffer prepared by the Vitis AI software application. The fetching of instructions and data is performed in parallel by the DPU core leveraging multiple memory ports.

#### 5.4.1.4. *DPU core hardware accelerator*

The DPU disposes of direct access to the memory and the PS through multiple AXI interfaces. Figure 23 illustrates a sample FPGA SoC architecture including a DPU core. With reference to the figure, $S$ is the AXI-lite subordinate interface leveraged by the SW-tasks running in the PS for the configuration of the DPU core; $M^{\text{INS}}$ is the AXI manager interface used by the DPU core for fetching the DNN instructions to be executed from the DRAM memory; and $M^{\text{DATA}}$ is the AXI manager interface leveraged by the DPU core for reading and writing data from/to the DRAM memory. This latter interface is used to fetch the parameters (mainly weights) of the various DNN layers, to fetch the input image to be processed, to read/write intermediate results generated during DNN inference, and eventually writing the final DNN outputs. Unfortunately, the DPU core is distributed as a closed-source IP block only. To the best of records, no detailed information about its internal behavior is publicly available.

### 5.4.2. Profiling the DPU

Although the DPU is a proprietary accelerator distributed as a closed-source module, it was possible to understand several aspects of its execution behavior by developing a custom hardware module deployed on the FPGA fabric to perform advanced profiling of the DPU execution and memory access patterns. It is worth stressing the fact that such an advanced analysis was possible only thanks to the hardware programmability of FPGA SoC — the same analysis would be very difficult, if not impossible, to be conducted on commercial GPU-based SoC platforms.

To obtain precise measurements, we developed a multi-channel hardware profiler that we integrated in the Xilinx Vitis AI stock hardware design (i.e., the default configuration for IPs and connections setup in the default hardware design provided with the Vitis AI framework). We connected our hardware profiler to probe all of the interfaces and ports of the DPU, such as the AXI interfaces and the interrupt line, and accurately keep track of the interaction of the DPU with the DRAM memory with the fine granularity of each clock beat. It is connected in parallel with the stock connections, hence ensuring that the execution of the DPU is not perturbed by the profiler.

Our hardware profiler is capable of recording the behavior of the DPU at different levels: **(i)** the DPU bus activity for a given DNN model, i.e., the number of read/write transactions issued over time, their burst length, and the corresponding amount of exchanged data, **(ii)** the structure of execution phases of the DPU core, **(iii)** the execution time of the phases and the variability of the total inference time and **(iv)** the identification of pipelining among the execution phases (i.e., the time in which multiple execution phases are overlapped).

To consider a representative profiling campaign, we focused on popular state-of-the-art DNN models that are part of modern ADAS applications: they include **(i)** a lane detection DNN based on a VpgNet model [44], **(ii)** a plate detection DNN model, **(iii)** an object detection DNN based on a Yolov3 model [45], **(iv)** an object detection DNN based on an SSD model [46], and **(v)** a pedestrian detection DNN based on an SSD model. For each of such DNN models, we recorded 1000 DPU executions (to perform inference of the network) by means of our hardware profiler. The latest version (v1.3.2) at the time of writing of Xilinx Vitis AI was used together with the stock Vitis AI configuration based on Petalinux [47] as provided by Xilinx. The profiling was conducted on a Xilinx ZCU102 development board equipped with the Xilinx Zynq Ultrascale+ FPGA SoC. The considered DPU architecture is the default one (version 3.3) provided with Vitis AI for the Zynq Ultrascale+. The DPU clock is left to the default value of 330Mhz. All the DNN models under analysis operate on images. Following the stock Vitis AI working flow, each execution involves the analysis of a single image and produces an output result. In the following, we refer to a single execution as a *DPU job*. During our profiling campaign, the input image for each DPU job was randomly picked from the CityScapes dataset for ADAS applications [48].

### 5.4.2.1. *Profiling the DPU bus activity*

Table 4 reports the bus activity recorded with our hardware profiler. The columns of Table 4 report the per-job bus activity of the DPU, in order **(1)** number of read transactions for instruction fetching (i.e., issued via the $M^{\text{INS}}$ port), **(2)** data words fetched corresponding to instructions, **(3)** number of read transactions for data read (i.e., issued via the $M^{\text{DATA}}$ port) **(4)** data words fetched corresponding to data read, **(5)** number of write transactions for data write **(6)** data words written to the memory. Such results are reported for each of the DNN models under analysis.

Table 4: Profiled per-job bus activity of the DPU core for the DNN models for ADAS applications under analysis.

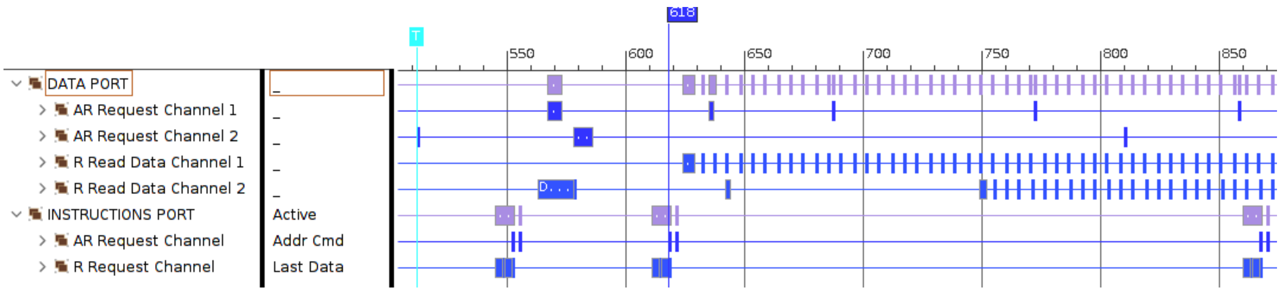
| **DNN bus activity** | **Num Instr trans** | **Data instr words (size)** | **Num Read trans** | **Data read words (size)** | **Num write trans** | **Data write words (size)** |
|---|---|---|---|---|---|---|
| **Lane Detect** | 17186 | 68744 (275KB) | 91939 | 1179184 (17.99MB) | 48314 | 424350 (6.48MB) |
| **Plate Detect** | 2347 | 9388 (37KB) | 7607 | 83027 (1.27MB) | 246 | 16960 (0.3MB) |
| **Plate Num** | 9872 | 39488 (154KB) | 53327 | 579962 (8.85MB) | 6075 | 48908 (0.7MB) |
| **Object Detect (Yolo)** | 16060 | 64240 (251KB) | 84410 | 1011665 (15.44MB) | 25457 | 540416 (8.24MB) |
| **Object Detect (SSD)** | 9920 | 39680 (155KB) | 68943 | 725208 (11.06MB) | 4705 | 554510 (8.46MB) |
| **Pedestrian Detect (SSD)** | 11655 | 46620 (182KB) | 56597 | 639932 (9.76MB) | 5505 | 506096 (7.72MB) |

Note that, even though we performed 1000 executions, the amount of interactions with the memory changes very little from one execution to another: we recorded changes in the order of less than $0.1\%$ only. The bus activity of the DPU also resulted to be independent of the input image. This first observation provides a hint on the *high predictability* of the execution of the DPU core. Indeed, memory access and memory contention are the major sources of unpredictability for FPGA-based accelerators — a predictable bus activity has hence a strong positive impact on the predictability of the whole system [49].

Besides the amount of data exchanged, other important features to characterize the performance and predictability of the DPU core are how the data are exchanged, in other words, the parallelism of the manager ports $M^{\text{INS}}$ and $M^{\text{DATA}}$ (i.e., the number of transactions each port can have pending, also called number of outstanding transactions) and the burst length of the bus transactions. The parallelism of ports $M^{\text{INS}}$ and $M^{\text{DATA}}$ influences the execution time of the DPU. Typically, the higher the parallelism of the port, the higher the data throughput. Unfortunately, these numbers are not publicly disclosed by Xilinx.

Thus, we developed a specific functionality in our hardware profiler to retrieve these characteristics. We experimentally found that the parallelism of the port $M^{\text{DATA}}$ is of 14 read outstanding transactions and 7 write outstanding transactions. Differently, the parallelism of the $M^{\text{INS}}$ port is of 2 outstanding read transactions (it is worth remembering that $M^{\text{INS}}$ is used by the DPU only for reading instructions). Concerning the burst length of transactions, we found that the burst length of the transactions issued on $M^{\text{DATA}}$ varies during the DPU execution, from the minimum of one word to the maximum of 256 word for both read and write transactions – these correspond to the limits defined by the AXI standard. Differently, the burst length of the transactions issued on the $M^{\text{INS}}$ port is fixed and equal to 4 words.

Figure 24: A particular of the hardware waveform track captured using a Xilinx System ILA recording the bus behaviour of the DPU core when accelerating the Yolov3 object detection DNN for ADAS applications considered in this work.



### 5.4.2.2. *Measuring the DPU execution phases*

From the profiling we noted how the DPU has three concurrent bus activities: read instructions, read data, and write data. Each of such activities is associated with an execution time contributing to the total execution time of a DPU job (also called *inference time*). Four execution phases for the DPU have been identified: **(1)** *Read instructions phase*: The instructions are fetched from the DRAM memory through the $M^{\text{INS}}$ port. **(2)** *Read data phase:* The DNN model data (weights, biases, activations) and the input image are fetched from the DRAM memory through the $M^{\text{DATA}}$ port. **(3)** *Write data phase:* The results of the computation are written to the DRAM memory through the $M^{\text{DATA}}$ port. **(4)** *Elaboration phase:* It is defined as the time when the DPU has not yet completed the job and no activity on the bus is performed (i.e., the hardware accelerator is making progress by only computing). The response time of each DPU phase has been measured leveraging our hardware profiler. For instance, to measure the execution time of the read instructions phase, our hardware profiler tracks the time the DPU is active on the $M^{\text{INS}}$ port, considering served and pending transactions. The hardware profiler stops the profiling when the DPU interrupt is raised, locally storing the profiled values that can be read by software as standard memory-mapped registers. Table 5 reports the minimum, average, and maximum recorded times for each of the DPU phases, accompanied by the measured total inference time, for the 1000 executions under analysis.

Table 5: Measured per-job execution times of the DPU phases and total measured per-job inference times.

| **DNN model inference times (ms)** | | **Instrc fetch** | **Read data** | **Write data** | **Pure Elaboration** | **Total Inference** |
|---|---|---|---|---|---|---|
| **Lane Detect (VpgNet)** | min | 2.23 | 6.15 | 4.17 | 0.01 | 7.09 |
| | avg | 2.26 | 6.81 | 4.19 | 0.04 | 7.1 |
| | max | 2.29 | 7.11 | 4.20 | 0.58 | 7.12 |
| **Plate Detect** | min | 0.29 | 0.51 | 0.12 | 0.01 | 0.73 |
| | avg | 0.3 | 0.71 | 0.13 | 0.02 | 0.74 |
| | max | 0.31 | 0.74 | 0.14 | 0.2 | 0.75 |
| **Plate Num** | min | 1.27 | 2.46 | 0.56 | 0.01 | 3.05 |
| | avg | 1.28 | 2.99 | 0.57 | 0.02 | 3.06 |
| | max | 1.32 | 3.01 | 0.58 | 0.2 | 3.07 |
| **Object Detection (Yolov3 ADAS)** | min | 2.28 | 7.49 | 3.46 | 0.01 | 7.99 |
| | avg | 2.31 | 7.93 | 3.47 | 0.1 | 8.01 |
| | max | 2.35 | 8.01 | 3.48 | 0.23 | 8.02 |
| **Object Detection (SSD ADAS)** | min | 1.52 | 6.92 | 3.74 | 0.01 | 8.39 |
| | avg | 1.54 | 8.3 | 3.76 | 0.1 | 8.4 |
| | max | 1.56 | 8.4 | 3.77 | 0.7 | 8.41 |
| **Pedestrian Detection (SSD ADAS)** | min | 1.60 | 7.95 | 3.47 | 0.01 | 9.10 |
| | avg | 1.62 | 8.87 | 3.48 | 0.1 | 9.11 |
| | max | 1.64 | 9.11 | 3.49 | 0.6 | 9.12 |

#### 5.4.2.3. *Observations*

This extensive profiling campaign allows us to make some observations, which will be leveraged to design an appropriate model of the DPU with the purpose of bounding response times during DNN acceleration. The observations follow:

**The software-DPU interactions are limited to the DPU configuration:** we developed a specific functionality in our hardware profiler to record the interactions between the Vitis AI control SW-task and the DPU during execution. We detected that the software-hardware interactions of the tested DNNs are limited to the configuration phase of the DPU. This means that no software-hardware interactions are present during the execution phase (inference) of the DNN model on the DPU.

**The bus activity of the DPU is constant and independent from the specific job:** from the results reported in Table 4, it emerges that the bus activity of the DPU does not vary job by job. This suggests that the DPU implements the very same and predictable behavior given the structure of the network, the resolution of the input image, and the size of the outputs. This also means that the amount of instructions fetched, read data, and write data is fixed and known a priori.

**The DPU exhibits high parallelism in the execution of phases:** by checking the results of Table 5 it is possible to notice how, for all the tested DNNs, the results for the read data phase and the total inference time are very similar. Also, the sum of the read instruction phase, read data phase, and write data phase is way higher than the total measured inference time. Guided by this observation, we developed a specific functionality in our hardware profiler to measure the parallel execution of **(i)** the instruction and write data phases of the DPU, with **(ii)** the DPU read data phase. The results are reported in Figure 25. The results showed how at least 97-98% of the execution of the former phases is overlapped with the one of the latter. This has been confirmed by also visually inspecting the bus activity using the Xilinx Integrated Logic Analyzer (ILA) — an excerpt of the waveform track of the bus signals is reported in Figure 24.

From these considerations, it is possible to conclude the following: **(i)** the DPU executes multiple phases in parallel, hence suggesting an internal pipelined implementation (as it is common for many FPGA-based accelerators); and **(ii)** the read data phase provides the dominant contribution to the total inference time, i.e., the instruction fetching phase and the write data phase are almost completely hidden due to the pipelined behavior. These conclusions motivate the definition of a series-parallel model for the DPU.

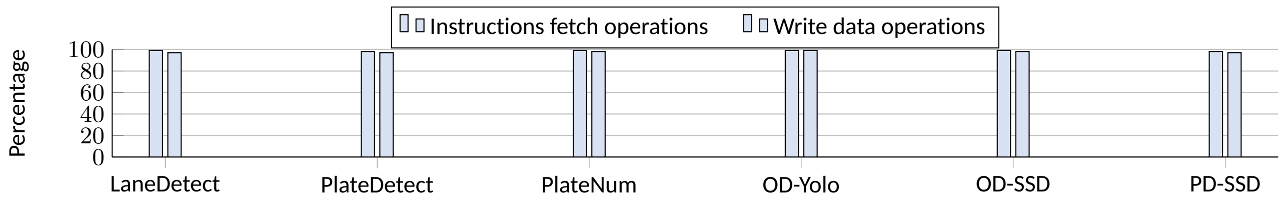


Figure 25: Minimum measured percentage of the total inference time for which DPU instructions operations and write operations are overlapped with the DPU read data operations.

**Limited fluctuations of response times:** the results reported in Table 5 finally show how the fluctuations of the response times are quite limited. They are mainly attributed to the delays experienced at the memory controller to access the external DRAM.

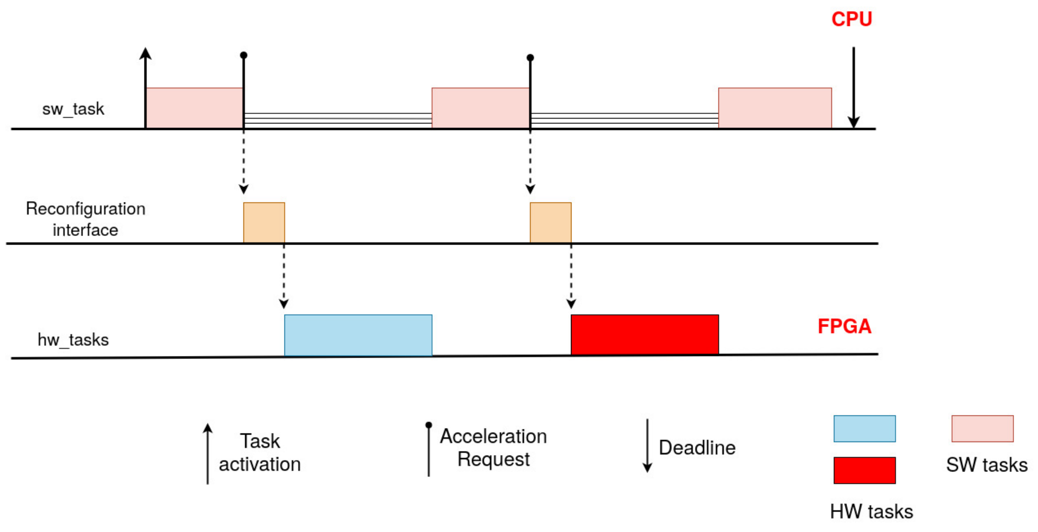
## 5.5. Heterogeneity and Design Time Optimization for FPGA Offloading

DART is a tool that fully automates the FPGA design flow for a real-time, dynamic partially reconfigurable (DPR) co-designed system that comprises both software and hardware components. The tool targets the Zynq

7-series and Ultrascale+ FPGA-based SoCs by Xilinx. DART fully automates the partitioning, floorplanning, and implementation (routing and bitstream generation) phases of the DPR flow.

In a typical co-design flow, for implementing real-time systems on a DPR-enabled FPGA-based SoC platform, the ensemble of tasks of the system are first classified into (*i*) software tasks, and (*ii*) hardware tasks. Software tasks are regular software activities executed on the CPUs available on the SoC, while hardware tasks are hardware description language (HDL) implementations of computationally-intensive functions to be offloaded on the FPGA. Figure 26 depicts a sample of a programming model supported by DART, where a software task invokes two hardware acceleration requests at different times. After each invocation the software task self-suspends until the acceleration request is finished. Upon receiving each acceleration request, the reconfiguration interface loads the bitstreams onto the FPGA and the accelerator starts to execute. At the completion of the acceleration, the software task is notified.

Figure 26: An example of a DPR-based HW-SW codesign system.



At the beginning of the design flow, the hardware tasks are logically partitioned and assigned to one or more reconfigurable regions (RRs) defined in the total FPGA area. Each RR can host more than one hardware module, which will be executed in a time-multiplexed manner. DART adopts a partitioning based on the timing behavior of both software and hardware tasks of the system. The timing requirements are mapped into timing-related constraints to ensure that the partitioning does not violate the timing requirements of the software and hardware tasks.

Figure 27 represents a block diagram of DART's internal organization, including its inputs and outputs. DART takes as inputs: (*i*) the HDL sources of hardware tasks; (*ii*) the timing requirements of software tasks; and (*iii*) the description of the FPGA internal architecture. DART's outputs include: (*i*) the bitstreams of the design; (*ii*) the necessary files for the FRED runtime environment[2]. Inside DART, the partitioning and floorplanning are combined into a single mixed integer linear programming (MILP)-based multi-objective optimization problem. The two DPR design steps were fused into a single optimization problem by converting the input timing requirements and floorplanning related constraints into a series of MILP constraints. For instance, if two hardware tasks are partitioned on the same RR, then the RR must contain enough FPGA resources to host both of them in mutual exclusion, hence requiring the maximum of the FPGA resources requested by each hardware task for each resource type (CLB, DRAM, DSP). At the same time, the two hardware tasks must be capable of tolerating the delays that can be originated due to contention of the RR, i.e., in the worst case, one of the two must wait for the entire time the other occupies the RR before being able to be dynamically configured on it.

The partitioning step inside the optimization engine of DART produces the total number of RRs and the mapping between hardware tasks and RRs. This step takes as input the timing requirements of both the software

[2]FRED is described in Deliverable [34].

and hardware tasks, as well as some FPGA-related attributes as an output. This block also produces the FRED related output files. The partitioning design step is implemented together with the floorplanning step. Floorplanning refers to geometrically mapping the RRs (produced by the partitioning step) on the physical FPGA fabric and it is known to be a non-trivial task due to the vast design space to be explored under a set of constraints. The DPR floorplanning-related constraints can be classified as (*i*) general constraints; (*ii*) structural constraints; and (*iii*) resource-related constraints. General constraints are used to encode the properties of hardware tasks and RRs as well as to describe the relationship between the two. Structural constraints are used to ensure the integrity and consistency of the RRs, in order not to violate the technological and structural floorplanning. Finally, resource constraints are used to pose restrictions on the resource consumption of hardware tasks and the resource availability inside RRs. The outputs of a floorplanning step are descriptions of the placements of the RRs on the FPGA fabric.

Finally, DART fully automates the placement, routing, and bitstream generation design steps in the DPR design flow by utilizing a set of auto-generated Tcl scripts that interact with, and command, the vendor design tools. In addition to the FPGA bistreams of the design, DART also produces the necessary files for the FRED runtime. The runtime requires the mapping between hardware tasks and RRs, as well as additional information such as the Linux device tree of the accelerator to execute the design. These files are produced inside DART after the completion of the optimization step.

Besides the DART tool, we also provide an IP design template including a standardized set of design scripts, called *DART IPs*, to ease the design process using the proposed DPR-based design flow. DART IPs is also a repository with a list of ready-to-use hardware IPs and static parts for DART. Appendix A.1 and A.2 include additional description of DART IPs.

Figure 28, page 42, presents an example of block-based of the FRED static part with three partitions. Besides the partitions, this design also includes the PS part, interconnect logic, and input and output decouplers for each partition. During the FPGA reconfiguration process, the behavior of the area under reconfiguration is undefined since its logic resources may be in an inconsistent state. In particular, logic resources may produce transient signals while being programmed. These signals can cause troublesome spurious transactions in other modules, such as the AXI interconnects or the system interrupt controller. To protect the system for these events, each reconfigurable slot is protected by a partial reconfiguration decoupler (denoted as PR decoupler in Figure 28), a Xilinx's library IP that binds the wires of the slot interface to safe logic values during the reconfiguration process [50]. Fred-Linux controls each decoupler through a single control register mapped into the system address space through an AXI-Lite slave interface.

Table 6 shows the resource utilization report including three memcpy IPs, considering the board PYNQ-Z1 (xc7z020clg400-1).

More details can be found about DART in this paper [51] and at its repository[3]. DART IPs is available at its repository[4].

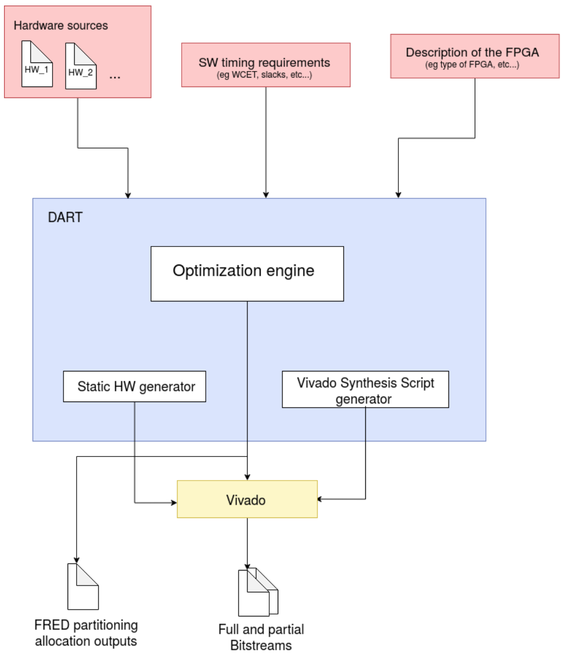
## 5.6. Summary

This section has described the fundamental problem of providing quality of service guarantees in heterogeneous systems, and the need for composable mechanisms that promote freedom from interference. Several techniques have been presented to achieve this goal, and the ongoing work for controlling memory interference has been outlined. The approaches have in common that they operate on profiling information received by the Extrae profilier, providing a unified interface with the other sections of this deliverable. Furthermore, the analysis that promotes timing analysis is done at the granularity of the Task Dependency Graph.

Fundamentally, the Logical Execution Time paradigm promotes many of the features that are requested from a timing perspective. This paradigm has been described, with its strenghts in predictable end-to-end latencies

[3]DART repository `https://repo.retis.sssup.it/pr_tool`.
[4]DART IPs repository `https://gitlab.retis.santannapisa.it/a.amory/dart_ips/`.

Figure 27: Block diagram of the DART flow.


Hardware sources
HW_1
HW_2
...
SW timing requirements
(eg WCET, slacks, etc...)
Description of the FPGA
(eg type of FPGA, etc...)
DART
Optimization engine
Static HW generator
Vivado Synthesis Script generator
Vivado
FRED partitioning allocation outputs
Full and partial Bitstreams

| Site Type | Used | Fixed | Available | Util% |
|---|---|---|---|---|
| **Slice LUTs** | 6009 | 0 | 53200 | 11.30 |
| **LUT as Logic** | 5548 | 0 | 53200 | 10.43 |
| **LUT as Memory** | 461 | 0 | 17400 | 2.65 |
| **LUT as Distributed RAM** | 10 | 0 | | |
| **LUT as Shift Register** | 451 | 0 | | |
| **Slice Registers** | 7986 | 0 | 106400 | 7.51 |
| **Register as Flip Flop** | 7986 | 0 | 106400 | 7.51 |
| **Register as Latch** | 0 | 0 | 106400 | 0.00 |
| **F7 Muxes** | 0 | 0 | 26600 | 0.00 |
| **F8 Muxes** | 0 | 0 | 13300 | 0.00 |

Table 6: FPGA resources used by the design.

without complex synchronization mechanisms to ensure operation on the expected data. The main research question being worked on is the integration of the LET paradigm with the consideration needed for the AMPERE system, utilizing middleware architectures like ROS and AUTOSAR Adaptive. It has been shown how LET-like semantics could be achieved with high priority gate keepers, a process that will be further refined in the ROS executor.

Additionally, the FPGA has been characterized from a timing predictable perspective, its main feature being the very regular, clock level behaviour and predictable memory access patterns. In the context of neural networks, a careful characterization has been presented that allows introspection of closed-source IP blocks, using a custom multi-channel hardware profiler. It was shown that the amount of memory interactions is stable over all measured parameters, and the only source of variance in the execution time can be attributed to interference at the memory subsystem level – motivating the need for the freedom from interference techniques. Furthermore, the DART tool has been presented, which allows fully automated FPGA bitstream generation for runtime reconfiguration. DART allows timing related constraints to be used to ensure that partitioning does not violate the real-time properties of the system.

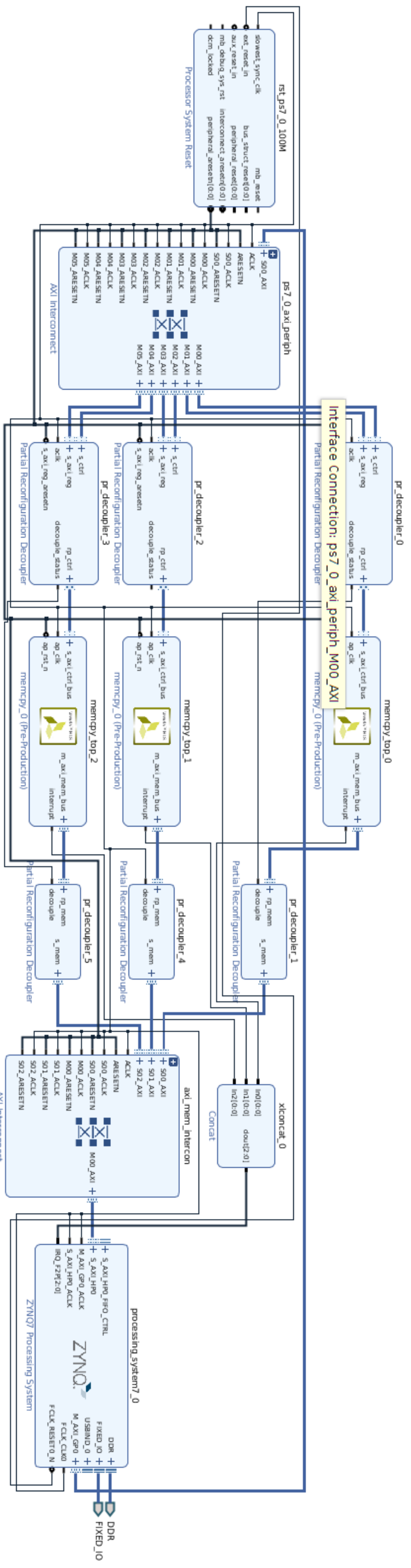


Figure 28: FRED static part block design with three partitions.

# 6. Conclusions

This deliverable has presented the ongoing work, up to Milestone 2, of the tasks in Work Package 3. The goal of Milestone 2 has been to report on the single-criterion a) software resilient techniques for parallel execution, b) energy optimisation framework, and c) predictable execution models. The meaning of single-criterion optimization is that each individual component performs optimization with respect to its area of focus, without considering cross-component implications, which is the target of the coming Milestone 3. For each of the areas, the report has outlined several complementary techniques that aim to fulfill this goal:

We have shown two different and complementary approaches for software resilient techniques. The first technique is based on the annotation of modeled tasks to indicate that the computation should be duplicated to provide redundancy and cross-validation of critical tasks. The necessary adaptations in the modeling language and programming model were outlined, and provides a compiler-assisted approach for resiliency. In addition to this, the second approach presents a programmer-in-the-loop technique, in which an external observer monitors the state of a task, by continously evaluating it against pre-defined predicates.

For energy-efficiency, this deliverable has provided a presentation and evaluation, showing how the model information and profiling information is used to derive energy characteristics about the system, and presented non-intrusive and intrusive techniques to optimize the system for energy-efficiency. As part of the single-criterion optimization, the main optimization goal has been to at low-overhead identify the most efficient DVFS operating point for each modeled task. It has outlined several interesting directions for future optimizations that are enabled by co-optimizing energy-efficiency and the remaining criteria. Furthermore, it has provided an investigation into the use of transprecision, an intrusive technique, to provide significant improvenets in energy-efficiency for tasks that are robust to lower precision arithmetics.

In the context of predictable execution models, the report has presented the main challenges for timing-predictability on heterogeneous systems, as used in AMPERE, and techniques to counteract them. The extension of Logical Execution Time semantics to advanced communication middleware such as AUTOSAR Adaptive and ROS provide the necessary infrastructure to provide predictable execution times in significantly parallel execution environments. Furthermore, a detailed characterization of FPGA timing and execution semantics, in particular for DNN networks, have been performed, showing that the FPGA is a very predictable accelerator well suited for real-time acceleration. In connection to this, profile-based worst-case execution time and cross-task interference techniques have been outlined and are in ongoing development.

Taken together, the single-criterion optimization goal of the milestone has been fulfilled for each component. For each component, future directions for the work have been presented, which will be implemented as part of the next milestone. In preparation of this, all components have been aligned to operate on the same data structures and principles. The main component is the Task Dependency Graph which provides the necessary input for each component, including performance counter traces provided by Extrae, which is the main input for the analyses. This coherency paves the way for the coming integration and multi-criteria optimization framework, the next step for the work of Work Package 3.

# 7. Acronyms and Abbreviations

| | |
|---|---|
| CPU | Central Processing Unit |
| D | Deliverable |
| DAG | Direct Acyclic Graph |
| DPR | Dynamic Partial Reconfiguration |
| DSML | Domain Specific Modeling Language |
| DVFS | Dynamic Voltage Frequency Scaling |
| FPGA | Field-Programmable Gate Array |
| GPU | Graphics Processing Unit |
| HDL | Hardware Description Language |
| HPC | High-Performance Computing |
| MDE | Model-Driven Engineering |
| MILP | Mixed Integer Linear Programming |
| MS | Milestone |
| OCM | On-Chip Memory |
| ODAS | Obstacle Detection Avoidance System |
| OS | Operating System |
| PPM | Parallel Programming Model |
| RR | Reconfigurable Region |
| SLG | Synthetic Load Generator |
| T | Task |
| TDG | Task Dependency Graph |
| WP | Work Package |

# 8. References

[1] BSC, "Extrae," 2021, https://tools.bsc.es/extrae.

[2] AMPERE, "Grant Agreement," 2018.

[3] ——, "D3.1: Multi-criteria optimisations requirements," 2020.

[4] Eclipse APP4MC, "SLG Linux," https://git.eclipse.org/c/app4mc/org.eclipse.app4mc.addon.transformation.git, 2021.

[5] AMPERE, "D1.3: First release of the meta model-driven abstraction release," 2021.

[6] ——, "D2.2: First release of the meta parallel programming abstraction and the single-criterion performance-aware optimisations," 2021.

[7] ——, "D6.2: Refined AMPEREecosystem interfaces andintegration plan," 2021.

[8] R. Ferrer, S. Royuela, D. Caballero, A. Duran, X. Martorell, and E. Ayguadé, "Mercurium: Design decisions for a s2s compiler," in *Cetus Users and Compiler Infastructure Workshop in conjunction with PACT*, vol. 2011, 2011.

[9] GNU, "GOMP Project," https://gcc.gnu.org/projects/gomp/, 2021.

[10] S. Che, M. Boyer, J. Meng, D. Tarjan, J. W. Sheaffer, S.-H. Lee, and K. Skadron, "Rodinia: A benchmark suite for heterogeneous computing," in *2009 IEEE international symposium on workload characterization (IISWC)*. Ieee, 2009, pp. 44–54.

[11] A. Balsini, L. Pannocchi, and T. Cucinotta, "Modeling and simulation of power consumption and execution times for real-time tasks on embedded heterogeneous architectures," *ACM SIGBED Review*, vol. 16, no. 3, pp. 51–56, 2019.

[12] RI5CY: RISC-V core. [Online]. Available: https://github.com/pulp-platform/riscv

[13] G. Tagliavini, S. Mach, D. Rossi, A. Marongiu, and L. Benini, "Design and evaluation of smallfloat simd extensions to the RISC-V ISA," in *2019 Design, Automation Test in Europe Conference Exhibition (DATE)*, 2019, pp. 654–657.

[14] R. Cavicchioli, N. Capodieci, and M. Bertogna, "Memory interference characterization between CPU cores and integrated GPUs in mixed-criticality platforms," in *22nd IEEE International Conference on Emerging Technologies and Factory Automation (ETFA)*, 2017.

[15] L. Abeni, L. Palopoli, G. Lipari, and J. Walpole, "Analysis of a reservation-based feedback scheduler," in *IEEE Real-Time Systems Symposium (RTSS)*, 2002.

[16] X. Zheng, P. Ravikumar, L. John, and A. Gerstlauer, "Learning-based analytical cross-platform performance prediction," in *International Conference on Embedded Computer Systems: Architectures, Modeling, and Simulation (SAMOS)*, 2015.

[17] X. Zheng, L. K. John, and A. Gerstlauer, "Accurate Phase-Level Cross-Platform Power and Performance Estimation," in *ACM/EDAC/IEEE Design Automation Conference (DAC)*, 2016.

[18] X. Zheng, H. Vikalo, S. Song, L. K. John, and A. Gerstlauer, "Sampling-based binary-level cross-platform performance estimation," in *Design, Automation Test in Europe Conference Exhibition (DATE)*, 2017.

[19] P. H. Saeed, Wurst and Dasari, "Machine Learning Based Cross-Platform Runtime," in *Euromicro Conference on Real-Time Systems (ECRTS), Waters Workshop*, 2019.

[20] C. Mendis, S. Amarasinghe, and M. Carbin, "Ithemal: Accurate, Portable and Fast Basic Block Throughput Estimation using Deep Neural Networks," 2018.

[21] "S32V Vision and Sensor Fusion Evaluation Board," https://www.nxp.com/design/development-boards/automotive-development-platforms/s32v-mpu-platforms/s32v-vision-and-sensor-fusion-evaluation-board:SBC-S32V234.

[22] "ARM® Cortex® -A53 MPCore Processor - Technical Reference Manual," https://static.docs.arm.com/ddi0500/f/DDI0500.pdf.

[23] S. K. Venkata, I. Ahn, D. Jeon, A. Gupta, C. Louie, S. Garcia, S. Belongie, and M. B. Taylor, "SD-VBS: The San Diego Vision Benchmark Suite," in *IEEE International Symposium on Workload Characterization (IISWC)*, 2009.

[24] A. Fisher, C. Rudin, and F. Dominici, "Model class reliance: Variable importance measures for any machine learning model class, from the "rashomon" perspective," 01 2018.

[25] D. Marquardt and R. Snee, "Ridge Regression in Practice," *American Statistician - AMER STATIST*, 1975.

[26] R. Tibshirani, "Regression shrinkage and selection via the lasso," *Journal of the royal statistical society series b-methodological*, 1996.

[27] H. Zou and T. Hastie, "Regularization and variable selection via the elastic net," *Journal of the Royal Statistical Society Series B*, 2005.

[28] D. Basak, S. Pal, and D. Patranabis, "Support Vector Regression," *Neural Information Processing – Letters and Reviews*, 2007.

[29] P. J. Mucci, S. Browne, C. Deane, and G. Ho, "PAPI: A portable interface to hardware performance counters," in *Proceedings of the department of defense HPCMP users group conference*, vol. 710, 1999.

[30] L. Uhsadel, A. Georges, and I. Verbauwhede, "Exploiting hardware performance counters," in *2008 5th Workshop on Fault Diagnosis and Tolerance in Cryptography*. IEEE, 2008, pp. 59–67.

[31] RapiTime, https://www.rapitasystems.com/products/rapitime.

[32] U. Analyser, https://github.com/upscale-sdk/UpScale/tree/master/analysis_flow/Analyzer.

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

[34] AMPERE, "D4.2: Independent run-time energy support, and predictability, segregation and resilience mechanisms," 2021.

[35] C. M. Kirsch and A. Sokolova, *The Logical Execution Time Paradigm*. Berlin, Heidelberg: Springer Berlin Heidelberg, 2012, pp. 103–120.

[36] J. Staschulat, I. Lütkebohle, and R. Lange, "The rclc executor: Domain-specific deterministic scheduling mechanisms for ros applications on microcontrollers: work-in-progress," in *2020 International Conference on Embedded Software (EMSOFT)*, 2020, pp. 18–19.

[37] "Ros2 - rclc executor repository." [Online]. Available: https://index.ros.org/p/rclc/

[38] M. Qasaimeh, K. Denolf, J. Lo, K. Vissers, J. Zambreno, and P. H. Jones, "Comparing energy efficiency of CPU, GPU and FPGA implementations for vision kernels," in *2019 IEEE International Conference on Embedded Software and Systems (ICESS)*, 2019, pp. 1–8.

[39] *AMBA® AXI™ and ACE™ Protocol Specification*, ARM, aRM IHI 0022D.

[40] Y. Umuroglu, N. J. Fraser, G. Gambardella, M. Blott, P. Leong, M. Jahre, and K. Vissers, "FINN: A framework for fast, scalable binarized neural network inference," 2017.

[41] *CHaiDNN official github.*, Xilinx, https://github.com/Xilinx/chaidnn.

[42] H. Sharma, J. Park, D. Mahajan, E. Amaro, J. K. Kim, C. Shao, A. Mishra, and H. Esmaeilzadeh, "From high-level deep neural models to FPGAs," in *2016 49th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*. IEEE, 2016, pp. 1–12.

[43] *Vitis AI User Guide*, Xilinx, uG1414v1.3.

[44] S. Lee, J. Kim, J. S. Yoon, S. Shin, O. Bailo, N. Kim, T.-H. Lee, H. S. Hong, S.-H. Han, and I. S. Kweon, "Vpgnet: Vanishing point guided network for lane and road marking detection and recognition," in *2017 IEEE International Conference on Computer Vision (ICCV)*, 2017, pp. 1965–1973.

[45] J. Redmon, S. Divvala, R. Girshick, and A. Farhadi, "You only look once: Unified, real-time object detection," in *2016 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, 2016, pp. 779–788.

[46] W. Liu, D. Anguelov, D. Erhan, C. Szegedy, S. Reed, C.-Y. Fu, and A. C. Berg, "Ssd: Single shot multibox detector," in *Computer Vision – ECCV 2016*, B. Leibe, J. Matas, N. Sebe, and M. Welling, Eds. Cham: Springer International Publishing, 2016, pp. 21–37.

[47] *Petalinux Tools Documentation*, Xilinx, uG1144.

[48] M. Cordts, M. Omran, S. Ramos, T. Rehfeld, M. Enzweiler, R. Benenson, U. Franke, S. Roth, and B. Schiele, "The cityscapes dataset for semantic urban scene understanding," in *Proc. of the IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, 2016.

[49] F. Restuccia, M. Pagani, A. Biondi, M. Marinoni, and G. Buttazzo, "Is your bus arbiter really fair? restoring fairness in AXI interconnects for FPGA SoCs," *ACM Transactions on Embedded Computing Systems (TECS)*, vol. 18, no. 5s, p. 51, 2019.

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

[51] B. Seyoum, M. Pagani, A. Biondi, and G. Buttazzo, "Automating the design flow under dynamic partial reconfiguration for hardware-software co-design in FPGA SoC," in *Proceedings of the ACM Symposium on Applied Computing (SAC)*, 2021, pp. 481–490.

# A. Appendix

The following additional textual resources are provided in this appendix:

1. A user manual for *DART IPs* for IP design (Section A.1);
2. A user manual for *DART IPs* for Static Part design (Section A.2);

## A.1. Guidelines and Structure for Hardware IP Design

This section details how to use DART IPs resources to ease the design of DART and FRED compatible hardware IPs for FPGA offloading. All IPs should follow these guidelines and this structure in order to reuse the scripts and minimize the integration issues with DART and FRED. The rest of this section show the IP guidelines and the structure for the **two kinds of IPs: HLS and RTL**.

### A.1.1. HLS IPs

HLS IPs are those created from *vivado_hls*, based on C source code. These IPs use a different synthesis script and have a different directory structure, presented next.

#### A.1.1.1. *HLS IP structure*

All HLS-based IPs must have the following directory structure:

```
./
├─ LICENSE
├─ readme.md
├─ hw
│  ├─ build.tcl
│  ├─ src
│  │  ├─ memcpy.cpp
│  │  └─ ...
│  └─ tb
│     ├─ tb.cpp
│     └─ ...
└─ sw
   ├─ Makefile
   └─ src
      ├─ memcpy.cpp
      └─ ...
```

- the *readme.md* file documents the IP, including usage/perfomance/power reports, configuration parameters, authors, etc;
- the file *hw/build.tcl* is a script synthesis for Vivado (for Verilog or VHDL sources) or for Vivado HLS (for cpp sources). This script refers to the common synthesis script located at the *scripts* directory, at same level of the IP directory;
- the *sw* directory must have an example FRED-enabled application to test the IP;
- the *sw/Makefile* compiles the example application;

#### A.1.1.2. *Running synthesis for the HLS IPs*

Go to the directory *ips/ip-name/hw* and run:

```
$ vivado_hls build.tcl
```

The generated IP is located in the directory *ips/ip-name/hw/ip-name/solution_0/impl/ip*.

### A.1.2. RTL IPs

RTL IPs are those created from *vivado*, based on VHDL or Verilog source code. Their directory structure is presented next.

#### A.1.2.1. *RTL IP structure*

All RTL-based IPs must have the following directory structure:

```
./
├── build.tcl
├── LICENSE
├── README.md
├── hw
│   ├── bd
│   ├── hdl
│   │   └── my_module.vhd
│   ├── ips
│   ├── sim
│   │   └── tb.vhd
│   └── xdc
│       └── constraint.xdc
├── vivado
└── sw
    ├── Makefile
    └── src
        ├── test_app.cpp
        └── ...
```

- the *build.tcl* script is set to rerun the synthesis to create a vivado design;
- use the *readme.md* to document the design including, for example, usage/perfomance/power reports, configuration parameters, authors;
- in the case of an IP design, the directory *hw/bd* is typically empty since IPs are usually described in RTL or HLS, not with BD;
- the IP description files (.vhd and/or .v) must be placed in the *hw/hdl* directory;
- if there is a testbench, this must be placed in the *hw/sim* directory. VHDL, Verilog, and SystemVerilog are accepted as testbench languages;
- if there are constraint files, these must be placed in the *hw/xdc* directory;
- a IP might be composed of other sub-IPs. In this case, they must be linked in the *hw/ips* directory. Just place symbolic links to point to other IPs directories;
- the vivado directory must be empty. This is where the vivado design is saved;
- the *sw* directory must have an example FRED-enabled application to test the IP;
- the *sw/Makefile* compiles the example application;

#### A.1.2.2. Running synthesis for the RTL IPs

Go to the directory *ips/ip-name/hw* and run:

```
$ vivado -mode batch -source build.tcl
```

The generated IP is located in the directory *ips/ip-name/vivado/ip-name.runs/impl_1/*.

## A.2. Guidelines and structure for the static part of the design

The static part of the FPGA design is the part that does not suffer DPR. All static parts must follow these guidelines and this structure in order to reuse the scripts and minimize the integration issues with DART and FRED.

### A.2.1. Structure

All static parts must have the following directory structure:

```
./
├─build.tcl
├─LICENSE
├─README.md
├─hw
│ ├─bd
│ │ └─design_1.tcl
│ ├─hdl
│ ├─ips
│ ├─sim
│ └─xdc
└─vivado
```

- the *build.tcl* script is set to rerun the synthesis to create a vivado design;
- use the *readme.md* to document the design including, for example, usage/perfomance/power reports;
- the static part is typically based on a block design, thus, the BD file must be placed in the *hw/bd* directory;
- since the static part is based on block design, the *hw/hdl* directory, where hardware description files (.vhd and/or .v) are saved, must be empty;
- if there is a testbench, this must be placed in the *hw/sim* directory;
- if there are constraint files, these must be placed in the *hw/xdc* directory;
- the IPs under DPR must be linked in the *hw/ips* directory. The default IP is the memcpy IP. Just replace the symbolic links to point to other IPs directories;
- the vivado directory must be empty. This is where the vivado design is generated.

### A.2.2. Running synthesis for the static part

Go to the directory *static/static-name/* and run:

```
$ vivado -mode batch -source build.tcl
```

The generated design is located in the *vivado* directory. The generated DCP file is located in the directory *static/static-name/vivado/proj-name/proj-name.runs/synth_1/proj-name_wrapper.dcp*.