SD1 “ARCHITECTURE DEFINITION AND EVALUATION PLAN FOR LEGATO’S HARDWARE, TOOLBOX AND APPLICATIONS”

Version 2

Document Information

<table>
<thead>
<tr>
<th>Contract Number</th>
<th>780681</th>
</tr>
</thead>
<tbody>
<tr>
<td>Project Website</td>
<td><a href="https://legato-project.eu/">https://legato-project.eu/</a></td>
</tr>
<tr>
<td>Contractual Deadline</td>
<td>31. August 2018</td>
</tr>
<tr>
<td>Dissemination Level</td>
<td>Public</td>
</tr>
<tr>
<td>Nature</td>
<td>Report</td>
</tr>
<tr>
<td>Author</td>
<td>Osman Unsal (BSC)</td>
</tr>
<tr>
<td>Contributors</td>
<td>Marcelo Pasin (UNINE), Pascal Felber (UNINE), Christian Göttel (UNINE), Isabelly Rocha (UNINE), Jens Hagemeyer (UNIBI), Nils Kucza (UNIBI), Micha vor dem Berge (CHR), Gunnar Billung-Meyer (CHR), Michaela Körber (CHR), Miquel Pericàs (CHALMERS), Christof Fetzer (TUD), Do Le Quoc (TUD), Yoav Etsion (Technion), Tobias Becker (Maxeler), Amani Al-Mekhlafi (HZI), Hans Salomonsson(MI), Xavier Martorell (BSC), Raúl de la Cruz (BSC), Leonardo Bautista (BSC), Behzad Salami (BSC), Paul Carpenter (BSC)</td>
</tr>
<tr>
<td>Reviewers</td>
<td>Pascal Felber (UNINE), Isabelly Rocha (UNINE), Pedro Trancoso (CHALMERS), Christof Fetzer (TUD)</td>
</tr>
</tbody>
</table>
The LEGaTO project has received funding from the European Union’s Horizon 2020 research and innovation programme under the Grant Agreement No 780681.
# Change Log

<table>
<thead>
<tr>
<th>Version</th>
<th>Description of Change</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Initial compilation from different WP folders</td>
</tr>
<tr>
<td>2</td>
<td>Split Frontend chapter into subsections, assigned contributors</td>
</tr>
<tr>
<td>3</td>
<td>Split Applications chapter into subsections, assigned contributors, fixed small errors in introduction</td>
</tr>
<tr>
<td>4</td>
<td>Split Backend chapter into subsections, assigned contributors</td>
</tr>
<tr>
<td>5</td>
<td>First integrated version</td>
</tr>
<tr>
<td>6</td>
<td>Second integrated version after self-review by section authors</td>
</tr>
<tr>
<td>7</td>
<td>Third integrated version after internal review</td>
</tr>
<tr>
<td>8</td>
<td>Final version for submission</td>
</tr>
<tr>
<td>9</td>
<td>Edits to address reviewer comments from M9 review</td>
</tr>
</tbody>
</table>
Index

1 Executive Summary .................................................. 9

2 Introduction .............................................................. 10
  2.1 The LEGaTO Software Stack at a Glance .................. 11
  2.2 Achievements During the Work Period .................... 13

3 Specification of Application Optimization and Evaluation Plans .... 16
  3.1 Executive Summary .............................................. 16
  3.2 Introduction .................................................... 16
  3.3 Smart Home ..................................................... 18
    3.3.1 Application Description ................................ 18
    3.3.2 Requirements ............................................. 19
    3.3.3 Computational Characteristics, Code and Data Structures 20
    3.3.4 Metrics and Optimization Goals ....................... 21
  3.4 Smart City ...................................................... 22
    3.4.1 Application Description ................................ 23
    3.4.2 Requirements ............................................. 26
    3.4.3 Computational Characteristics, Code and Data Structures 26
    3.4.4 Metrics and Optimization Goals ....................... 31
  3.5 Machine Learning ............................................... 33
    3.5.1 Application Description ................................ 33
    3.5.2 Requirements ............................................. 36
    3.5.3 Computational Characteristics, Code and Data Structures 36
    3.5.4 Metrics and Optimization Goals ....................... 37
  3.6 Infection Research ............................................. 38
    3.6.1 Application Description ................................ 38
    3.6.2 Requirements ............................................. 39
    3.6.3 Computational Characteristics, Code and Data Structures 40
    3.6.4 Metrics and Optimization Goals ....................... 43
  3.7 Secure IoT Gateway ............................................ 44
    3.7.1 Application Description ................................ 45
    3.7.2 Requirements ............................................. 46
    3.7.3 Computational Characteristics, Code and Data Structures 46
    3.7.4 Metrics and Optimization Goals ....................... 50
  3.8 Summary .......................................................... 51

4 Definition/Design of the Front-End Toolbox .......................... 54
  4.1 Executive Summary .............................................. 54
  4.2 Introduction .................................................... 54
  4.3 Software Architecture Specification .......................... 55
    4.3.1 The OmpSs Programming Model ......................... 55
4.3.2 Multicomputer (OmpSs@Cluster) ............................................. 58
4.3.3 Energy-efficiency ................................................................. 60
4.3.4 Fault-tolerance ................................................................. 60
4.3.5 DFiant: A High-level Hardware Description Language ............. 60
4.3.6 Maxeler MaxCompiler ......................................................... 62
4.4 Programming Model .............................................................. 65
4.4.1 Basic OmpSs .................................................................... 65
4.4.2 Annotations for Mapping ................................................... 66
4.4.3 Annotations for Fault-tolerance ......................................... 67
4.4.4 Annotations for Security .................................................... 68
4.4.5 Annotations for XiTAO ........................................................ 68
4.4.6 Annotations for Reconfiguration ...................................... 70
4.4.7 Annotating Design Constraints in DFiant .......................... 71
4.4.8 Annotations for Maxeler .................................................... 72
4.4.9 Integrated Development Environment ................................ 74
4.4.10 Efficient Support for Irregular Data Structures .................. 75
4.4.11 Directive-based Checkpointing ......................................... 75
4.4.12 Static Kernel Identification ............................................... 76
4.4.13 Trusted Tasks ................................................................. 77
4.4.14 Compiler-based Protection Against Security Threats .......... 78
4.4.15 Compiler-based Protection Against Software Faults .......... 81
4.5 OpenStack Extensions for a Task-based Programming Model .... 83
4.5.1 Logical-level Orchestration of Hardware Resources ............ 83
4.5.2 Node Composition and Dynamic (Re-)configuration of the Communication Infrastructure ................................. 83
4.6 Conclusion ........................................................................ 84

5 Definition/Design of back-end Runtime System ......................... 85
5.1 Executive Summary ................................................................. 85
5.2 Introduction ..................................................................... 85
5.3 Task-based Energy Model ...................................................... 86
5.3.1 Energy-related Metrics ..................................................... 86
5.3.2 Energy Model Customized for LEGaTO ........................... 87
5.4 Resource Management .......................................................... 88
5.4.1 Redfish API .................................................................. 89
5.4.2 Node Composition ........................................................... 91
5.4.3 Dynamic Reconfiguration of Communication Infrastructure at Runtime ......................................................... 91
5.4.4 OpenStack Middleware .................................................... 92
5.5 The LEGaTO Runtime Library ................................................. 94
5.5.1 Hardware Topology Discovery ......................................... 94
5.5.2 GPU Components ............................................................ 94
5.5.3 FPGA Components .......................................................... 94
5.5.4 CPU components ............................................................. 95
5.5.5 Maxeler Run-time Components ...................................... 95
5.5.6 Heterogeneous Scheduling .............................................. 96
5.5.7 Topology-aware Scheduling ............................................ 97
5.5.8 ML-support for Heterogeneous Scheduling ................. 98
5.5.9 Distributed Scheduling .................................................... 99

5.6 Conclusion ....................................................................... 100

5.7 Future Work ..................................................................... 101

6 Deployment and System Characterization .................................... 102
6.1 Deployment and Deployment Planning .................................... 102
6.2 System Characterization .......................................................... 104
6.3 Sample Application ............................................................... 105

7 Execution of the Research Activities and Project Milestones .......... 106
7.1 Research Activities ................................................................. 106
7.2 Project Milestones ................................................................. 107

8 Deployment and Operations .......................................................... 108
8.1 Deployment and System Characterization ............................... 108
8.2 Sample Application ............................................................... 109

9 Conceptualization of the Back-end Runtime System ................. 110
9.1 Conceptualization ................................................................. 110

10 Back-end Runtime System ........................................................... 111
10.1 Conceptualization ................................................................. 111

11 Conclusion and Future Work ..................................................... 112
11.1 Conclusion ..................................................................... 112
11.2 Future Work ................................................................... 113

Appendix A: LEGaTO Runtime Library Design .................................... 114
Appendix B: Sample Application .................................................... 116
The LEGaTO Stack

WP5 and the LEGaTO Stack
Smart mirror prototype (UI left / object detection and face recognition right)
Scheme of the smart mirror in regards to the three base components
Face recognition pipeline structure.
Speech recognition pipeline structure.
Conceptual sketch of the urban-scale air quality forecast system.
Levels of parallelism in Alya code. From left to right: 0) Initial mesh in sequential mode, 1) extra-node parallelism (by means of MPI), 2) intra-node parallelism between chips and cores (by means of either MPI, OpenMP or taskification), and finally, 3) data-level parallelism (by means of vectorization or accelerator ISAs).
Vectorization principle used in Alya to allow data-level parallelism. Element attributes of the geometrical mesh are sorted by element type (e.g., TRI03, QUA04) and packed in large nodal property arrays of size VECTOR_SIZE * NUM_VECTORS.
Mesh used for the baseline test. Left: vertical cut. Right: horizontal cut.
Wind streamlines and vorticities of the baseline test after 100 iterations.
Pixel-wise semantic segmentation of a crossing in Zurich from the cityscape dataset
DL model is getting increasingly larger and more complex
Pruning of a DL model by removing weights and/or neurons
Feed forward neural network.
Convolution is very common in DL computer vision tasks. The filter between the two layers are swiped over the input image to create the latter image. Source: [23]
Typical workflow of a biomarker discovery project
 auc of features combination
Comparison of connections with and without the Secure IoT Gateway
Overview of all IoT Gateway components
Possible IoT Bridge hardware
The overall software toolchain for the LEGaTO platform.
OmpSs@CUDA compilation environment
OmpSs@OpenCL compilation environment
OmpSs@FPGA compilation environment, based on autoVivado

The four.osf/five.osf OmpSs@FPGA compilation environment, based on autoVivado.

/zero.osf/one.osf/zero.osf Version 1.0 1 / 4
DX.Y "TITLE OF THE DELIVERABLE" Version 1.0
Document Information
Contract Number 780681
The LEGaTO project has received funding from the European Union’s Horizon 2020 research and innovation programme under the Grant Agreement No 780681.
4.5 Nanos@Cluster virtual memory layout
4.6 DFiant dataflow types: simplified inheritance diagram
4.7 Logical architecture of a data-flow computing system with one CPU and one DFE.
4.8 Compiling a data-flow application with MaxCompiler.
4.9 Task directive and clauses
4.10 Target directive and clauses
4.11 XiTAO and virtual topologies
4.12 C code of a simple computation inside a loop.
4.13 An MaxJ description that generates the data-flow implementation shown in figure 4.14.
4.14 A data-flow implementation for the computation inside the loop body.
4.15 Interaction between host code, manager and kernel in a dataflow application.
4.16 Intel SGX enclaves.
4.17 Our compiler-based protection mechanism transforms original code: (a) replicating original instructions with ILR for fault detection (b) and covering the code in transactions with Tx for fault recovery (c). Green lines highlight instructions inserted by our compiler using ILR and Tx.

5.1 WP3 and the LEGaTO Stack
5.2 The overall software toolchain for the LEGaTO platform.
5.3 LEGaTO middleware stack for resource management.
5.4 Composed node state changes during node composition process.
5.5 Dynamic reconfiguration of the communication infrastructure
5.6 The structure of a TAO
5.7 The structure of the DPA scheduler, as currently implemented in the XiTAO prototype, showing the work-stealing based load balancer (steps 1-2) and the pipelined TAO scheduler (steps 3-5).
5.8 The structure of the performance tracking table.
5.9 The overall methodology for FPGA undervolting experimental study
5.10 Power saving gain and reliability costs of FPGA BRAMs through undervolting

6.1 WP2 and the LEGaTO Stack
6.2 Overview of the RECS|Box server architecture [63]
6.3 RECS|Box Denbe (3 RU) and RECS|Box Durin (2 RU)
6.4 PCIe communication within the RECS|Box Server
6.5 PCIe communication combined with direct links between the FPGAs
6.6 Direct communication between FPGA and Host-2-Host PCIe communication
6.7 Overview of the proposed edge server architecture
6.8 Cost-optimized edge server architecture
6.9 Architecture of a Maxeler DFE
6.10 Architecture of an MPC-C series system. DFEs are directly integrated into a server.
6.11 Architecture of an MPC-X series system with 8 DFEs. Multiple MPC-X nodes and CPU nodes are connected through an Infiniband network.
List of Tables

3.1 Summary of Alya hotspots for explicit and implicit solvers. . . . . . 29
3.2 Top 5 AUC values, the correlation of this biomarkers with the dis-
ease and the probability . . . . . . . . . . . . . . . . . . . . . . . . . . 41
3.3 confidence interval of AUC for a specific number of biomarkers
after combining . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 43
3.4 Summary of the LEGaTO applications. . . . . . . . . . . . . . . . . . 51
3.5 Optimization objectives for each use case . . . . . . . . . . . . . . . 52
3.6 Energy baseline and optimization targets. . . . . . . . . . . . . . . . 53
4.1 DFiant Hierarchy Example: Inheritance, Polymorphism, Recursive
Composition, and Inlined View . . . . . . . . . . . . . . . . . . . . . . . 63
6.1 RECS|Box 4 model feature comparison . . . . . . . . . . . . . . . . . 111
6.2 Selected RECS|Box microservers . . . . . . . . . . . . . . . . . . . . . 112
6.3 TCO calculation of different server types . . . . . . . . . . . . . . . . 118
7.1 Summary of LEGaTO’s components contributions to the project’s
objectives . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 127
1. Executive Summary

This document, named super deliverable SD1: Architecture definition and evaluation plan for LEGaTO’s hardware, toolbox and applications, includes the overall project software and hardware architecture design as originally proposed in four deliverables:

- D5.1 Specification of application optimization and evaluation plans (WP5)
- D4.1 Definition/design of front-end toolbox (WP4)
- D3.1 Definition/Design of back-end Runtime System (WP3)
- D2.1 Hardware and firmware specification (WP2)

The contents of these deliverables have been consolidated in a single document, in order to reflect the tightly-integrated nature of all LEGaTO software and hardware stacks and to present the entire project in a more coherent and concise manner.

The document contains a summary of the technical achievements during the first nine months of LEGaTO project. During the period, application optimization plans were developed, the hardware specification has been drafted and first experiments conducted on the LEGaTO hardware, the definition and specification of the front-end toolbox and the back-end runtime system has been completed and first energy optimization results has been obtained.
2. Introduction

The final outcome of the LEGaTO Project will be a complete software toolset that includes a programming environment and an execution framework for energy-efficient execution of concurrent applications running on state-of-the-art heterogeneous hardware substrate. This document defines the target architecture for such a toolset, including the applications, the programming model and respective APIs, the runtime and the hardware architecture. It also defines the energy savings targets by which the success of the project will be measured during its evaluation phase.

Each chapter that follows focuses on the particular deliverable for the software stack that is targeted, namely the applications (D5.1 in the Application Development and System Integration Work Package, WP5), the programming model (D4.1 in the Tool Chain Front-End Work Package, WP4), the runtime (D3.1 in the Tool Chain Back-End Work Package, WP3), and the hardware (D2.1 in the Cloud to Edge Hardware Platform Work Package, WP2).

In this introductory chapter, a broad picture of complete project’s software toolset is presented. In this picture, the reader will find a detailed layered set of components spanning all Work Packages. Such presentation is made beforehand, aiming at giving a global understanding of the project before delving into each technical Work Package details. One will find in such picture a high-level explanation about how hardware and software components are entangled in order to reach the project’s goals.

Chapter 2 also summarises LEGaTO’s main achievements in the period preceding the document’s delivery date (months 1 to 9). These achievements are presented for each technical Work Package. In a nutshell, WP5 has produced specifications for all application use cases, including requirements, bottlenecks, metrics and optimisation plans. WP4 progressed in the definition of the front-end tool box, including programming model (annotations for fault-tolerance, energy requirements, etc.), execution framework and languages. WP3 defined the back-end runtime system with capabilities for FPGA support, execution management and reconfiguration, and hardware backend interfaces. WP2 produced a detailed hardware and firmware specification for LEGaTO, including definitions for the IoT gateway, network infrastructure, reconfigurable hardware support and the microserver testbed.

Chapter 3 refers to Work Package WP5 and describes five use case applications developed within LEGaTO. It presents a description of each application, their requirements, their main computational characteristics, code and data structures, along with metrics and optimisation goals. The first use case is a smart home application, processing privacy-sensitive information in order to provide assisted living recommendations. The second use case is a smart city application to model air quality in near-real time using fluid dynamics. The third use case is an energy-efficient machine learning application with surrounding perception and trajectory calculation for self-driving cars. Use case number four is a computation intensive monte carlo method for obtaining reliable classifiers,
to identify specific biomarkers that allow for more precisely diagnosing each disease. The fifth use case is focused on IoT security and usability, developing a gateway for (secure) network connections of local and remote network devices.

Chapter 4 starts presenting OmpSs, the main programming model of LEGaTO’s project, to be developed in Work Package WP4. As LEGaTO extends previous work, this chapter presents the distributed memory version of OmpSs, being developed in LEGaTO to support dynamic scheduling of tasks across multiple computing nodes. It follows with the presentation of energy-efficiency, checkpointing and fault-tolerance functionalities, relating the front-end interface with the back-end implementations and additions to an integrated development environment. It then presents the extensions that will be implemented for integrating DFiant and Maxeler’s MaxCompiler for developing applications including code in FPGAs, as well as for static kernel identification. In the same chapter, we describe how LEGaTO will efficiently tackle irregular data structures in a distributed environment, how it will offer protection against security threats and software faults. The chapter concludes with specifications for middleware management extensions for a task-based programming model, logical-level orchestration of hardware resources, and node composition and dynamic reconfiguration.

Work Package WP3 takes care of LEGaTO’s back-end runtime system, presented in Chapter 5. The chapter starts presenting the middleware system functionalities. It describes LEGaTO’s heterogeneous resource allocation and monitoring (CPU, GPU, FPGA, etc.) and its topology-aware distributed scheduling, with elastic placement and runtime power management. It proposes APIs for node composition, dynamic reconfiguration and hardware topology management, task replication, energy-efficient heterogeneous multi-level checkpointing. It also presents how to deal with performance monitoring and debugging as well as with fault-tolerance.

Chapter 6 describes the underlying hardware developed in LEGaTO’s Work Package WP2, to be used used for the applications and use-cases. The chapter presents a firmware architecture and the interfaces with the software layers (runtime system). It describes how LEGaTO will capitalise on existing architectures as Christmann’s RECS|Box and Maxeler’s DFE and specifies a modular edge platform for supporting the application use cases.

2.1. The LEGaTO Software Stack at a Glance

To set the stage, we include a brief reminder of the LEGaTO stack in Figure 2.1. In addition to the description of the particular stack level (hardware, middleware, runtime, applications), Work Package identifiers were also included in the figure.

The LEGaTO project will apply the energy-efficient software toolset for heterogeneous hardware to five applications. The first application will be healthcare. The project will not only demonstrate a decrease in energy consumption in the healthcare sector; it will also show that the toolset will increase healthcare application resilience and security; both of which are critical requirements in this area. In three further applications, the project will demonstrate ease of programming and energy savings possible through the use of the LEGaTO project
software–hardware framework for secure IoT gateways, smart homes, and smart city applications. The fifth application will be based on machine learning (ML), where the project will demonstrate how to improve energy efficiency by employing accelerators and tuning the accuracy of computations at runtime. In addition, the machine learning use case will be used to further optimize the energy efficiency in the two other use cases, as well as within the runtime.

On the software side, LEGaTO will have a task-based toolset to start with. This task-based toolset, has three pillars: at the higher-level the tool-chain is driven by the task-based OmpSs Programming Model, which is an extension of OpenMP intended to influence future generations of the standard. This is coupled to the Mercurium compiler which can handle CPUs, and GPUs, and is being extended to handle FPGAs. The third element of the pillar is the Nanos runtime, which schedules tasks to compute elements in data-flow fashion. One ambition of the LEGaTO project is to enhance this tool-chain with energy-efficiency transformations and elevate the enhanced tool-chain to a high maturity level through an IDE release by the end of the project. A major research direction is to make tasks moldable, which enables a more efficient resource management and sharing. Such generalized tasks are currently part of the task-based runtime titled XiTAO. By the end of the project, the promising research ideas from XiTAO will be integrated to the common runtime.

Among the LEGaTO hardware substrates, FPGAs provide both a promise and challenge. On one side, FPGAs are very power efficient, and can substantially accelerate such applications as Deep Neural Network inference (which will be deployed as part of the ML use-case by Data Intelligence). On the other side, FPGAs have been notoriously difficult to program. This will be addressed in LEGaTO through the DFiant high level synthesis (HLS) language available to develop key kernels for the FPGA fabric; the project will integrate these kernels into the common

![LEGaTO Stack Diagram](image-url)
LEGaTO runtime by the end of the project.

This hardware platform features GPU and FPGA accelerators that are already energy efficient. Even so, by the end of the project, we are planning to further expand the energy efficiency of this heterogeneous platform by endowing it with a hardware interconnection fabric. Moreover, the microclusters in this heterogeneous platform will be dynamically composed as requested by the LEGaTO runtime. OpenStack will be extended to support these dynamic compositions of microclusters, incorporate the specific needs of the different heterogeneous microservers as well as the communication infrastructure between them. Using OpenStack as a middleware layer allows accessing the hardware and deploying the applications in a standard uniform way, supporting cloud computing use cases.

The second heterogeneous platform is based on Dataflow Engines (DFEs). DFEs are also based on FPGAs, but their architecture is highly optimised for throughput-oriented computations of large data sets, making them ideally suited for HPC and big-data workloads. OmpSs will provide support to offload large, static subgraphs to DFE accelerators. These will be then compiled into DFE kernels with the MaxCompiler toolchain.

### 2.2. Achievements During the Work Period

This section briefly introduces the contributions during the work period in each of the Work Packages.

- **D5.1 Specification of application optimization and evaluation plans (WP5)**
  - For each application, requirements have been defined
  - For each application bottlenecks have been identified
  - For each application metrics and optimization goals has been determined

- **D4.1 Definition/design of front-end toolbox (WP4)**
  - Definition of Programming Model support - through annotations - for fault-tolerance and security is complete
  - Extending the SCONE toolchain - a shielded execution framework that enables unmodified legacy applications to run inside Intel SGX enclaves, with more functionalities. SCONE now can provide secrets and configuration management service and supports the R programming language
  - Focused on developing the DFiant language and toolchain, as well as began implementing a RISC-V processor using the tools

- **D3.1 Definition/Design of back-end Runtime System (WP3)**
  - The XiTAO runtime has been extended to support single-ISA heterogeneous devices and to automatically select the amount of resources for each computation.
- Targeting fault tolerance and energy efficiency, a study of voltage scaling in FPGA BRAMs has been performed.
- Basic implementation of GPU checkpointing is complete
- Implementation of Redfish API into the RECS Master was achieved
- improved the OmpSs@FPGA infrastructure to support the ZCU102 Xilinx development board.
- Ensured that OmpSs runs properly on the Intel i7, Intel i5 and in the Huawei Taishan 2180 and 2280 nodes of the hardware platform.
- Definition of firmware and software development to support dynamic (re-)configuration of the communication infrastructure and processing elements
- Definition of middleware stack and mechanisms to enable dynamic (re-)configuration of the communication infrastructure
- Discussion and definition of various interfaces between software components like Firmware, OpenStack, LEGaTO middleware and runtime and applications
- Defining the tool-chain backend interface for communicating with the DFiant environment
- Evaluated tradeoffs in security, performance and energy consumption with different CPU architectures (Intel, AMD, ARM), leading to the design of a heterogeneity- and energy-aware task-based scheduler

- **D2.1 Hardware and firmware specification (WP2)**
  - Detailed definition of the Secure IoT Gateway development, including software specification and preliminary hardware selection
  - Analysis and discussion about possible optimization paths for the Secure IoT Gateway, result is that there is no obvious optimization path using the LEGaTO toolflow
  - Definition of metrics and optimization goals like high throughput and low latency for VPN connections
  - First performance measurements for VPN encryption, using different CPU types and software configurations
  - First specification of potential hardware developments, based on application feedback and a detailed TCO analysis
  - Definition of firmware and software development to support dynamic (re-)configuration of the communication infrastructure and processing elements
  - Definition of middleware stack and mechanisms to enable dynamic (re-)configuration of the communication infrastructure
  - Discussion and definition of various interfaces between software components like Firmware, OpenStack, LEGaTO middleware and runtime and applications
  - RECS Testbed deployment and maintenance, support for testbed access and porting of tools to RECS hardware
- Continued hardware development of RECS microservers and RECS infrastructure
- Development of edge server architecture based on application requirements
- Established a cross-WP understanding of the flexible high-speed, low-latency communication infrastructure for integration into the LEGaTO toolflow
- First specification of potential hardware developments, based on application feedback and a detailed TCO analysis
- Evaluated aggressive undervolting on FPGAs for energy efficiency

3.1. Executive Summary

This chapter describes the optimization and evaluation plans for the use case applications in WP five of LEGaTO. It covers the contents of what was originally described as deliverable D five one “Specification of application optimization and evaluation plans” in the project work plan.

3.2. Introduction

The LEGaTO project targets five use-case applications that are used to evaluate different aspects of the tools that are developed in the project. The overall objectives of WP five are:

- Integrating of all developed components from the project into a working setup to support the use-case applications
- Optimization of the use-case applications with the support of the LEGaTO tools and development demonstrators for these use-cases
- Evaluation of the project’s objectives

Four of the use cases are application oriented and they will be used for optimization and evaluation as described above whereas the fifth use case, an IoT

---

**Figure 3.1. WP5 and the LEGaTO Stack**
gateway, is used as an integration example for the hardware and tool flow of the project.

Figure 3.1 illustrates the role of WP5 in LEGaTO. The use case applications form the input to the overall toolflow development and they will result in optimised implementations running on the hardware. In the following, we describe the optimization that we want to achieve for these applications as well as their evaluation plans.

We first summarise the relevant optimization objectives. The LEGaTO project considers the following four general objectives as stated in the DoA for improving applications:

1. Improve Energy Efficiency
2. Increase MTBF (by 5x)
3. Increase code base security (by 10x)
4. Increase designer productivity (by 5x)

In addition, we also consider the following objectives. These are not explicitly stated as objective in the DoA but could also be relevant:

5. Reduce TCO
6. Reduce Latency
7. Reduce Cost

The LEGaTO project uses several programming models that will be instrumental to achieving the objectives outlines above. The targeted programming models are:

1. OmpSs & XiTao
2. MaxJ
3. Dfiant

These programming models and their associated tools are described in more detail in Chapter 4.

In the following subsections we provide a detailed description of each application. This includes a description of the application requirements, computational behaviour and structure of the algorithm, and the optimization goals for each application.

In the summary of this chapter (section 3.8), we provide a short overview of all applications on their key characteristics, baselines as well as specific optimization goals.
3.3. Smart Home

Current smart living environments are based on the simple automation of sub-systems consisting of sensors, information processing, and actuators. New approaches mainly driven by large enterprises push big-data approaches, collecting as much information about the user as possible to derive the current action and to anticipate future behavior. The development of assisted living can be seen as a move from isolated applications realized as simple embedded systems, towards cyber-physical systems gathering and processing large amounts of data from a high number of distributed smart devices. Additionally, the smart home has to process interaction of different users simultaneously; conflicting actions have to be recognized (e.g., one user opening a window and another one closing it again) and compromises can be suggested. Different interaction schemes can be combined adaptively, e.g., switching from touch to speech interaction while cooking or using text-based output while phoning. Providing this functionality is highly computational intensive. Since the collected data contains personal and highly sensitive information, cloud-based processing is undesirable. To address these privacy issues, we target resource-efficient edge computing.

3.3.1. Application Description

At Bielefeld University a research apartment for the evaluation of smart home prototypes was developed. This smart home is used in LEGaTO for implementation of a uniform configurable platform as well as the generation of data sets. Key domains of the apartment are the kitchen, the entrance hall and a personal fitness coach. In these domains prototypes are evaluated, like a smart kitchen with object detection, sensors, and actuators for opening cupboards and drawers. Each prototype is developed autonomously and can be operated independently. Therefore, heterogeneous prototypical information processing is embedded within each area. Currently, conventional PC hardware components are used, not focusing on space constraints and energy efficiency. Porting the functionalities onto embedded devices was postponed before. All sensors and actuators are interconnected via a common middleware to utilize the greater potential of a smart home environment.

For everyday use, smart home environments require an intuitive and comfortable interface for user interaction. Depending on the desired functionality of an individual assistance system, specific modalities for interaction are preferred by the user, e.g., touch, speech, gestures or even emotions. Within LEGaTO, a uniform configurable platform for distributed interaction and information processing in smart homes is developed. The platform can be configured to integrate arbitrary combinations of basic functionalities like face, object and speech recognition. As an example implementation, this platform is used to realize a central human-machine interface for smart homes integrated in a wardrobe mirror. Figure 3.2 shows the current prototype of such a smart mirror. In addition to information and control of the state of the smart home environment (heating, ventilation, and air conditioning (HVAC), security, illumination and many more), the smart mirror provides cognitive abilities targeting more sophisticated assistive functionalities including, e.g., virtual try-on of a dress, guidance to tie a necktie or interpretation of user intention.
For an increased support in the daily routine in smart home environments a situational memory is essential. In a first implementation the database will be based on the user interaction of the uniform configurable platform developed within LEGaTO. Later it can be enhanced by the additional sensor information of the smart home environment. Within this database the everyday behaviour of the residents and environmental influences are represented. Data sets and evaluations of this situation memory will then be publicly shared. Based on this historical knowledge machine learning algorithms are usable to predict the behaviour or detect anomalies of the resident’s behaviour. This can vary from controlling light, regulating actuators or presenting information on its own, which are regularly accessed, for behaviour prediction and health related problems, user misbehaviours or other not planned events.

### 3.3.2. Requirements

For the user interface, the smart mirror prototype builds upon the open source project “magic mirror” [87]. In addition to the basic visualization provided by this environment, our prototype integrates compute-intensive methods for face, object and speech recognition.

Face recognition is used to provide personalized content for individual users. It is realized using deep convolutional networks, implemented in three steps with TensorFlow. In the first step, a pretrained graph from “WIDERFace” [91] is used to find faces in the input video stream. Subsequently, a feature representation is computed for every detected face using “FaceNet” [77]. Based on the extracted
features, a classifier is trained to determine the identity of the respective person using “Scikit-learn” [68].

Speech recognition provides an easy and comfortable possibility for direct interaction with the smart home environment. The smart mirror provides a local system for speech recognition using Mozillas “DeepSpeech” [2]. Therefore, audio streams are partitioned into 20ms slices and fed into a recurrent neural network, which outputs the corresponding character.

Object detection is another key feature for smart environments. For the proposed implementation, “YOLO” is used, which is capable of recognizing more than 9,000 object categories [74] in it’s biggest form. The version used in this project is “YOLOv3”. It is implemented using the toolkit called “Darknet” which is written in C [73].

For the image based functionalities typical image processing algorithms are also widely used (e.g., image scaling or color shifting). Therefore and for its fast image capturing capabilities, OpenCV is a necessary dependency. Googles TensorFlow [1] is also heavily used.

3.3.3. Computational Characteristics, Code and Data Structures

The face recognition component of the smart mirror operates on a full HD (1920 x 1080p) camera stream with 30 FPS. It consists of three steps, visualized in Figure 3.4. First, every frame is passed through a neural network from WIDERFace which is trained to identify faces in images. It returns a bounding box around each face and a probability to be a face for each box. Each detection is copied into a separated image and scaled to 160 x 160 pixels, if the probability is high enough. In the second step, a numerical representation of each face is calculated by a neural network called FaceNet. The pre-trained FaceNet model returns a 128-dimensional unit hyper sphere per face. This representation is fed into a classifier created with sklearn in the third step. Due to previous training of this classifier, it calculates the identity of each face inside the image. For visualisation purpose the bounding boxes and identities are annotated on the images. The annotated image and the identities are returned and visualised on the smart mirror. In the current implementation, no step is calculated in parallel and each
single image is fully processed. On the reference architecture using a workstation (i7-770K@4.2 GHz, Nvidia GeForce GTX 1080 Ti, 32 GB DDR3) a throughput of 25 FPS on full HD images is achieved. The GPU is utilized by 40% and 3 GB of video memory is used.

![Figure 3.4. Face recognition pipeline structure.](image)

DeepSpeech2 by paddlepaddle is used for speech recognition [2][67]. It provides a local server which receives audio streams and returns a transcript of what is understood. This way it can be used by other smart devices as well. It’s pipeline structure is shown in Figure 3.5. For the neural network of DeepSpeech2, the audio stream is cut into 20 ms slices. Each slice is then fed into the network which returns the probability for each character (from a-z, white space and '). A Matrix consisting of all probabilities for all slices is then passed to a language decoder. Here, a language model is used to extract the transcript, which is finally returned. The implementation runs on the Nvidia GeForce GTX 1080 Ti with a utilisation of 13% and 1 GB of video memory usage.

![Figure 3.5. Speech recognition pipeline structure.](image)

The full HD camera stream of the face recognition is also feed into the neural network of YOLO. It works as a single shot detector which returns bounding boxes and labels for trained objects. In the current version it utilizes about 30% of the Nvidia GeForce GTX 1080 Ti and takes 1 GB of video memory.

In the whole setup, the video stream is shared between the face recognition and the object detection. The utilisation of the GPU (Nvidia GeForce GTX 1080 Ti) for the total smart mirror is between 70% to 75% and 5 GB of video memory with a power consumption of 650 Watt. The frame rate of face recognition and object detection drops to 10 FPS if both are running. Main parts of the examined functions are neural networks using TensorFlow. Optimizing TensorFlow inference for embedded devices can therefore be beneficial, because of the power consumption of 260 Watt of the GPU.

### 3.3.4. Metrics and Optimization Goals

Due to the nature of smart home devices, i.e., processing sensitive personal data of the users, increasing privacy is one of the main objectives. This is a main driver...
for providing a more and more decentralized solutions with local processing instead of cloud-based processing. Therefore, using the hardware architecture and toolchain of the LEGaTO framework implicitly addresses the privacy optimization goal. Using a decentralized processing in the smart home increases the necessity to consider costs as well as energy consumption. Big expensive and high-power server solutions are not desirable for the use within a smart home. Embedded systems and optimized edge computation are therefore the targeted solution within LEGaTO. In addition this smart home use case can be utilized as a test setup to evaluate the benefits of the LEGaTO toolchain in terms of designer productivity and code base security.

Optimizing three base components of smart homes (object detection, speech recognition, and face recognition) with the LEGaTO tool-flow to run on a small embedded solution consisting of small microservers with CPU, GPU and FPGAs is the goal of this use case. Such a hardware platform can be used near the sensor and actuator individually, forming a closed system or shared within the home environment. In the second case, data streams like a video or audio stream are sent from a sensor to a small home server to be processed. Securing the data paths within this local cloud provides an important use case for the IoT Gateway presented in Section 3.7.

To optimize the smart home use case towards the optimization goals of Table 3.5, we will map the application on the hardware using the LEGaTO tool-flow. Baseline will be the reference architecture described above. In order to utilize the full potential of the hardware, we will focus on the key computational components based on frameworks like TensorFlow and Darknet. Porting base functionalities to embedded hardware using OmpSs will be the starting point for the optimization of the use cases. Next step will be a increased utilization of GPU or the usage of FPGAs.

As key metrics for the optimization performance, the energy efficiency and the overall costs (price) will be evaluated. Performance of the use case will be evaluated in terms of FPS and detection accuracy. Energy consumption will be evaluated by measuring power consumption of the power supply over time. The first prototype provides a starting reference of around 10 FPS for object detection and face recognition with utilization of 70% of the Nvidia GeForce GTX 1080 Ti and a power consumption of 650 Watt. Reducing the overall power consumption to around 50 Watt is a major goal to build a suitable in home device. The costs will be dependent on the amount of used hardware, e.g., embedded CPU, GPU, FPGA, hardware platform, and power supply, and its corresponding price.

### 3.4. Smart City

In many urban areas air quality and associated impacts on public health are matters of growing concern. The emission and dispersion of critical pollutants (PM$_{2.5}$, NO$_2$ and ground-level O$_3$) correlate with cancer, asthma, cardiorespiratory problems, brain development in children and reduction of life expectancy in general. Public administration and health agencies are tasked to monitor the quality of air and, eventually, to make model forecasts to assist on the adoption of reacting measures and to warn on air pollution episodes affecting vulnerable groups of citizens.
According to the air quality report elaborated by the EEA [31], large parts of Europe are affected by air pollution values that exceed European standards and the World Health Organization (WHO) Air Quality Guidelines (AQGs). This report estimates that more than 82% of the urban population in the EU-28 were potentially exposed to annual average concentration levels higher than the WHO reference limits for PM$_{2.5}$ and O$_3$ between 2013 and 2015. Unfortunately, this situation is common and growing in many large metropolitan areas of EU and elsewhere.

As a consequence, air quality monitoring networks and modelling forecasting systems are critical to increase awareness and, ultimately, to assist decision-makers on the adoption of measures to protect public health. However, and despite the fact that air quality has a significant impact on human health, most cities do not have a prediction and alert system able to nowcasting and forecasting pollutants at urban-scale (street-level) yet. Gaussian dispersion models, which are commonly used, are insufficient to characterize near-surface wind fields due to the lack of urban geometries (buildings) in the model. Today, the increase in computational capabilities is making possible a near-future scenario in which Computational Fluid Dynamics (CFD) models are used to simulate urban-scale winds and pollutant dispersion operationally. These CFD-based models are known to be more accurate and reliable than current modeling approaches which are still unable to resolve the urban morphologies.

In this upcoming scenario, the LEGaTO stack will be pivotal, both in terms of leveraging the processing capabilities and improving the energy-efficiency of an operational urban-scale air quality modelling system for the Smart City use case. The Smart City use case aims at demonstrating that monitoring of urban air quality through CFD simulations is feasible for nowcasting predictions (short-range forecast – 2h/6h) in an operational workflow (daily run). These objectives respond to the current social and environmental concerns by providing a state-of-the-art computational system that would allow, for the first time, high-resolution (<10 meters) simulations down to the street level.

### 3.4.1. Application Description

The Smart City use case is composed of four different components that, individually, meet a specific project objective and, in combination, will allow the development of an operational urban air quality modelling system at street-level resolution based on CFD. Figure 3.6 shows the key components of the air quality modelling system. Most of the individual components have already been developed within other research projects. In this operational workflow, different inputs are required by the CFD model to simulate urban-scale winds.

**Meteo data**

CFD models require initial boundary conditions to confine the physical problem into a finite computational domain (e.g., a city mesh). These boundary conditions set the inputs of our CFD simulation, defining how the fluid, or wind in our case, enters (inlet) or leaves (outlet) the domain. In other words, boundary conditions connect the region of interest (our urban area) with its surroundings.

In a urban-scale context, street-level winds are nested with meso-scale Numerical Weather Prediction (NWP) models that furnish the time-dependent boundary
conditions. These boundary conditions are set with daily-basis meteorological forecasts data generated by weather forecast agencies. This data defines initial conditions for wind fields, velocity and direction, at meso-scale level (∼1-3 km resolution). Then, wind fields are downscaled to street-level resolution (<10 meters resolution) and are used as initial values for the simulation. In this pre-processing, a logarithmic wind profile is used for the lowest portion of the planetary boundary layer (PBL) to simulate the roughness effect near the surface and the buildings.

**Sensor assimilation**

In order to model and forecast urban-scale pollutant dispersion, it is not only necessary to dispose of high-resolution near-surface wind fields, but also to characterize the sources of pollutants at street-level (mainly derived from vehicle combustion) through sensors or emission inventories.

Using the city of Barcelona as testbed, real-time data from an IoT network of environmental sensors will be used to initialize and validate the CFD-based high-resolution urban-scale air quality model. This network is being developed at the BSC in a joint collaboration with the Barcelona city council (IMI – Municipal Institute of Information) within the framework of the H2020 GrowSmarter project [36] where, as a proof of concept, an initial sensor deployment has already been made at the Barcelona’s 22@ district. Each monitoring node, composed of an Arduino board, has been developed in the framework of the H2020 CAPTOR project [15] and integrates several sensors on a single device: an anemometer for wind direction and velocity, and several NO$_2$, O$_3$ MOX sensors.

---

*Figure 3.6. Conceptual sketch of the urban-scale air quality forecast system.*
CFD-based wind and pollutant dispersal modelling

Within the Smart City use case, the CFD-based simulator is the main core of the whole application. To this end, the BSC has developed an urban-scale wind forecasting system based on coupling the meso-scale WRF model with the in-house Alya modelling system. Alya [8] is a multi-physics code developed at BSC and designed to solve engineering coupled problems on large supercomputing systems such as MareNostrum. In order to simulate the air flow through the urban-scale morphologies (buildings), the CFD model solves the incompressible Navier-Stokes equations on a computational mesh that represents the city geometry. However, the use of CFD techniques requires massive computing resources for several reasons.

First of all, meshes with high-resolution (tens of meters) near-surface must be used in CFD simulations to capture the main geometrical features of a city (buildings), which in turn exert a major control on the ultimate wind flow. These meshes can easily reach hundreds of millions of elements making unfeasible to run simulations on single nodes. Second, over every fraction of time for the forecast, the CFD model must compute the stationary wind field (steady-flow) at the pedestrian level (~2-10 meters over mesh ground) to transport pollution particles. In an idealistic case, reaching a stationary flow should be desirable. However, the urban morphology is by default very turbulent due to its geometrical irregularities, and significant wakes are produced in the leeward regions of the buildings. This turbulent flow forces to simulate a longer period of time and average the wind fields across all the iterations to obtain a mean flow. Finally, in an operational (hourly basis) forecast context, the entire workflow must be re-run constantly in a short period time to update forecasts in time, being this the main constraint for a successful deployment.

Considering all the previous statements, the steps to follow by the CFD-based simulator for each forecast iteration can be summarized as:

1. Set the initial boundary conditions for the current forecast iteration given the downscaled wind field of the NWP model.
2. Solve the LES incompressible Navier-Stokes equations in the city mesh until a stationary flow is reached. In case of a high-turbulent case, a mean flow is computed with the averaged wind fields of the previous steps.
3. Combine the pollutant data from the real-time sensors with the urban wind model. Wind speed and direction are used to weight sensor data down-stream using a Kriging process interpolation.
4. Store the results of the pollutant transport model to construct nowcasting air quality maps at 10 meters resolution based.

Data gathering and streaming

After the simulation, data is collected directly from the CFD model, and the environmental network. Pollutant data is post-processed and published using an Open Data format in a BSC repository. The post-processing may require large
amount of memory resources to be interpolated and stored in databases de-
pending upon the mesh size and refinement. Finally, the data provided, which
includes two groups of sources, might be used by visualization and analysis tools
for their study:

1. Real-time measurements from the monitoring network on a half-hourly
basis, and

2. High-resolution (10 meters) nowcasting maps at any point in the target
area on a half-hourly basis for the pedestrian level (~2-10 meters over
ground); offering the following attributes for each group (real-time and
simulated):

   (a) Wind fields: wind speed (ms⁻¹) and wind direction (degrees)
   (b) Concentration values for NO₂ and O₃ (µg/m³)

3.4.2. Requirements

The Smart City application relies heavily on the Alya code, which is the main
computational code in the whole use case. Alya is one of the twelve simulation
codes of the Unified European Applications Benchmark Suite (UEABS) and thus
complies with the highest standards in High Performance Computing (HPC). The
code has been tested on several supercomputers worldwide, where it has proven
to efficiently scale up to 100K cores for industrial applications.

Alya is almost a self-contained software, and does not depend on any external
library or application. However, in order to run in parallel, a mesh partitioning
library such Metis is required. However, to avoid versioning problems, this li-
brary is already included in the Alya repository. Alya is written in Fortran and C,
and can use OpenMP, OmpSs and MPI standards to allow intra- and extra-node
parallelization. It has also been optimized for accelerator-based platforms such
as many-core architectures with SIMD instructions and GPGPUs. The Alya code
has been tested in many different operating systems, but GNU/Linux OS with
Intel/GCC compilers are the most recommended.

Numerically, wind flows can be computed using low precision floating point op-
erations (4-byte) due to the velocity magnitudes (<100 ms⁻¹) and the precision
required (2 decimals) in the street-level flows. On the other hand, reduced pre-
cision floating point operations such as 2-byte, or even integer precision, might
be also enough, but they require a previous study first.

3.4.3. Computational Characteristics, Code and Data Structures

The Alya CFD-based simulator used in the Smart City application is the main
compute-intensive part of code. As commented previously, it solves the follow-
ing incompressible Navier-Stokes equations (momentum and mass conserva-
tion) to obtain the street-level wind fields,

\[
\frac{\partial \mathbf{u}}{\partial t} + \rho (\mathbf{u} \cdot \nabla) \mathbf{u} - \rho \nu \Delta \mathbf{u} + \nabla p = \mathbf{f},
\]
\[
\nabla \cdot \mathbf{u} = 0,
\]
where \( \rho \) is the fluid density, \( \nu \) the kinematic viscosity \( (\nu = \mu/\rho) \) and \( p \) the pressure. Dimensional variables \( \bar{\vec{u}} \) and \( \bar{\vec{f}} \) represent the mean turbulent velocity and the external forces (e.g., gravity), respectively. Alya solves the physical equations on unstructured meshes using a variational multiscale finite element method (FEM) for the spatial discretization. The time discretization is based on the trapezoidal rule, and the linearization is carried out using the Picard method.

After space and time discretizations and linearization, the resulting algorithm consists in solving at each linearization and timestep the following algebraic system:

\[
\begin{bmatrix}
A_{uu} & A_{up} \\
A_{pu} & A_{pp}
\end{bmatrix}
\begin{bmatrix}
\bar{\vec{u}} \\
\bar{\vec{p}}
\end{bmatrix}
= 
\begin{bmatrix}
b_u \\
b_p
\end{bmatrix}
\tag{3.2}
\]

where \( \bar{\vec{u}} \) and \( \bar{\vec{p}} \) are the velocity and pressure nodal unknowns to find at each timestep. The four sub-matrices and two RHS in Equation 3.2 come from the assembly of the weak form terms of the Navier-Stokes equations. \( A_{uu} \) includes the Galerkin momentum and stabilization terms. \( A_{up} \) includes the Galerkin pressure gradient and the stabilization terms. \( A_{pu} \) includes the velocity divergence operator as well as the part of the pressure stabilization involving the velocity in the momentum residual. Finally, \( A_{pp} \) includes only the pressure stabilization.

In order to obtain the wind flow in the urban geometries, a semi-implicit fractional step solver is used, solving the momentum explicitly (convective terms) and the pressure implicitly (viscous terms). The fractional step technique (see Algorithm 1) segregates the two ruling equations in 3.1. This means firstly to freeze or extract the pressure gradient term from the momentum equation and secondly to solve the continuity equation computing the new pressure. A last correcting step is taken to upgrade the momentum, in a step called projection or correction. The main advantage of decoupling the velocity and the pressure is twofold. First, the implicit treatment of viscous terms allows a stable solution even at a larger time-step size, which is limited in fully explicit schemes due to the small grid size near the wall. And second, it reduces the cost of the simulation due to the better numerical stability and the savings in computation time because the reduction of converge iterations.

As shown in Algorithm 1, first all the input parameters are read, including the mesh (city geometry), and the initial values and boundary conditions provided by the downscaled NWP model. Then, to allow parallel runs, the global mesh is partitioned depending upon the number of computational nodes to be used in the simulation. Alya can exploit three different levels of parallelism to be found in current supercomputers (see Figure 3.7). Over each level of parallelization, a new subdomain is created and associated to a logical or physical computational unit.

After assigning the different subdomains to each computational node, the local \( Q \) (Schur complement preconditioner that approximates \( A_{pp} \)), \( A_{up} \) and \( A_{pu} \) matrices for each domain are assembled. They remain constant, and therefore can be assembled just once at the beginning of the execution. On the other hand, the \( A_{uu} \) matrix and the RHS’s \( b_u \) and \( b_p \) from Equation 3.2 must be assem-
Algorithm 1 Semi-implicit incompressible Navier-Stokes solver with fractional step method implemented in Alya.

1: Read input (mesh, boundary conditions, initial values)
2: Partition of global mesh (extra-node MPI parallelism)
3: Assemble $Q, A_{up}, A_{pu}$ (remain constant)
4: for $t = 0, \text{timestep}_{stationary}$ do
5: Compute timestep $\delta t$
6: while timestep not converged do
7: Solve Momentum equation explicitly $\bar{u}^*$
8: Solve Pressure equation implicitly $p^{t+1} (Q \Delta p = F_c)$
9: Correct Velocity $\bar{u}^{t+1} = \bar{u}^* - \delta t / \rho \nabla (p^{t+1} - p^t)$
10: Compute residuals $||\bar{u}^{t+1}||$
11: end while
12: Write $\bar{u}^{t+1}$ and $p^{t+1}$ unknowns
13: end for

Figure 3.7. Levels of parallelism in Alya code. From left to right: 0) Initial mesh in sequential mode, 1) extra-node parallelism (by means of MPI), 2) intra-node parallelism between chips and cores (by means of either MPI, OpenMP or taskification), and finally, 3) data-level parallelism (by means of vectorization or accelerator ISAs).

bled over each timestep and convergence iteration of the semi-implicit algorithm. Finally, within the timestep and the convergence loop, we find the three main computational-intensive parts of the fractional step algorithm (lines 7, 8 and 9 in Algorithm 1), where the momentum and the pressure solvers take approximately $\approx 60\%$ and $\approx 30\%$ of the whole execution time, respectively. For the current analysis, we will concentrate on the explicit solver for the momentum equation, which uses the Algorithm 2, and has shown to be a good candidate for using the LEGaTO stack due to its computation intensity and the low communication patterns required for the solver (see Table 3.1).

The Algorithm in 2 shows the steps required by the explicit solver of the $\bar{u}^*$ momentum term (line 7 in Algorithm 1) to be computed. This solver requires the assembly of the $A^e$ matrix and $b^e$ RHS for each element in the MPI subdomain, gathering elemental properties (e.g., $\rho, \nu, \vec{F}, \vec{u}$) from global arrays to local arrays, computing each elemental term, and finally scattering back the results to
Algorithm 2: Explicit solver for $\vec{u}^*$ using OmpSs commutative multidependences for the $A_{uu}$ matrix and $b_u$ RHS assembly.

1: Partition local mesh in $n_{subd}$ subdomains of size $Chunk_{size}$
2: Store connectivity graph of subdomains in structure $subd$
3: for $isubd = 1, n_{subd}$ do
4:     $task = subd(isubd)$
5:     $nneig = SIZE(task%lneig)$
6:     !$OMP TASK
7:     !$OMP COMMUTATIVE([subd(task%lneigh(i)), i=1,nneig])
8:     !$OMP PRIORITY(nneig) PRIVATE(task,...) SHARED(...)
9:     for Elements $e$ in $isubd$ do
10:        Gather: global to element arrays
11:        Computation: element matrix and RHS: $A^e, b^e$
12:        Scatter: assemble matrix and RHS $A^e, b^e$ into global $A_{uu}, b_u$
13:    end for
14:    !$OMP END TASK
15: end for
16: !$OMP TASKWAIT
17: Solve $\vec{u}^*$ explicitly ($\vec{u}^* = \vec{u}^t + \delta t M^{-1} (b_u - A_{uu} x^t)$)

<table>
<thead>
<tr>
<th>Operation</th>
<th>Communications</th>
<th>Solvers</th>
</tr>
</thead>
<tbody>
<tr>
<td>Time minimization</td>
<td>AllReduce</td>
<td>Both</td>
</tr>
<tr>
<td>Elemental assembly</td>
<td>None</td>
<td>Both</td>
</tr>
<tr>
<td>Sparse Matrix Vector</td>
<td>Async SendRecv</td>
<td>Both</td>
</tr>
<tr>
<td>Dot product</td>
<td>AllReduce</td>
<td>Implicit</td>
</tr>
<tr>
<td>Converge criteria</td>
<td>AllReduce</td>
<td>Both</td>
</tr>
</tbody>
</table>

Table 3.1. Summary of Alya hotspots for explicit and implicit solvers.

the global matrix and RHS of the MPI subdomain. Once the global matrix and the RHS are built, the $\vec{u}^*$ is solved explicitly by an Sparse Matrix-Vector (SpMV) operation (line 17) that requires MPI_SendRecv communications to synchronize nodal contributions from elements lying on the boundaries of different MPI subdomains. This computation is parallelized across multiple cores within a node using OpenMP. The SpMV computation is split between boundary nodes and internal nodes. First, boundary node’s contributions are sent and received asynchronously between MPI neighbours (MPI_ISend/MPI_IRecv). Then, the SpMV is computed for the internal nodes, which do not depend on any external contribution. Finally, an active wait is issued (MPI_Wait) and the SpMV is performed for the boundary nodes once all contributions have been received.

In order to parallelize effectively the element assembly (elemental loop in line 9) within a node, Alya can use OmpSs taskification with commutative multi-dependences clause. The COMMUTATIVE(multideps) clause avoids any possible race condition when the global matrix is being updated by concurrent threads computing nodal contributions from adjacent elements. This OmpSs clause allows us to express incompatibilities between subdomains. The mesh of each MPI process is partitioned into disjoint sets of elements (subd), and by prescribing the
neighboring information in the OpenMP pragma, the runtime takes care of not executing neighboring subdomains at the same time. In order to dispatch critical domains earlier, a higher priority is given to these subdomains with a higher number of neighbors (PRIORITY(nneig)). This method presents good spatial locality and IPC and circumvents the use of ATOMIC pragmas or graph coloring algorithms over element loop parallelization, which in turn present worst performance.

Within the elemental loop of each task subdomain, line 11 is the most compute-intensive code of the whole explicit solver, and therefore a good candidate for further optimizations by means of data-level parallelism. Depending on the underlying hardware, autovectorization of the generated code may be activated by using just compiler flags. However, the vectorization will be inefficient if the compiler is not able to autovecorizer the appropriate loops. In order to support the compiler in this task, some refactoring of the data structures can be performed. Figure 3.8 shows how data is arranged for subroutines within the elemental loop in line 9. Elements of the same geometrical type (triangles or quads in 2D and tethras or hexas in 3D meshes) are grouped and processed in packs of VECTOR_SIZE length to ease autovecorization in the compiler. This Struct of Array (SoA) layout facilitates to the compiler to insert common operations over a group of VECTOR_SIZE elements by means of arranging consecutive data elements in the unit-stride dimension. Lists are padded with null values (0) when the number of elements is not multiple of VECTOR_SIZE.

In order to show an example of vectorizable loop in Alya, let us consider a typical element matrix assembly as shown in Algorithm 3. Let us denote nnode and ngaus as the number of nodes and Gauss integration points required to carry out the FEM integration rule of the geometrical element. Finally, Ae, Jac, and N represent the element matrix, the weighted Jacobian, and the shape function, respectively. In this loop the contribution of each geometrical element is computed and stored in a local Ae submatrix, which in a future step will be scattered into the global algebraic system Auu for the final SpMV operation.
Algorithm 3 Vectorization strategy for elemental matrix assembly ($\mathcal{A}_e$) in Alya code. The picture depicts the parameters used for the integration rule over a 2D triangle (TRI03).

```plaintext
do ig = 1, ngaus
   do jn = 1, nnode
      do in = 1, nnode
         Ae(1:VECTOR, in, jn) = Ae(1:VECTOR, in, jn) + 
                                Jac(1:VECTOR, ig) * N(1:VECTOR, in, ig) * 
                                N(1:VECTOR, jn, ig)
      end do
   end do
end do
```

By just defining the VECTOR_SIZE constant in the code in compilation time, we can assist the compiler to perform autovectorization in the unit-stride dimension of all involved arrays, thus assembling VECTOR_SIZE elements at the same time in one inner loop iteration. This strategy can be vastly used in the element matrix and RHS construction code, which is full of this loop pattern to add each property variable into $\mathcal{A}$ matrix and $\mathbf{b}$ RHS of the algebraic system. Finally, note that this formalism can be relatively easily applied to port the assembly operation to GPU architectures by just varying the VECTOR_SIZE.

### 3.4.4. Metrics and Optimization Goals

The operational air quality forecasting system at Barcelona is a very compute-intensive application, and is currently using the resources of the MareNostrum supercomputer. The main optimization goals to seek for this use case are porting the main compute-intensive kernels of the CFD engine (Alya code) by means of using the LEGaTO stack making use of the accelerator-based architectures such as FPGAs. The performance metrics mentioned previously will be used to evaluate how successful the optimization process has been with the LEGaTO stack. The offloading of the compute-intensive kernels will be done using the task-based LEGaTO toolset in a completely transparent manner, with a single source for multiple accelerators and compute engines; this will increase programmer productivity for heterogeneous CPU-FPGA-GPU based systems. Finally, since the air quality forecasting system is very computationally intensive, and it is also long running in terms of wall-clock time, it is likely to suffer from frequent correctable and uncorrectable errors as well as silent data corruptions; our goal is to take just the right amount of application-level checkpoints through runtime guidance; thus improving the MTBF substantially while limiting the energy overhead of checkpointing to application targets. In addition, as a side optimization goal, we will evaluate to which extent the IoT Gateway technology described in Section 3.7 could be used to secure the IoT infrastructure of environmental sensors.

In order to evaluate the LEGaTO version of the Smart City use case against the baseline one, it is needed to verify the correctness of the simulated results first. To that end, the wind field on some specific coordinates of the city mesh, known as witness points, will be collected and compared against the baseline version. Then the witness points will be evaluated using statistical metrics such as rela-
tive error, root mean squared deviation (RMSD), bias or R-squared.

Once that the LEGaTO version is checked, two metrics will be used to evaluate the ported version: the elapsed time (seconds used per CFD simulation timestep), and energy-efficient metrics (watts consumed per CFD simulation timestep and FLOPs per Watt).

The baseline evaluation will be the unoptimized Alya code running the smart city application. We will then evaluate the application by applying the LEGaTO optimizations mentioned above including OmpSs taskification to express efficient parallel execution of the application, using single or mixed-precision floating point with minimal loss of quality, and vectorization for efficient execution on accelerators. The overall target is at least 2× energy savings by applying the LEGaTO optimizations.

The testset used for the baseline comparison is based on a mesh with a single square building block of 22 meters height and 150 meters long on each side (see Figure 3.9). The whole urban-area mesh is composed of almost 1 million tetrahedral elements, with a higher element refinement on the boundary layers of the building and on its leeward wall. As initial condition for the simulation, a west-component (180 degrees) logarithmic wind field profile (10 m/s\(^{-1}\) on the top boundary) is used.

![Figure 3.9. Mesh used for the baseline test. Left: vertical cut. Right: horizontal cut.](image)

In order to set a fair comparison for the LEGaTO stack, a single MareNostrum IV node (2 × Intel Xeon 8160 with 24 cores each) was used to run the CFD model in a pure-MPI parallel way over 100 time-steps (see Figure 3.10). Different metrics were taken from the job execution for future comparisons with the optimized versions. The results obtained were: walltime execution 28.75 seconds, where the momentum element matrix construction took 12.63 and the implicit solver (pressure unknown) took 5.62 seconds respectively. The rest of time was spent on mesh reading and partitioning and writing the output results. The whole simulation consumed a total of 974 Joules on one MareNostrum IV node as reported by SLURM queue system.

The SmartCity use case is aligned with environmental policy objectives and strategies coordinated by the EU state members (see Clean Air Policy Package [30]). As a consequence, the results of the SmartCity use case might be of interest for EU cities in order to implement air-quality monitoring system and pollution reduction measures on urban areas with high-population densities. In addition, the LEGaTO goals on energy consumption and computing performance optimiza-
tions would allow a broader number of cities to afford this technology in order to make use of complex and accurate models for pollutants transport at urban-scale. One example for this technology transfer is the CLASS project [21], led by BSC, where one of the partners, the city council of Modena, might be interested in the deployment of the Smart City use case on their city.

3.5. Machine Learning

Autonomous driving is believed by many to drastically reduce the number of accidents and deaths in traffic. Today, human errors are the cause of 94% of traffic accidents [84]. Moreover, autonomous driving is a key component in reducing the environmental impact of our current transportation system by enabling more efficient driving, planning and business models around vehicle sharing.

However, developing these self-driving software and hardware systems are non-trivial from multiple aspects. To mention a few:

1. Optimal combination, specification and configuration of sensors
2. Develop robust and efficient algorithms that
   (a) perceive its surrounding and
   (b) use this information to travel on an optimal route to its destination.
3. Efficient and affordable hardware for executing the compute intensive self-driving algorithms
4. Making software and hardware robust against errors and attacks

One of the biggest issues, is the energy consumption in today’s systems. This consumption needs to be at least an order of magnitude smaller than in the typical development environment [85]. This is one of the major targets of the LEGaTO Project, please see Table 3.5, as well as increased robustness and safety, which makes the LEGaTO Project very relevant to the automotive industry.

3.5.1. Application Description

One of the key challenges for an autonomous car is to perceive its surroundings and understand what is a car, pedestrian, which lane the car is in and thus where it can drive safely. This part of the system is often referred to as the *perception*
A common research task for perception is pixel-wise semantic segmentation, where every pixel in an image is classified to belong to one of multiple classes of interest, e.g., road, pedestrian, car, lane marking, etc., see Figure 3.11. There is a large research interest in improving this task and the current state of the art algorithms are based on deep learning (DL) [34, 76, 44], which is a subfield of machine learning (ML), which is a subfield of artificial intelligence (AI). DL has seen a resurge last decade with major improvements in computer vision and language processing applications [50, 59, 6]. Access to large datasets, improved algorithms and increase of computational capacity due to use of accelerators, primarily GPUs, have been credited for the recent success.

Although the many success stories of deep learning, which has spurred wave of public and corporate interest in AI [54, 83, 3, 13], there are still many unsolved issues. One of these are how to make the DL architectures more efficient, especially to be used in embedded devices. However, the trend in DL research is that the models get larger and more complex, as can be seen in Figure 3.12. This is problematic since these models require more computations and have a larger energy consumption, which is especially problematic for embedded devices [41]. Consider AlphaGo, Google’s famous AI powered Go program that beat the world champion Lee Sedol. AlphaGo has a power consumption of 1 MW compared to 20 W for a human [80].

Counter-intuitively with the growth of model size and complexity, it has been shown that out of the millions of parameters used in common DL architectures, more than 90% of these can be removed with insignificant reduction in accuracy, leading to a much lower memory footprint for storing the model as well as less computations and significantly lower energy usage [41]. This process is referred to as pruning and can be applied to both weights and neurons and there are various methods proposed in the literature [55, 43]. The gist of this method is captured in Figure 3.13.

Another popular method to increase the efficiency of DL is reducing the numerical precision of weights and activations. This is referred to as quantization. Typically, single precision floating point numbers are used to represent weights and activations. Hwang and Sung showed that fixed-point networks with ternary
weights (+1, 0, and -1) show only negligible performance loss when compared to the floating-point counterparts [45].

By applying pruning, quantization and some other tricks Han et al. reduced the storage requirement of the network by 35-39x, 3-4x speedup and 3-7x energy efficiency. This is how important these techniques are for efficient DL. Hence they need to be considered in the context of energy efficiency for the use case of machine learning in the LEGaTO Project [42].

Robustness to hardware failure is of uttermost important for autonomous driving. How to gracefully and quickly recover from a hardware failure in the compute node? One approach would be to have a replicated fail-over system, but that would be very expensive. Another approach would be to distribute the computations of the ML system on several cheaper compute nodes in such a way that in the case of HW failure the system can continue, but with slightly less accuracy. This will be explored in the project on the edge server architecture developed in the project (see Section 6.4).

Taking all of the above into account, three main activities will be carried out
in the ML use case:

1. Implementation of state-of-the-art DL algorithm for pixel-wise semantic segmentation using the tool chain developed in the LEGaTO Project, analyse its performance and provide feedback to the library developers.

2. Develop a tool(s) that optimises a DL model for a specified target platform by applying pruning, quantization and other suitable methods. This tool will significantly ease the today tedious process of porting a research DL system to an embedded device.

3. Develop a novel distributed pixel-wise semantic segmentation system to be deployed and evaluated on the LEGaTO edge server architecture (see Section 6.4).

3.5.2. Requirements

Hardware
The purpose of the application is to be portable to vehicles and other embedded devices with limited resources. The mainstream hardware to use would be GPUs, but FPGAs have emerged as an interesting alternative for deep learning. The available test beds in the LEGaTO Project include the relevant hardware needed, both GPGPUs and FPGAs, for testing and experimentation.

Software
To efficiently run deep learning algorithms on Nvidia’s GPUs the cuDNN and cuBLAS libraries developed by Nvidia are highly optimised for common operations used for DL, e.g., convolutions and linear algebra. In addition to this we will use the popular open source deep learning library TensorFlow for the benchmark implementation.

3.5.3. Computational Characteristics, Code and Data Structures

Deep learning is loosely based on the functionality of the brain, which is why the models are often referred to as artificial neural networks. Its basic computational unit is the artificial neuron which sums its inputs, add a bias $b_i$ and last apply an activation function $\sigma$, e.g., tanh: $y_i = \sigma(\sum_j x_j + b_i)$. These neurons are organized in a network. Figure 3.14 demonstrates a simple feed forward neural network (FFNN) with inputs to the left and outputs to the right. FFNNs are typically only two to three fully connected layers (i.e., all neurons in previous layer is used as input to all neurons in the next layer) wide, or deep, while more modern architectures for image recognition can be significantly deeper, hence the term deep learning.

In computer vision, a very important operator is the convolution, in which a filter is slid over the previous image/feature map. This is demonstrated in Figure 3.15. This operation is very computationally heavy, but does not use as much memory as the fully connected layers.

The dataset to be used to train the pixel-wise semantic segmentation is the popular cityscape dataset. It consists of 5000 annotated images of road scenes. Its size is approximately 11GB [22].
The LEGaTO project has received funding from the European Union's Horizon 2020 research and innovation programme under the Grant Agreement No 780681.

3.5.4. Metrics and Optimization Goals

Multiple of the metrics in the LEGaTO Project are highly interesting for the ML application. As mentioned in Section 3.5, the energy consumption of today’s state-of-the-art perception system are too high and needs to be decreased. Thus, energy is the most important metric for the ML use case. The goal is to decrease the energy consumption of the application to 30W. Latency and throughput are also very important, since lower latency can be the difference between an accident or not. Today’s best system, when it comes to accuracy, is MASK R-CNN with a throughput of only three images per second executed on the very powerful Nvidia Tesla P100 GPU using approximately 250 W, i.e. 83 W/image [44]. Our goal is to start from the MASK R-CNN model and then optimise this model, with negligible decrease in accuracy, to predict 10 images per second with a maximum latency of 100 ms consuming 25 W, i.e. 2.5 W/image. This corresponds to a reduction of energy by 97.0%. Increased MTBF and code base security is of course also important for having operational and secure autonomous vehicles.

The roadmap of reaching the goals is to first port a reference implementation in...
TensorFlow (Python) to use OmpSs compiled for CPU, then GPU and last FPGA. We will then compare the OmpSs FPGA implementation with an implementation in DFIant. In parallel with this work, we will develop a hardware aware pruning and quantization tool that significantly will reduce the energy consumption, the model size and increase the productivity of AI research/development teams. Another major benefit of applying quantization and pruning is that the models can be deployed to cheaper hardware, enabling IoT applications to increase their level of artificial intelligence. Finally, we will develop and deploy a distributed DL model on the LEGaTO edge server architecture and optimise the model for minimal energy usage and maximum fault tolerance.

In depth experiments will be carried out and feedback will be provided to the library developers. Initial encouraging results carried out after M9, were presented during the interim review meeting, and will be presented in detail in the next delivery. We will make use of the different LEGaTO tools for the different metrics as described in Table 3.5.

3.6. Infection Research

The strategies to select the top features from thousands that can predict the cases and to estimate the appropriate sample size to get significant result. Some studies used the whole biomarkers, which are measurable values as indicators of a biological state, and risk factors that can be more the ten thousands to investigate the disease and find from them predictors to predict the diagnosis, determine the state of the disease, and measure the effectiveness of the treatment. However, because of the limitation of the funds and consuming time the researchers go to the pilot studies that mean small simple size, sometimes less or around 20 cases. This can guarantee a pure random-based correlation and find good association between the biomarkers and disease only by chance. A study used whole-genome microarray analysis to investigate the transcriptomes of periprosthetic hip tissues to identify genes that are differentially transcripted between chronic periprosthetic hip infection and aseptic hip prosthesis loosening. In this study more than fifty thousand biomarkers were analyzed for only 24 patients citeFrankKlawonn2016.

3.6.1. Application Description

Along with the advancement of high-throughput technologies that allow measuring the whole or large parts of the genome, transcriptome, proteome or metabolome, came a strong hope to find a single biomarker for each disease or state of a disease to be diagnosed with very high certainty. Some of this technologies are:

- microarrays that generate data from experiments on DNA, RNA, and protein microarrays which allow researchers to investigate the expression state of a large number of genes.
- next generation sequencing which is capable of producing large numbers of reads at exceptionally high coverages throughout the genome.
- mass spectrometry which is an analytical technique measures the masses within a sample.
However, this dream did not come true and it seems to be unrealistic from today’s point of view. Biological systems are probably too complex for simple single-cause single-effect associations. Nevertheless, there are biomarker candidates that show a high correlation with specific diseases but are not reliable enough to function as predictors for the presence of a specific disease alone. Therefore, instead of relying on a single biomarker, the idea is to combine biomarkers that are not good enough for the diagnosis of a specific alone but can jointly provide a diagnosis with high certainty.

From a formal point of view, we face the following problem. We have $n$ instances – usually patients – from which we have measured $m$ attributes (biomarker candidates). The patients are assigned to $c$ different classes, i.e., different diagnoses or different states of a certain disease. The number of classes $c$ is usually small, in many cases even $c = 2$ where we only want to distinguish patients suffering from a certain disease from patients who do not have this disease. Typically, we have $m \gg n$. Our ultimate goal is to find a classifier that can predict the class (diagnosis) based on the values of the $m$ attributes. Due to the fact that we have to face $m \gg n$, we cannot directly build a standard classifier based on the given data set [48].

One application scenario are biomarker pilot studies with very small samples sizes that do often not target a classifier to predict the correct diagnosis. Instead, their intention is to check whether it is worthwhile to continue to collect more samples in order to construct a classifier based on such a larger data set. We have developed computation intensive methods based on Monte Carlo simulation and permutation tests that can check the potential of such small biomarker pilot studies and also give an estimate of the required sample size to obtain a sufficiently reliable classifier from the larger sample.

Once the sample size is large enough to construct a classifier, feature selection techniques are required to reduce the number of attributes (biomarker candidates) drastically. This is the second application scenario. As a possible way to evaluate the predictive power of a classifier, we could apply cross-validation and because of the small sample size we would prefer to use leave-one-out cross-validation which is again computationally expensive.

3.6.2. Requirements

The requirement that is needed to be computed is a dataset with a matrix of numeric features, the class labels is at the last column. Class labels could be numerical or character values but should be considered as a factor that should not have more than ten levels. The dataset should be without missing values or at least has already been treated by suitable imputing methods that replace missing values by appropriate estimates. The most intensive part in the computations are:

- Computing the performance measures for single biomarkers – i.e., how much single biomarkers can contribute to predict the diagnosis – analysing the correlation between the features (biomarker candidates) within each group (diagnosis).
- Training the prediction model.
The applications depend on the following R packages:

- **Biocomb** contains functions for the data analysis with the emphasis on biological data, including several algorithms for feature ranking, feature selection, classification algorithms with embedded validation procedures.

- **ROCR** and **pROC** provide classification performance measures like AUC and multiclass AUC.

- **discetization** provides the probably most prominent performance measure for classification problems: Shannon entropy.

- **e1071** contains various classifiers like naive Bayes or support vector machines.

- **randomForest** is required for random forest, a classifier that in many of our applications turned out to yield very good results.

- **MASS** is used for linear discriminant analysis, a classifier very popular in statistics.

- **ggplot2** is a rich package to create graphics that we need for visualisation of our results.

### 3.6.3. Computational Characteristics, Code and Data Structures

Figure 3.16 shows the typical workflow of our biomarker discovery projects. The green parts in the diagram correspond to computations (except for the bottom right box which only includes documentation).

The global evaluation of biomarker candidates requires to compute suitable classifier performance measures like AUC, which is the area under the receiver operating characteristic curve, or entropy for thousands of features. Checking whether there are more “good” biomarkers than expected requires large numbers of Monte Carlo simulations or carrying out computationally expensive permutation tests. The Monte Carlo simulations as well as the permutation tests require again to compute suitable classifier performance measures now in the range of up to 1000 times more than for the given dataset.

Judging the potential of biomarker combinations requires correlation analysis which is again a bottleneck because its runtime depends quadratically on the number of biomarker candidates. In addition, the common Pearson correlation is often not suitable due to very skewed distribution, so that rank correlation coefficients like Spearman’s rho or Kendall’s tau should be used, which can better cope with skewed distributions but are based on combinatorial considerations and are therefore computationally more expensive than the Pearson correlation. The sample size estimation again needs Monte Carlo simulations or permutation tests.

Classifier construction and evaluation requires also extensive computation. Our often preferred method of random forests is based on hundreds of decision trees and the evaluation is usually based on the leave-one-out cross-validation.
Figure 3.16. Typical workflow of a biomarker discovery project

<table>
<thead>
<tr>
<th>biomarker</th>
<th>AUC</th>
<th>positive correlation</th>
<th>probability</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>0.8840909</td>
<td>FALSE</td>
<td>7.647916e-08</td>
</tr>
<tr>
<td>2</td>
<td>0.8750000</td>
<td>TRUE</td>
<td>1.803840e-07</td>
</tr>
<tr>
<td>3</td>
<td>0.8647727</td>
<td>TRUE</td>
<td>4.506965e-07</td>
</tr>
<tr>
<td>4</td>
<td>0.8647727</td>
<td>TRUE</td>
<td>4.506965e-07</td>
</tr>
<tr>
<td>5</td>
<td>0.8534091</td>
<td>TRUE</td>
<td>1.178612e-06</td>
</tr>
</tbody>
</table>

Table 3.2. Top 5 AUC values, the correlation of this biomarkers with the disease and the probability

technique where the construction of the classifier has to be repeated once per sample in the dataset.

In the following, we present some of the core functions in our application that are executed up to a million times or even more. Run times that depend on the dataset refer to an example dataset of moderate size in terms of the biomarker candidates (here 2000).

An example for a classifier performance measure is AUC which is computed by the function `compute.aucs`. This function also yields the probability that a corresponding (high) AUC value could occur in a random dataset which has no association between the biomarker candidates and the classes (diagnoses) to be predicted. This function needs three other functions to run, it is almost found in all other functions and takes around 4 seconds and the CPU utilization while running this function is 45%. Table 3.2 shows the highest 5 AUC values and their probability for a total data which consist of 2000 biomarkers and 62 patients citealonData. Such AUC values might not be sufficient for medical test. Nevertheless, a combination of biomarkers with such high AUC values might lead to a classifier with sufficient predictive power.
The function `auc.combi` is required for sample size estimation. It computes the classifier performance measure AUC for a combination of selected biomarkers taking their correlations into account. This function needs two other functions to run and it takes around 0.1 seconds with double precision and the CPU utilization while running this function is 17%.

```r
1 auc.combi=function(data,centralTendency="mean", method.cov="standard")
   {
2   delta.mu = differences.means(data,method=method.centralTendency)
3   covam = cova.mat(data,method=method.cov)
4   return(pnorm(sqrt(t(delta.mu) /%*% solve(covam\$sigma+covam\$sigma1)
5   /%*% delta.mu)))
```  

The function `combinedAUC` extends the function `auc.combi` and tests different numbers of biomarker combinations to select the optimal number of biomarkers to be combined for the prediction of the diagnosis. This function needs three other functions to run and the run time depends on the number of features. The CPU utilization while running this function is 47%.

Figure 3.17 visualizes a result of the application of the function `combinedAUC`. The x-axis shows the number of biomarkers to be combined and the y-axis the resulting AUC value.

![AUC of features combination](image)

*Figure 3.17: auc of features combination*

`CI.auc.trueFeatsComb` computes a confidence interval for the AUC, i.e., a lower and an upper bound for the AUC of a combination biomarkers (in a possible test dataset, given the actual dataset is representative) see Table 3.3. The confidence interval computation is based on bootstrap sampling which can be seen as a special type of Monte Carlo simulation. Again, a large number of simulations is needed here. This function needs three other functions to run and takes around 15 seconds and the CPU utilization while running this function is 45%.
<table>
<thead>
<tr>
<th>No. combined features</th>
<th>combined AUC</th>
<th>upper band</th>
<th>lower band</th>
</tr>
</thead>
<tbody>
<tr>
<td>5</td>
<td>0.9264723</td>
<td>0.9264723</td>
<td>0.9264723</td>
</tr>
<tr>
<td>10</td>
<td>0.9436745</td>
<td>0.9436745</td>
<td>0.9436745</td>
</tr>
<tr>
<td>20</td>
<td>0.9625266</td>
<td>0.9625266</td>
<td>0.9492810</td>
</tr>
<tr>
<td>25</td>
<td>0.9948289</td>
<td>0.9948149</td>
<td>0.9768047</td>
</tr>
<tr>
<td>30</td>
<td>0.9989070</td>
<td>0.9989013</td>
<td>0.9917759</td>
</tr>
</tbody>
</table>

Table 3.3. confidence interval of AUC for a specific number of biomarkers after combining

Finally the function `calculate.sampleSize` estimates the required sample size to validate biomarkers from a small pilot study in an extended study and also to validate a classifier that is based on the combination of the corresponding biomarkers. This function needs two other functions to run and takes around 0.1 seconds and the CPU utilization while running this function is 17%.

The time complexity of the loop is considered as $O(1)$.

**Loop Structures**

There are various loops, most of them having a high potential for parallelization.

- The classifier performance measure, e.g., AUC, needs to be calculated for each biomarker candidate, in Monte Carlo simulation for randomly generated biomarkers and during permutation tests repeatedly for biomarkers with “shuffled” diagnoses. Also within the leave-one-out cross-validation for classifier evaluation, the classifiers performance measures needs to be calculated repeatedly.

- The same applies to the correlation between biomarker candidates.

- The above-mentioned loop structures can be easily parallelized because the computation within each repetition of the loop can be seen as independent. The computation of the combined performance measure is an incremental computation and needs additional work to be rendered for efficient parallelization.

**3.6.4. Metrics and Optimization Goals**

A main optimization goal is a significant speed-up which would not only mean to obtain results faster but also to enable us to handle larger data sets (especially in terms of biomarker candidates) which cannot be handled by the implementation we use at the moment. As an example we have simulated the entropy values for only 3 biomarkers 166 times for 66 observations and 4 classes which took 14.89 hours. Real datasets have more than 50,000 biomarkers. The estimated energy consumption for the calculation of 3 biomakers was 1.26kWh. An additional benefit would be that we also save energy once the modified code can run faster. As performance measures, we propose the factor of speed-up
and we would also estimated the saved energy. The runtime of our original implementation was so long that sometimes errors occurred, so that computations had to be restarted and repeated. Therefore, we are also interested in code base security in terms of number of failures.

First prototypical implementations of some of our basic functions by MAXELER showed that there is a high potential for a very good speed-up. This speed-up might even enable us to analyse datasets we could not touch yet, would drastically reduce the energy consumption for datasets as we use them now and would also lead to better code base security. The Forschungszentrum Jülich provides a MAXELER-based platform to which we can get access. Our plan is to re-implement parts of our R code so that it can run in Jülich.

### 3.7. Secure IoT Gateway

When it comes to Industry 4.0 and Internet of Things (IoT), more and more devices communicate with each other through different networks such as local area networks (e.g., company network with its subnets) and wide area networks (e.g., internet and remote company networks). Depending on the purpose of the machine using IoT, there is a big need to secure the communication and prevent possible attacks. A successful attack can cause big damage. For example: Manipulation in traffic controlling systems can switch all traffic lights to green during rush hour. Because attackers can use any part of the communication, the only way to achieve secure communication is to protect the whole way from the sender to the receiver. This goal is associated with a high level of complexity and expert IT know-how.

![Comparison of connections with and without the Secure IoT Gateway](image)

*Figure 3.18. Comparison of connections with and without the Secure IoT Gateway*

The general goal of the Secure IoT Gateway is to provide a solution for securing network connections of local and/or remote network devices, with a focus on IoT usage while easing the use as much as possible. Companies benefit from
this solution because they can secure their IoT communication in an easy way without handling the complexity themselves.

The Secure IoT Gateway will not directly improve the general project objectives like energy efficiency because it is not a classical application. It helps other use cases to achieve their goals by reducing the complexity of security for them.

### 3.7.1. Application Description

In the LEGaTO project the Secure IoT Gateway takes a special place, as it is not a classical use case. It covers a common complex problem: The safety of IoT communication. The Secure IoT Gateway is therefore not a real application but an enabler for the other use cases, so that they do not need to handle that complexity themselves. Within the development of the Secure IoT Gateway, no compute-intensive parts will be programmed or modified, but a homogeneous, management software which will be described in the next subsections. To do this, all necessary OS, security and networking components will be brought together, connected and configured carefully. The only compute-intensive part of the Secure IoT Gateway is the encryption of the VPN network stream which has been evaluated within the first months of LEGaTO. The preliminary outcome is that modern CPUs can handle this encryption through their integrated hardware encryption units. We see no possibility to optimize this further, thus the Secure IoT Gateway architecture will not be optimized by using one of the programming models as listed in Table 3.5.

The Secure IoT Gateway is not a single application. It consists of four components that work together to create a secure environment for communication. A human user (normally a system admin) will only see and use the Network Cockpit component to configure the system including the other components. The other three components do not need human interaction.

The core concept of the Secure IoT Gateway is that any communication between network devices must be routed through one of the Secure IoT Gateway devices and will be encrypted on-the-fly. Any network traffic can be controlled by rules.

The system is based on four specialized components, with each having an individual task within the system:

- **Network Cockpit** Allows users to configure the system components and to view and assess the components status in a web application. The Network Cockpit is multi-client capable.

- **Gateway Cluster** The central component for providing gateways. Gateways ensure communication between different locations and regulate communication between clients through an integrated firewall. Connections can be established via Network Gateways and IoT Bridges. The Gateway Cluster is multi-client capable.

- **Network Gateway** A local network component, connecting the existing network with a Gateway Cluster and allowing connections by IoT Bridges and other VPN clients. Connections between the components can be regulated. The Network Gateway is not multi-client capable.
IoT Bridge Allows a safe connection of devices to the network. The encrypted connection can be established with a Gateway Cluster or a Network Gateway. The IoT Bridge is not multi-client capable.

3.7.2. Requirements

The Secure IoT Gateway is the enabler for the other use cases. It provides secured network connections between distributed devices which the other use cases can rely on. Therefore, the Secure IoT Gateway is a requirement for other use cases. During the project it will be evaluated which use case will use the Secure IoT Gateway within their project setting.

To run the Secure IoT Gateway it is necessary to have a basic IT infrastructure that connects your devices to the local network and the internet.

Each Secure IoT Gateway component has different hardware requirements that will be listed in the next section where each component is described in more detail. In general, the hardware requirements can differ in real project setups depending on the amount and type of devices that will be connected to it.

3.7.3. Computational Characteristics, Code and Data Structures

The Secure IoT Gateway is different from the other use cases in this project. It is more a tool that helps to simplify the complexity of security for other use cases. The computational characteristics, code and data structures are best described by its components. Therefore all components are described in this section.

3.7.3.1. Network Cockpit

The Network Cockpit allows users to configure all devices (Gateway Cluster, Network Gateways and IoT Bridges) of secured networks and to view and assess their status. It is multi-client capable.
Technical Layout
The Network Cockpit runs on a Linux system. Multiple Docker Containers provide the basis, with each taking individual tasks. The programs MariaDB, RabbitMQ and NGINX run inside the containers, making the Network Cockpit available as a web application. The Network Cockpit is realized using PHP 7, HTML, CSS 3 and ECMAScript 6. The communication with Network Gateways and the Gateway Cluster takes place via RabbitMQ and a REST-API.

Within the different languages, the following Frameworks are used:

- Symfony 4 (PHP)
- Bootstrap (HTML, CSS 3)
- Vue.js (ECMAScript 6)

Functions
Status - One of the most important functionalities is to get an overview of all connected Gateways and IoT Bridges subordinated to them. For each of these devices it will be possible to see its status as well as detailed information of how it is connected to the system, usage statistics, security information and more.

Provisioning - In order to ease the configuration and installation process a provisioning service will be available. Local IoT Bridges and Network Gateways can auto configure themselves with the help of a preregistering service. The configuration will be customized in advance and the device authenticates itself at the provisioning service with its serial ID and gets the corresponding configuration data.

Network configuration - All devices in the network can be managed through the Network Cockpit. Devices can be enabled and disabled, the configuration can be changed and rules can be configured.

Rules - For each device rules can be set. A rule determines if network traffic to or from the device is allowed or disallowed. Rule can rely on time, network characteristics like ip ranges and more.

Hardware Requirements
The Network Cockpit has common requirements regarding the hardware. A system similar to the following configuration is recommended:

- Intel or AMD CPU (x86-64), minimum of 4 Cores
- At least 16 GB RAM
- At least 32 GB HDD
- 1 Gbit/s (better 10 Gbit/s) Ethernet network

The software can also run as a virtual machine within an existing system or on the Gateway Cluster. The hardware requirements are independent of the amount of IoT devices.

3.7.3.2. Gateway Cluster (optional)
The Gateway Cluster is the central component for providing gateways. Gateways ensure communication between different locations and regulate communication between clients through an integrated firewall. Connections can be established via Network Gateways and IoT Bridges. The Gateway Cluster is is multi-client
capable.

**Technical Layout**
The Gateway Cluster consists of several sub-components. Together, they provide the required functions. A Linux/KVM-based virtualization system constitutes the basis. Inside the virtual machine, a Master Gateway provides the network connection for the Customer Gateways and allocates the public IP addresses. Clients are automatically supplied with an autonomous gateway, separate from the other clients. The Gateway Cluster is a server based on x86-64 with several LAN ports.

**Cluster Functions**

- Automatic configuration via Network Cockpit
- Provisioning of the virtualization platform based on Linux KVM
- Automatic provisioning of Customer Gateways based on OPNSense
- Automatic provisioning of a connection to the Master Gateway
- Automatic provisioning of further network connections, e.g., to a VLAN within a data center (optional)
- Provisioning of status and performance information

**Master Gateway Functions**

- Automatic configuration via Network Cockpit
- Allocation of local IP addresses to the Customer Gateways for provisioning
- Providing a connection to the provisioning server and to WAN
- IDS/IPS

**Customer Gateway Functions**

- Automatic provisioning and configuration via Network Cockpit
- Providing status information
- Providing VPN servers for network and IoT bridges
- Management of network and IoT bridge communication via rules
- Managing communication to the connected networks

**Hardware Requirements**

Hardware requirements for the Customer Gateway are following recommendations by OPNSense[65]. Effects of overbooking of cores must be researched during the course of the project.

- Intel or AMD CPU (x8664) with Intel VT/AMD-V support
- 2 CPU cores and 4 GB RAM for the base system
- 2 CPU cores and 4 GB RAM for the Master Gateway
- Gigabit Network for management
- 10GB network for Gateway VMs

The hardware requirements do correspond to the amount of VPN connections. During the project it will be analysed how to determine the needed hardware for the Gateway Cluster for a given amount of connections.
3.7.3.3. Network Gateway

The Network gateway is a local network component, connecting the existing network to a Gateway Cluster and allows connections of IoT Bridges. Connections between the components can be regulated. The Network Gateway is not multi-client capable. The Network Gateway is a server based on x86-64 with several LAN ports.

**Functions**

- Automatic provisioning and configuration via Network Cockpit
- Providing status information
- Providing a VPN server for IoT bridges
- Establishing a VPN connection to the Gateway Cluster (optional)
- Managing network communication via rules
- IDS/IPS

**Hardware Requirements**

Hardware requirements for the Customer Gateway are the same recommendations by OPNSense from above and do also correspond with the amount of VPN connections.

3.7.3.4. IoT Bridge

An IoT Bridge allows a safe connection of devices to the network. It is a small device that is placed between the secured network and the device that should be connected to the IoT Gateway. The encrypted connection can be established with a Gateway Cluster or a Network Gateway. An IoT Bridge is not multi-client capable.

![Figure 3.20. Possible IoT Bridge hardware](image)

**Functions**

- Automatic provisioning and configuration via Network Cockpit
  - Connection to Network Cockpit over network
  - Connection to IoT device over network
  - Locating the Network Cockpit over DHCP, DNS and a centralized redirect server
  - When problems arise, a fixed IP will be placed onto the LAN interface automatically.
- Providing status information
- Establishing a VPN connection to the Gateway Cluster or Network Gateway
Managing network communication via rules

**Hardware Requirements**

A potential IoT Bridge hardware is shown in Figure 3.20.

- ARM or MIPS-based CPU
- 128 MB RAM
- At least 2 x 1 Gbit/s Ethernet network

The hardware requirements do not correspond with the amount of IoT devices.

**3.7.4. Metrics and Optimization Goals**

From a user perspective the main optimization goal is to provide a solution with almost no configuration effort. The Secure IoT Gateway architecture should be implementable in a (semi-)automatic way. This is the reason why there are no special metrics from this point of view that could show the quality of the outcome of the Secure IoT Gateway task. The only key metric that is important here will be the (subjective) user feedback.

From a technical perspective it is crucial that the added security layer will affect the network performance as less as possible. The Secure IoT Gateway may establish many VPN connections where all traffic must be encrypted on the fly. Compared to unencrypted communication, this encryption needs computational time, that shows as latency and potentially limited bandwidth.

Encryption can be done very efficient by modern CPUs or additional acceleration devices, like encryption cards or FPGAs. It is expected that an ideal setup for typical sizes can be achieved by a careful selection of the right hardware components.

Additional to the latency the network throughput is another metric that needs to be optimized. If only the latency will be optimized, a simple solution is to reduce the throughput. Less throughput means less computational effort leading to less latency. Less throughput would mean that less information can be transported in the same time. This would have many disadvantages. Longer messages would take more time to communicate them and less IoT devices could communicate at the same time. Therefore it is important to keep the maximum throughput as high as possible whilst keeping the latency as low as possible. The importance of both metrics and the relation of them strongly depends on the use case.

Besides an optimization with the right choice of hardware setting, there will also be an optimization by selecting the right software to achieve a high network throughput. In order to rate the choice of technology to implement a secure and effective VPN, a benchmark comparison between WireGuard and OpenVPN was made. WireGuard aims to provide a secure VPN being pretty simple to configure and highly effective using Curve25519 for key exchange, ChaCha20 and Poly1305 for data authentication and BLAKE2s for hashing - this combination is in NaCL (part of the Noise protocol framework) an alternative to AES which is used by OpenVPN. In this benchmark test, WireGuard and OpenVPN were both tested using embedded hardware with an Intel Xeon D-1518 processor with four cores and 2.2 GHz. The installation of WireGuard is in fact very simple with the “Quick Start Guide” while the installation of OpenVPN is more complex. Out of the box,
WireGuard is faster than OpenVPN. OpenVPN can be configured to assimilate to WireGuard: * choose the 2p2 modus * set the MTU of the TUN device to 64k * choose UDP instead of TCP/IP as protocol * add fast-io * deactivate the internal fragment * choose AES-256-CBC as cipher With this configuration and four parallel tunnels, OpenVPN and WireGuard both saturate 1GbE. But after upgrading the network to 10GbE, OpenVPN has a bandwidth of 6.8 Gbits/s using four parallel tunnels while WireGuard just reaches 2.1 Gbits/s.

The overall optimization will probably be to find the right configuration setting of software and hardware components for the given use cases. Thus, there is no need for using the LEGaTO toolflow.

3.8. Summary

The previous sections gave a detailed description of the use cases of the LEGaTO project. Four of these use cases are application oriented and their purpose is to develop optimised application demonstrators and evaluate the LEGaTO tools. The fifth use case is a Secure IoT Gateway that is used as an enabling technology that adds extra security and acts as integration example. It is therefore omitted in the following optimization-oriented comparisons. First, we give a summary of the four application use cases in terms of their programming languages, their components targeted for optimization and their high-level optimization goals as shown in Table 3.4.

<table>
<thead>
<tr>
<th></th>
<th>Smart Home</th>
<th>Smart City</th>
<th>Machine Learning</th>
<th>Infection Research</th>
</tr>
</thead>
<tbody>
<tr>
<td>language(s) used</td>
<td>C, C++, Python, Javascript</td>
<td>Fortran90</td>
<td>C, Python</td>
<td>R</td>
</tr>
<tr>
<td>components targeted for optimization</td>
<td>face, object, and speech recognition</td>
<td>Element-matrix construction and SpMV</td>
<td>Matrix-Matrix Multiplication, 2D Convolutions</td>
<td>CutIndex</td>
</tr>
<tr>
<td>optimization goal</td>
<td>Taskifying using OmpSs for CPU, GPU and FPGA and porting to embedded edge server</td>
<td>Porting to C/HLS and OmpSs-FPGA taskification</td>
<td>Taskifying using OmpSs for CPU, GPU and FPGA</td>
<td>Implementing relevant code parts with MaxJ</td>
</tr>
</tbody>
</table>

In the following we provide a more detailed view of the individual optimiza-
tion goals of all applications by mapping them to optimization objectives as described in section 3.2 and indicate which programming models and technologies that might be explored to achieve these objectives. This is illustrated in Table 3.5. As was noted in the individual application sections, not all programming models are equally relevant to all applications and we described which programming models and optimization strategy appears to be most promising.

Table 3.5. Optimization objectives for each use case

<table>
<thead>
<tr>
<th>Programming Model</th>
<th>Smart Home</th>
<th>Smart City</th>
<th>Machine Learning</th>
<th>Infection Research</th>
</tr>
</thead>
<tbody>
<tr>
<td>OmpSs, XiTao</td>
<td>1,4,7</td>
<td>1,2,4</td>
<td>1,2,3,5,6,7</td>
<td>1,3</td>
</tr>
<tr>
<td>MaxJ</td>
<td>1,4,7</td>
<td>1,4</td>
<td></td>
<td>1,3</td>
</tr>
<tr>
<td>DFiAnt</td>
<td>1,4,7</td>
<td></td>
<td>1,4</td>
<td></td>
</tr>
</tbody>
</table>

As shown in Table 3.5, all applications target energy optimizations as one of their objectives. There therefore provide a more detailed overview of the energy baselines and optimization targets in Table 3.6.

The initial outcome of the optimization and evaluation described here will be reported in deliverable D5.2 “First report on development and optimization of use-cases” which will be completed at M20. The final results we be reported in D5.3 and D5.4 at M36.
Table 3.6. Energy baseline and optimization targets.

<table>
<thead>
<tr>
<th>Energy Baseline</th>
<th>Smart Home</th>
<th>Smart City</th>
<th>Machine Learning</th>
<th>Infection Research</th>
</tr>
</thead>
<tbody>
<tr>
<td>Energy Baseline</td>
<td>Smart Mirror prototype - Face, gesture, and object detection running with 18 FPS each, with a power consumption of 650 Watt on a desktop workstation</td>
<td>2.7 Wh on one single MN4 node for baseline test after 100 iterations</td>
<td>83 Joule per image on Nvidia Tesla P100</td>
<td>1,26 kWh (14.89 hours) for 3 Biomarkers (a real data set has up to 50 000 biomarker candidates)</td>
</tr>
<tr>
<td>Energy Optimization Target</td>
<td>embedded edge Server with 50 Watt for a suitable performance</td>
<td>One order of magnitude lower (≈0.3 Wh)</td>
<td>2.5 Joule per image, i.e. energy reduction of 97%</td>
<td>Increase of energy efficiency by factor 5</td>
</tr>
</tbody>
</table>
4. Definition/Design of the Front-End Toolbox

4.1. Executive Summary

This chapter presents the functionalities to be developed in LEGaTO’s Work Package 4 (Tool-Chain Front-End). It essentially comprises the deliverable D4.1 “Definition/Design of back-end Runtime System (M9)” of the original plan. The chapter is structured in three main parts. The first part presents the general software architecture for the front-end toolchain. The second part contains a detailed description of LEGaTO’s programming model and a number of annotations and extensions to be developed throughout the project. The third and final part details the extensions to be added to infrastructure software in order to support LEGaTO’s task-based programming model.

4.2. Introduction

The front-end tools of LEGaTO offer a programming interface for the use case applications as described in Chapter 3 and will be supported by a correspondent runtime system as described in Chapter 5, developed in Work Package 3 (Tool-Chain Back-End). Before describing in detail each of all the functionalities (both font- and back-end), in the text that follows we briefly introduce LEGaTO’s architecture as a whole.

The LEGaTO programming model front-end is shown on the left side of Figure 4.1, depicting the overall software toolchain for the cluster runtime. The LEGaTO front-end consists of the tools that process the source code and generate the LEGaTO binary targeting the heterogeneous platforms. These tools include a Mercurium-based toolchain to analyze OmpSs source code and generate Nanos/XiTAO/FPGA/GPU binaries, and two high level programming methodologies to generate dataflow kernels: DFiant and MaxJ. Mercurium [7] has been previously developed at the Barcelona Supercomputing Center as a source-to-source compilation infrastructure aimed at fast prototyping.

The LEGaTO execution model works as follows. A user runs a LEGaTO application via the SLURM job scheduler. The job scheduler assigns a partition of resources that consists of several heterogeneous nodes interconnected by a configurable network. The application then starts executing on the partition and begins by querying/discovering the available hardware. Initially, the hardware will be partitioned statically and managed by different LEGaTO runtime components, such as the CUDA libraries, FPGA libraries, the Maxeler environment, the Nanos library and the XiTAO library. At a later stage we will consider a dynamic implementation in which hardware (and nodes) can be added and removed at runtime. Once all the runtime components are active, the OmpSs master thread begins. From this moment onwards, asynchronous tasks may be created. In the LEGaTO execution model all tasks are offloaded to available runtime components where they are executed in a non-preemptive way. A dynamic runtime scheduling component then takes decisions on where to execute particular tasks, depending on the availability of hardware, the efficiency of executing a task on a particular hardware, and considerations such as parallelism, overheads and locality.
The sections that follow in this chapter will present a comprehensive set of functionalities to be offered as front-end tools for programming applications in LEGaTO. Its software architecture specification will introduce all main aspects on which the project is focused as fault-tolerance, heterogeneity, multicomputer execution and energy efficiency. It follows with many details on the programming models to be developed, extended and integrated in the context of the project. It will conclude with a number of extensions to be implemented in the infrastructure management (OpenStack) in order to support the execution of the proposed task model.

### 4.3. Software Architecture Specification

In this section we briefly present OmpSs, the main programming model of LEGaTO’s project, as it will be extended according to the project’s objectives. In particular, it sheds light on a distributed memory version of OmpSs, being developed in LEGaTO to support dynamic scheduling of tasks across multiple computing nodes. It also defines how LEGaTO will tackle energy-efficiency and fault-tolerance, relating the front-end interface with the back-end implementations. Then the two last subsections present extensions for integrating the use of DFiant and Maxeler’s MaxCompiler for developing applications including code in FPGAs.

#### 4.3.1. The OmpSs Programming Model

OmpSs is the programming model that BSC proposes to be further developed in the LEGaTO project, in order to target programming for heterogeneous architectures. OmpSs is used as a prototype platform to develop extensions to OpenMP, and demonstrate their usefulness. BSC has used OmpSs to prototype OpenMP...
tasking, tasking dependences, task priorities, and task loops, characteristics that are now present in OpenMP 4.5. Task reductions has been also prototyped with OmpSs, and they will become available in the upcoming OpenMP 5.0 specification.

Regarding the support for heterogeneous architectures, OmpSs supports spawning tasks in GPUs (with CUDA and OpenCL kernels), and FPGAs, with kernels generated using High-Level Synthesis (HLS) tools and OpenCL. In the CUDA and OpenCL cases, the OmpSs compiler (Mercurium) allows to leverage existing kernels with OmpSs tasking. In the situations where HLS can be used, there is no need to use CUDA or OpenCL to express the accelerated kernels, but they can be written in C/C++.

Figure 4.2 shows the compilation environment for OmpSs@CUDA, where the Mercurium compiler places the compiled CUDA kernels inside the final binary. In this case, the kernels are compiled with the Nvidia CUDA compiler. The use of directives avoids the need to implement the data transfers by hand in the application code.

Figure 4.3 shows the compilation environment for OmpSs@OpenCL. In this case, the Mercurium compiler is used to provide the kernel in its source form to the OmpSs runtime system (Nanos). The data transfers and the kernel compilation and execution are implemented using the OpenCL interface provided by the vendor of the accelerator (GPU, FPGA).

Figure 4.4 shows the use of the Xilinx synthesis tools from C/C++, and the OmpSs autoVivado ecosystem. This infrastructure allows to compile, and synthesize FPGA IP cores, using the Xilinx Vivado HLS and Vivado tools. Those synthetized kernels can then be executed on the FPGA from the Nanos runtime system. In this case, Nanos is using the low-level xtasks infrastructure, also developed with OmpSs.

BSC is currently transitioning from OmpSs to a new version, named OmpSs-v2. OmpSs-v2 shares much of the characteristics with OmpSs. It also improves the
Figure 4.3. OmpSs@OpenCL compilation environment

Figure 4.4. OmpSs@FPGA compilation environment, based on autoVivado
support for task dependences, allowing better management of nested tasks, with the weak-dependences approach [69]. The support for HLS and for computer clusters (no shared memory) will be developed in the context of LEGaTO.

4.3.2. Multicomputer (OmpSs@Cluster)

OmpSs@Cluster is the distributed memory version of OmpSs, which enables dynamic scheduling of tasks across multiple nodes. OmpSs@Cluster is based on the OmpSs-v2 programming model [69], which improves the efficiency of task nesting, compared with OmpSs, via early release of dependencies and weak dependencies. These features are necessary for scaling to large numbers of nodes.

Early release of dependencies refers to the ability to incrementally release dependencies before the task and all its subtasks complete. This is possible, for example, when an array is only accessed (e.g., written) near the beginning of the task, in which case the dependency on it can be explicitly released once access to the array is no longer needed. In addition, the body of a task can complete without first having to wait for all of the task’s subtasks to complete. Any task that is dependent on the body of the task, but not an incomplete subtask, can begin execution immediately. If it is dependent on an incomplete task, the dependency is transferred accordingly.

Weak dependencies refers to the \textit{weakin}, \textit{weakout}, and \textit{weakinout} dependency types, which specify that data is required by subtasks but not the task itself. This means that the task can begin execution immediately, which means that it can immediately begin creating subtasks, resulting in increased parallelism discovery, which is very natural for a large number of problems.

4.3.2.1. Nanos 6 runtime system on distributed memory

OmpSs-v2 is implemented in Nanos 6, which is the successor of the Nanos++ runtime system. Initial development of distributed memory support was done in the ExaNoDe project [9].

Nanos 6 distributed memory model

The OmpSs@Cluster memory model uses a single virtual address space that is identically mapped in all compute nodes that are running the application. This means that a task is able to access a given data region using the same pointer, irrespective of the node on which it executes. The programmer also does not need to explicitly program data transfers to copy data between nodes, as this is done (using MPI) by the Nanos 6 distributed memory runtime.

Figure 4.5 shows the layout of the virtual memory of the cluster nodes managed by the Nanos 6 runtime system. During initialization Nanos 6 maps in every node a virtual memory region (VMR) large enough to handle the maximum memory requirements of the OmpSs application. The starting address of these VM regions is the same on every node. This means that all nodes can address the same data, with dependencies satisfied by data transfers performed by the runtime system, without address translation between nodes. Memory requests are served through custom allocators of the Nanos 6 runtime system. Subsequently, Nanos 6 divides each VMR into two distinct regions, which have different allocation semantics. The lower addresses of the VMR are reserved for conventional
memory allocations performed on that node, i.e., stack and heap allocations. Nanos 6 divides this set of addresses equally among the nodes of the cluster. This means that every address within this region is used to store the data allocated by one particular node of the cluster. The rest of the nodes of the cluster use the same addresses whenever they need to access the data, which simplifies the process of moving data around the cluster, since address translation is not needed. The higher addresses of the VMR are reserved for distributed allocations. An allocation from this memory region is implemented inside the runtime system as a collective operation across all nodes of the cluster.

**Nanos 6 execution model**

The OmpSs application begins executing on a master node, in the same way as it would do on shared memory. The code is executed serially and whenever a task directive is encountered, a new task is created and it becomes available for concurrent execution once its dependencies are resolved.

When running on distributed memory, the Nanos 6 scheduler can also offload tasks to another node, once ready for execution, i.e., when all strong dependencies have been resolved. During execution, the scheduler takes decisions regarding the node onto which the task should be offloaded. Before a remotely-executed task executes its body function, the runtime system copies any non-node-local data to the node on which the task will execute. The programmer needs to declare all the dynamically allocated data that the task uses and the way that the task will handle them using the OmpSs-v2 dependencies clauses. When executing on distributed memory, in addition to declaring the dependencies among tasks, these clauses provide the information that the runtime needs in order to perform any necessary data transfers before executing a task. This
scheme allows Nanos 6 to handle all data transfers without the need of a software directory, which simplifies its design and implementation and minimizes the amount of communication among the cluster nodes.

**Communication Layer**

The implementation of Nanos 6 requires communication among the cluster nodes for exchanging command and data transfer messages. Command messages include all the messages for offloading tasks, synchronization of nodes, sending information regarding the data locations and initiating data transfers. Data transfer messages are used to transfer data regions among nodes.

The communication layer of Nanos 6 operates as an abstraction layer that decouples the rest of the runtime system from the communication layer implementation. The current communication layer of Nanos 6 uses standard MPI.

**4.3.3. Energy-efficiency**

The LEGaTO system will obtain information about the energy efficiency of the tasks running on the system from two different sources. At the programming model level, OpenMP tasks can be already annotated with a priority level. This information can be used by the runtime system to map these high priority tasks to higher performance accelerators. In the reverse situation, tasks with lower priority can be mapped to higher energy efficient accelerators, that can be getting lower performance.

LEGaTO will implement scheduling policies dealing with the information given in the annotations. The implemented runtime will collect live measurements of performance and energy consumption from executing tasks and will decide accordingly.

**4.3.4. Fault-tolerance**

Although hardware-based fault-tolerance mechanisms such as ECC (for memory) and CRC (for interconnect) incorporate strong fault detection and correction properties, they do not have sufficient fault coverage for large distributed applications. They do not provide protection from a large class of faults such as those that impact the combinational logic that are dominant in current computing substrates. In addition, programmer annotations can help to boost reliability in a more specialized way, making the technique much more energy efficient. Therefore, software approaches are indispensable.

In the LEGaTO project, we will develop a fault-tolerance approach that will be at the compiler and runtime levels of the computing stack, supported by application level programmer supplied annotations. In the frontend section, we will describe the application-level programming model annotations for fault-tolerance as well as the compiler-based LEGaTO approach. The runtime approaches are detailed in the backend chapter, Section 5.7.

**4.3.5. DFiant: A High-level Hardware Description Language**

DFiant is a modern hardware design language (HDL) that incorporates modern programming concepts including object-oriented programming, type-safety, and polymorphism. The goal of DFiant is to improve designer productivity by decou-
pling the functionality of a design from constraints derived by design targets, predominantly target device, desired frequency, and I/O characteristics. Focusing on design functionality allows designers to express truly portable and composable hardware designs.

DFiant decouples functionality from timing constraints (in an effort to end the “tyranny of the clock” [86]). DFiant offers a clean model for hardware construction based on its core characteristics:

(i) a clock-agnostic dataflow model that enables implicit parallel data and computation scheduling; and

(ii) functional register/state constructs accompanied by an automatic pipelining process, which eliminate all explicit register placements along with their direct clock dependency.

DFiant borrows and combines constructs and semantics from software, hardware and dataflow languages. Consequently, the DFiant programming model accommodates a middle-ground approach between low-level hardware description and high-level sequential programming.

### 4.3.5.1. Improving Portability by Eliminating Clocks and Derived Registers

Clocking directs the design of synchronous hardware. Nevertheless, the clock itself is typically not part of the functionality of a design, but rather a performance and power constraint. In principle, a specific functionality can operate at different frequencies and thereby cater different constraint. Existing HDLs, however, implicitly impose clocking by introducing derived registers into the design, on top of registers that are part of the design's functionality (e.g., a state register in a state machine, register-file in a processor).

DFiant decouples the design functionality from its target constraints by removing derived register from the design itself and using the compiler to apply them properly based on functional requirements and design constraints. Specifically, note two major types of derived registers:

**Pipelining registers**

DFiant auto-pipelines the design by inserting registers to split long combinational paths. The amount of pipelining is determined by designer-specified constraints, such as the maximum path cycle latency, or the maximum propagation delay between registers.

**Synchronizers**

Sampling clock domain crossing (CDC) or asynchronous signals is exposed to metastability. Synchronizers, often composed of registers, are used to mitigate its effect and bring the design to the proper reliability. Since we aspire for a clockless design frontend, we want the synchronizers to be implicit. Currently, DFiant only supports a single clock backend, and does not require synchronizers. Further research may explore other backend options.
DFiant is a Scala library, hence it inherently supports type safe and rich language constructs. DFiant brings type driven development concepts to hardware design, by creating an extensible dataflow class hierarchy, with the trait DFAny at its head (similar concept to Scala’s Unified Types hierarchy). DFAny contains all properties that are common to every dataflow variable (e.g., .width represents the number of bits contained by the variable). Figure 4.6 illustrates a simplified inheritance diagram of DFiant’s dataflow types.

DFiant expands traditional structural composition capabilities by utilizing Scala’s object oriented features such as inheritance and polymorphism, as well as finite loops and recursive composition. The hierarchical compositions provide the scope and dependencies for the dataflow variables. The hierarchy itself is transparent to the dataflow graph, as if the entire design is flattened, inlined, and unrolled. Therefore, hierarchies in DFiant are synthesizable, highly reusable, and do not affect the design performance (may affect compilation time). Different composition examples are available in Table 4.1.

Interim Summary
DFiant is a dataflow HDL with advantageous semantics compared to modern RTLs and C++-based HLS tools, such as VHDL and Vivado HLS, respectively. DFiant provides a seamless concurrent programming approach, facilitates a versatile compositional and hierarchical expressiveness, and combines modern programming constructs such as object-oriented programming, type-safety, and polymorphism.

Maxeler MaxCompiler
Maxeler Technologies pioneers a dataflow-oriented programming model for its dataflow supercomputing systems that combine high-end servers with FPGA-based Dataflow Engines (DFEs). This dataflow programming model is realised by MaxCompiler, a comprehensive development, debug and simulation environment. More details on the programming model are provided in section 4.4.8 and a detailed description of dataflow systems and DFEs is given in section 6.5.

Maxeler dataflow systems are a combination of CPUs and DFEs. The basic logical architecture of such a system is illustrated in figure 4.7. A CPU host application computes the control-oriented parts of the application and offloads large-scale computations to the DFE. The DFE contains one or multiple data-flow kernels.
Table 4.1. DFiant Hierarchy Example: Inheritance, Polymorphism, Recursive Composition, and Inlined View

<table>
<thead>
<tr>
<th>Description</th>
<th>DFiant Code</th>
<th>Functional Drawing</th>
</tr>
</thead>
</table>
| Abstract base class, Box (defines only an interface)                       | ```scala
  type DFB = DFUInt[32] // Type alias
  // T=Top, B=Bottom
  abstract class Box(iT: DFB, iB: DFB) {
    val oT: DFB
    val oB: DFB
  }
```                                                                                           | ![Functional Drawing](Box.png) |
| Concrete Box implementation examples                                        | ```scala
  case class BoxY(iT: DFB, iB: DFB)
  extends Box(iT, iB) {
    val (oT, oB) = (iT * iT, iT + iB)
  }
  case class BoxE(iT: DFB, iB: DFB)
  extends Box(iT, iB) {
    val (oT, oB) = (iT + iB, iB)
  }
```                                                                                           | ![Functional Drawing](BoxY.png) |
| Box123, an abstract polymorphic composition of three Box instances          | ```scala
  abstract class Box123(iT: DFB, iB: DFB)
  extends Box(iT, iB){
    def b1Bld(iT: DFB, iB: DFB) : Box
    def b3Bld(iT: DFB, iB: DFB) : Box
    val b1 = b1Bld(iT, iB)
    val b2 = BoxE(b1.oB, b1.oT)
    val b3 = b3Bld(b2.oB, b2.oT)
    val (oT, oB) = (b3.oT, b3.oB)
  }
```                                                                                           | ![Functional Drawing](Box123.png) |
| BoxYEE, a concrete polymorphic composition of three Box instances + An inlined view of BoxYEE | ```scala
  case class BoxYEE(iT: DFB, iB: DFB)
  extends Box123(iT, iB) {
    def b1Bld(iT: DFB, iB: DFB) = BoxY(iT, iB)
    def b3Bld(iT: DFB, iB: DFB) = BoxE(iT, iB)
  }
```                                                                                           | ![Functional Drawing](BoxYEE.png) |
| Finite recursive composition example                                         | ```scala
  case class BoxBox(N: Int)(iT: DFB, iB: DFB)
  extends Box(iT, iB) {
    val b = BoxY(iT, iB)
    val bb : Box = if (N > 0)
      BoxBox(N - 1)(b.oT, b.oB)
    else
      b
    val (oT, oB) = (bb.oT, bb.oB)
  }
```                                                                                           | ![Functional Drawing](BoxBox.png) |

that perform the accelerated arithmetic and logical computations in a highly parallel and deeply pipelined fashion. Each DFE also contains a manager that is responsible for the connections between kernels, DFE memory, and the various...
interconnects such as PCIe, Infiniband and MaxRing. The CPU is also responsible for setting up and controlling the computation on the DFE. This is facilitated via the Simple Live CPU (SLiC) API and the MaxelerOS run time.

![Logical architecture of a data-flow computing system with one CPU and one DFE.](image)

Developing an application for a DFE-based system therefore includes three parts:

1. A CPU application written in a conventional programming language such as C/C++, Matlab, Python, FORTRAN or R;
2. One or multiple data-flow kernels written in MaxJ;
3. A manager configuration, also written in extended MaxJ.

MaxJ is Java-based meta-language that describes dataflow based on Maxeler dataflow extensions. It is important to note the MaxJ will not execute a Java program on the DFE; instead, it is used to describe a dataflow model that will be compiled into a configuration bitstream for the DFE. This compilation process is performed by MaxCompiler as illustrated in figure 4.8. The design typically starts with an existing CPU application where a performance-critical part needs to be accelerated. This part of the application will be targeted on a DFE. Designing a DFE application involves describing one or multiple kernels and a manager in MaxJ. A more detailed description of this programming model is given in section 4.4.8.

Once the dataflow kernels and the manager configuration have been described in MaxJ, MaxCompiler will generate a so-called max-file which is a binary configuration file for the DFE. The compilation process involves automatic scheduling and optimisation steps on the dataflow graph-level before invoking the FPGA-vendors backend tools for technology mapping and bitstream generation. The generated max-file can be linked to the CPU application. It should be noted that a simulation version for fast design simulation and verification can also be produced when the compiler is used in simulation mode. This simulation model will
be significantly faster than typical logic-level simulation for FPGAs. On the CPU side the host application can invoke the max-file via the SLiC API; MaxCompiler will generate the necessary function prototypes and header files automatically. The CPU code is then compiled as usual and is linked with to the max-file to create the application executable. At run-time, MaxelerOS will manage the interactions between host application and DFE.

MaxCompiler runs CentOS 6/7 or Red Hat Enterprise Linux 6/7 and requires GCC, Oracle Java 1.8, Apache Ant as well as the FPGA vendor backend tools (i.e., Intel Quartus when targeting MAX4-generation DFEs and Xilinx Vivado when targeting MAX5). Running a Maxeler dataflow application requires CentOS or Red Hat Enterprise Linux and MaxelerOS.

MaxCompiler can also be used to develop applications for Amazon EC2 F1 instances. This Amazon instance type uses FPGAs and is directly compatible to the latest generation MAX5 DFEs. Development for Amazon EC2 F1 is a feature in MaxCompiler as of version 2018.1. MaxCompiler itself can also be run directly on the Amazon cloud, removing the need for a local installation of the compiler.

### 4.4. Programming Model

In this section, we introduce all the components of the LEGaTO programming model, and the supporting tools.

#### 4.4.1. Basic OmpSs

The OmpSs programming model is based on tasks. The main directive is the `task` directive. Around the task directive, we describe the possibilities that the runtime has for obtaining parallelism.

The `task` annotation and its related clauses are shown in Figure 4.9. The clauses are used as follows:
Figure 4.9. Task directive and clauses

- **in**, **out**, **inout** clauses are used to specify the input, output, and input/output data read and/or written by the task. The runtime system uses this information to compute the dependences between the tasks.

- **concurrent** and **commutative** allow the specification of variants of **inout** dependences. Concurrent means that data is accessed with an explicit synchronization inside the task, thus allowing the runtime to execute them in parallel. Commutative indicates that the tasks can be executed in any order, even that still one at a time.

- **priority(P)** is used to specify the importance of the task. It is a hint to the Nanos scheduler, that can be used to favor the execution of higher priority tasks, depending on the scheduling policy.

- **label(taskname)** provides a name for the task, used for the instrumentation facilities to show the name in the visualization of the traces.

- **shared**, **private**, and **firstprivate** specify the data sharing between the code creating the task and the task itself. They have the same meaning as the equivalent clauses in OpenMP.

- **untied** is used to allow tasks to migrate from processor to processor when reaching a task scheduling point, (e.g., a taskwait).

- **final(expression)** indicates when the task on which it is associated will not create children tasks.

- **if(expression)** allows to indicate that the task is deferred or not. If the expression evaluates to **True**, then the task will be a regular deferred task. If the expression evaluates to **False**, then the task will be created and executed immediately.

Tasks are the basic unit of parallelism. Nevertheless, by default, they will only run on the cores of the host platform. We will provide compilation options in Mercurium to target different accelerators for all tasks available in each compilation unit.

4.4.2. Annotations for Mapping

Mapping the execution of tasks to various types of resources is done using the **target** directive. The syntax of this directive is presented in Figure 4.10. The clauses are used as follows:

- **device(devlist)** limits the code generation to the devices listed in the clause.
• `copy_deps` is used to include all data referenced in data dependence clauses to the list of data that needs to be copied to the devices. Alternatively, the `no_copy_deps` clause makes the compiler not to copy any of the data referenced by the data dependence clauses. Observe that this data can still be indicated to be copied by the `copy_*` clauses (see below).

• `copy_in`, `copy_out`, `copy_inout` clauses are used to list additional data regions that should be kept consistent with the data in the accelerator.

• `implements(function)` indicates that the associated task implements the same functionality that the named function. This feature allows the runtime system to invoke one of the functions in the respective device, as decided by the programmer, the scheduling policy, etc.

• `ndrange(...)` is used to indicate the kernel configuration for devices that require it, like CUDA and OpenCL accelerators.

• `shmем(M)` indicates the amount of shared memory that the kernel invocation requires in the target device. It is also related with invocations of CUDA and OpenCL kernels.

### 4.4.3. Annotations for Fault-tolerance

In LEGaTO, we will extend the OmpSs task pragma with fault-tolerance annotations. These annotations should be simple and intuitive for the programmer, and easy to implement in the runtime. A naive approach could bake specific fault-tolerance implementations in the pragma extension. An example of this approach is to annotate a task to run in duplicate (for fault detection) or triplicate (for fault correction) manner. However, this approach is not straightforward for general programmers to adopt and these baked-in annotations does not allow any flexibility to the runtime to choose the best fault-tolerance scheme dynamically. Another slightly better option would be to allow a replicate annotation to be used as part of the task definition. However, this is also not the most flexible option to be provided to the runtime; although it does not mention the specific policy (task duplication or triplication) and it this more flexible, it still imposes a somewhat rigid fault-tolerance implementation (task replication) on the runtime. Instead of these approaches, we propose a protect annotation as part of the task; this annotation would signal to the runtime that the task needs to be protected using some runtime-directed fault-tolerance mechanism. This mechanism could be completely selected by the runtime, and would include task replication as well as checkpointing or fingerprinting fault tolerance implementations to be selected dynamically at runtime. The absence of a protect clause from a task description would imply that this task instance does not need to be
protected. In some cases, the protection attribute is not statically linked to the task but depends on the dynamic factors such as a program phase. To allow the programmer to express this program phase dependent protection, we allow the protect clause to be applied as part of the evaluation of a condition at runtime, if the condition evaluation is positive then the dynamic task instance is protected and not otherwise.

4.4.4. Annotations for Security

Application tasks may need the security feature, guaranteed by the confidentiality and integrity of the runtime system (see subsection 4.4.13). Such integrity will be implemented using Intel SGX enclaves, or a similar runtime-based mechanism provided by the underlying architecture. Code generation for enclaves and multithreaded code is currently done using HAFT, an extension to the LLVM compiler, that modifies the task code to ensure that the runtime and task integrity is kept.

We will incorporate a new clause onto the task directive (trusted), that will make the compiler to offload the trusted tasks code onto separate files that will be compiled with the LLVM compiler supporting HAFT. This integration will provide task-based parallel applications with the security needed to avoid security threats.

4.4.5. Annotations for XiTAO

XiTAO is a novel runtime system that executes mixed-mode parallel applications targeting high parallelism, low scheduling overheads and minimal inter-task interference. In XiTAO each task in the computational DAG is a moldable parallel computation that is built from a tightly connected collection of fine-grained tasks. Hence such XiTAO tasks are called TAO (task assembly object). In the context of LEGaTO, we will add programming model annotations to address (1) resource requirements, (2) explicit identification of task assembly objects, and (3) application topology annotations.

A TAO is a partition of work that can dynamically obtain and exploit variable number of resources. Determining the optimal number of resources is a core consideration to be research in LEGaTO. We will consider several ways in which the resources to be provided to a TAO are determined:

- **static** The simplest option is for the programmer to directly specify the number of cores to be used via an annotation. We will leverage the OmpSs pragma system to provide this information. Assume TAO_F is the name of a TAO obtaining parameters A and B, then the following code snippet can be used to specify the number of cores:

  ```
  #pragma legato xitao static(8)
  TAO_F(A,B);
  ```

- **auto** The user may also request the system to determine dynamically the optimum number of resources. In this case the system may use a performance monitoring scheme to come up with a good solution. The following code snippet shows how to make use of this feature.
A second annotation concerns the identification of a task assembly. In the current XiTAO implementation, TAOs can only be specified as dynamic C++ objects. The specification of such objects is quite cumbersome to the programmer. This encourages an alternative approach in LEGaTO. One such approach is to identify a TAO by specifying a group of OpenMP tasks that execute statically as a task assembly. The OpenMP task_group directive will be used to specify a TAO. The following code is an example of how to generate and execute a TAO from a collection of OpenMP 4.x tasks with dependencies.

```
#pragma legato xitao auto
#pragma legato xitao task_group
{
    #pragma omp task out(A[N])
    F1();

    #pragma omp task in(A[N]) out(B[N])
    F2();

    #pragma omp task in(A[N]) out(C[N])
    F3();

    #pragma omp task in(B[N],C[N]) out(D[N])
    F4();
}
```

It will be the task of the LEGaTO toolchain compiler to take these tasks and generate a proper TAO object and correctly pass the parameters at runtime (both input parameters as well as resource specifier).

The third annotation, the topology information, can be used by the LEGaTO runtime to perform energy efficient schedules, taking into account parallelism, overheads and communication costs. The topology annotations describe application-level locality in a device-independent form. Although we plan to research this mechanism in the context of XiTAO, we will consider its applicability to other system components at a later stage.

The basic idea of topology annotations is to decouple the specification of tasks and their dependencies from the information on inter-task communication. While OpenMP 4.x-style task dependencies hint towards data communication, detailed information on actual communication requirements is not available. Tasks are not required to access all of their input data, and, when they do, it is not known how many times a certain element is accessed. Hence, in the context of XiTAO an alternative method to specify information on inter-task communication is being researched. The fundamental idea is to map an application DAG (called TAO-DAG in the context of XiTAO) on top of a virtual topology so that each node in the DAG is given an address in the virtual topology. In this virtual topology, task distance is supposed to represent sparsity in data communication. The further away two tasks are, the less amount of data they communicate. This information can then be used by the XiTAO scheduler to determine on which
node a certain task should be executed. Particularly in the context of heterogeneous computers, this requires taking care of an important trade-off: should we schedule a certain task to a powerful, yet more distant device, or should a task be scheduled to a slower, but closer device. To answer this question it is necessary to develop a cost model that takes into account distance, energy and performance.

In the current XiTAO prototype, only 1-D Cartesian topologies can be specified. In the LEGaTO project we will extend this model to more complex topologies and develop software-level annotations so that the programmer can choose the topology of her choice and specify the particular address of each task. Figure 4.11 shows how the concept of virtual topologies is implemented in the current XiTAO prototype using a stencil-based application (heat) as an example. The horizontal line labeled virtual topology specifies the location of each task in each iteration.

The following code snippet extends the previous code with a user-level annotation indicating that the TAO should execute at coordinate (0.25, 0.75). This code assumes that the application is mapped on a virtual 2D topology.

```
#pragma legato xitao auto
#pragma legato xitao task_group at (0.25, 0.75)
{
    ...
}
```

### 4.4.6. Annotations for Reconfiguration

We will provide program annotations with hints to the runtime system about when it is appropriate to change the functionality of the runtime system in various ways:

- **Change of the scheduling policy.** Different sections of the code and the parallelism can have different requirements regarding the scheduling policy being used. We will provide a runtime interface to change the scheduling policy during runtime, to better fit the structure of the parallelism.

- **Reconfiguration of the FPGAs.** Different sections of the applications may require different accelerated functionalities. We will provide a runtime interface to suggest the runtime system to replace the FPGA configuration...
with a new set of algorithms. This feature will require that the FPGA low-
level infrastructure supports either online partial reconfiguration or online
full reconfiguration.

4.4.7. Annotating Design Constraints in DFiant

DFiant decouples the functionality of a design from design target constraints to
produce portable code and improve designer productivity. In order to map a
design to specific frequency, area, and power goals, the designer introduces an-
notations that describe the target constraints. In this section we briefly overview
the annotation syntax and code generation process.

4.4.7.1. DFiant Annotations

Decoupling of functionality from target constraints allows designers to only de-
fine the target constraints at the top-level instantiation of the design (in practice,
any specific module may include specific constraints, but we focus on the simple
top-level example).

DFiant provides a straightforward interface for expressing constraints as part of
its object-oriented design. In DFiant, all modules must inherit the DFDesign
class, which comprises the common traits of a hardware design.

The following code depicts the creation of a new type of modules called MyDesign:

```scala
// create a module type named MyDesign
trait MyDesign extends DFDesign {
  // here lies the logic for modules of type MyDesign
}
```

When instantiating an object of type MyDesign, DFiant enables the designers
to introduce design constraints. Example constraints include frequency, timing,
power, resource usage.

The following example demonstrates how a designer can define constraints for
an object of type MyDesign:

```scala
// establish the target part number by importing its definitions
import Xilinx.FPGAs.XC7VX485T-2FFG1761C._

// instantiate a top-level MyDesign object
val topMyDesign = new MyDesign{}

// introduce design constraints
val topMyDesign.addConstraint(new PerformanceConstraint.MinimumFrequencyMHz(100))
val topMyDesign.addConstraint(new PowerConstraint.MaximumPowerWatt(20))
```

The design constraints are practically self-explanatory. The designer must first
import the DFiant library for the target device. This library includes the pow-
er/performance model for the specific target device, based on which the de-
sign will be created. The designer then adds constraints requiring the DFiant
toolchain to produce code that can operate at a frequency of at least 100MHz
on the target device and consume at most 20 Watts. If the combined target can-
not be met the compilation will fail with an informative error message.
4.4.7.2. Compiling with Design Constraints

The DFiant compiler parses the design and represents it internally as a dataflow graph. The dataflow graph incorporates the design functionality, namely the operations and their dependencies. It is up to the DFiant compiler to introduce derived registers that bound timing paths while abiding with any additional constraints. This is the key strength of DFiant. In existing HDLs, the designer must manually augment the design to introduce derived registers and abide by the design constraints.

The compilation process naturally includes conflicting constraints. For example, increasing frequency (i.e., reducing cycle time) requires pipelining the design by introducing pipelining registers. However, the additional registers also increase resource usage on an FPGA and thereby overall power consumption. In addition, the frequency itself is a key factor in the final power consumption of the design.

To resolve the multitude of intertwined constraints, the DFiant compiler uses common practices from operations research and expresses constraints resolution as optimization problem. Specifically, the compiler formalizes register placement as a constraint programming problem and uses the OscaR solver. The solver then generates an optimal register placement that abides by all constraints (or fails with an informative error message).

4.4.8. Annotations for Maxeler

As mentioned in section 4.3.6, MaxJ is a meta-language that uses Java syntax but differs from regular Java programming in that it described dataflow instead of describing computations by changing state. More specifically, MaxJ is used to describe a fixed compute structure that can perform computations by simply streaming through data. It does not describe a sequence of instructions that will be executed on a traditional processor.

Describing the full MaxJ programming model is beyond the scope of this document but here we provide a simple example that illustrates how a simple loop computation can be transformed into a dataflow description using MaxJ. Let us assume we want to calculate \( y = x^2 + 3x + 17 \) over a data set. Even though there is nothing inherently sequential in this computation, a conventional C program would require a for loop. This is illustrated in figure 4.12.

```c
for (i = 0; i < numDataElements; i++) {
    float x = input[i];
    float y = x * x + 3 * x + 17;
    output[i] = y;
}
```

*Figure 4.12. C code of a simple computation inside a loop.*

A MaxJ implementation for the same computation is shown in figure 4.13. It involves extending the kernel class that is part of the Maxeler Java extensions. The loop iteration in the C-code is replaced by streaming inputs and outputs; i.e., the kernel will not iterate over data but data will streamed through it instead. The computation itself is expressed in a very similar way as in the original C-code. The DFEVar object is used to handle run-time data. Since MaxJ describes a
dataflow pipeline rather than a procedure, we have to distinguish between run-time values and compile-time values. Regular Java variables and constructs will be evaluated at compile time to control the generation of the dataflow graph.

```java
class SimpleCalc extends Kernel {
    SimpleCalc() {
        DFEVar x = io.input("x", dfeFloat(8,24));
        DFEVar y = x * x + 3 * x + 17;
        io.output("y", y, dfeFloat(8,24));
    }
}
```

*Figure 4.13. An MaxJ description that generates the data-flow implementation shown in figure 4.14.*

The resulting dataflow implementation of this MaxJ kernel is shown in figure 4.14. It is a compute pipeline that contains two multipliers and two adders and that can process data by streaming it through the pipeline. This is a very simple example and practical dataflow kernels can contain many thousands of operators.

![Figure 4.14. A data-flow implementation for the computation inside the loop body.](image)

Practical development of MaxJ dataflow kernels involves focusing on high degree of pipelining and parallelism without worrying about scheduling or synchronisation. It is also possible to use custom arithmetic such as fixed point or non-standard integer or floating point formats. The scheduling of operations inside the kernel will be performed automatically by the compiler. The manager code describes how kernels are connected to memory and other IO interfaces, and the necessary synchronisation logic will also be generated by the compiler.

Figure 4.15 illustrates the full development process of a dataflow application. On the right side, a MaxJ compute kernel is shown. In the centre, a MaxJ manager
describes the connections between the kernel and the host CPU code, i.e. all communication will be facilitated via the PCIe interface. On the left, the host application uses a call the SLiC API to invoke the DFE kernel in the computation. The function prototype is automatically generated by MaxCompiler based on the kernel interfaces specified in the manager configuration.

![Diagram of system connections]

**Figure 4.15. Interaction between host code, manager and kernel in a dataflow application.**

### 4.4.9. Integrated Development Environment

The LEGaTO project will incorporate the developed energy-efficient software tools/environments, including OmpSs and aspects of XiTao into a standard Integrated Development Environment (IDE) with a view to increase programmer productivity as well as the performance and energy-efficiency of applications. A widely used environment, such as Eclipse will be used for IDE integration. The project will take a staggered approach to iteratively add more features into the IDE, through an OmpSs plugin. Multiple internal test releases will be contemplated, each with additional features. The plan is to start with relatively straightforward work items such as syntax highlighting for the OmpSs directives, and to include more advanced features such as integrating OmpSs profiling tools such as Paraver and enabling debugging sessions, into the IDE to help with performance analysis and debugging of the task-based application in the subsequent releases.
4.4.10. Efficient Support for Irregular Data Structures

Many applications work on irregular data structures, such as graphs, trees, and meshes. These applications are difficult to parallelize using tasks on distributed memory through OmpSs@Clusters (Section 4.3.2). In this task, BSC will investigate programming model and runtime support for efficient and scalable support for such irregular data structures. This work is planned to start before M12.

An important challenge is the typically small amount of processing per vertex, which means that with a single task per vertex, the overhead of creating a task, managing task dependencies, transferring data and scheduling the task for execution would be too high. In addition, the irregular non-contiguous data structures mean that the overhead of dependency checking and data transfers would still be too high, even if multiple vertices are aggregated into one task. The initial approach will be to allocate vertices in memory pools and introduce run-time system support to manage and batch up communication. Performance, energy and programmer productivity will be compared against an implementation using ArgoDSM Distributed Shared Memory.

4.4.11. Directive-based Checkpointing

Most applications use a vast number of temporary data that can be regenerated from other variables (i.e., state variables). For some applications, the amount of temporary data can even grow exponentially and then be freed at a particular point in time. Thus, checkpointing the entire memory used by the application at random points during the execution can lead to storing large amounts of unnecessary data. This leads to huge energy waste, as transferring a large dataset to stable storage is one of the most power-hungry tasks. In order to improve the checkpointing efficiency for large scale applications, the front-end will offer an API to explicitly declare the variables required to recover from failures and the best points in the execution to perform a checkpoint. The API needs to be simple and intuitive to maximize productivity.

Protected variables

The directives should allow users to declare variables that need to be protected during the checkpoint. In addition to registering a variable for checkpointing, it is important for the front-end to offer the possibility of updating the information related (e.g., pointer, size) to the variable to be checkpointed. This is important because many applications deallocate and reallocate variables during the execution. Other applications have variables that change in size during the execution because some parts of the dataset move across multiple processes as the execution progresses. Proper handling of these scenarios is mandatory to avoid segmentation faults at restart time. All this will be handled transparently to the user.

Type and position of checkpoints

Users should be allowed to checkpoint at any time during the execution. Some applications require to checkpoint at specific points in the execution. For instance, climate simulations often perform checkpoints at the end of a simulated month, and the wall-clock time is almost irrelevant. Also, users should be capable of declaring manually the level of resilience desired for each particu-
lar checkpoint. Including multiple resilience levels helps to increase efficiency while coping with multiple types of errors.

The following directives will be defined:

- **Initialize**: Initialize the checkpoint runtime with a set of configuration parameters derived from the system architecture and the needs of the application.

- **Protect**: It declares a variable to protect within the checkpoint. It gives the data type and size to the runtime and updates it if it has changed.

- **Checkpoint**: It declares a point in the execution where the checkpoint has to be taken. This directive accepts an argument to define the level of reliability necessary for the checkpoint in progress.

- **Finalize**: Finalizes the checkpoint runtime by releasing the used memory and removing temporary data.

### 4.4.12. Static Kernel Identification

There are several places in which identifying static super-tasks is needed in the context of LEGaTO. Super-tasks refer to tasks that contain multiple subtasks. Super-tasks are needed to effectively use FPGAs, where multiple tasks need to be arranged as a single kernel in order to effectively use the FPGA space and reduce communication between CPU and FPGA. In the context of XiTAO, super-tasks are needed to implement elasticity, the main scheduling method for interference-freedom. In XiTAO such super-tasks are equivalent to Task Assembly Objects (TAO). They are conceptually similar to non-preemptive nested parallel runtimes.

Generating such super-tasks from the programmer perspective is quite complex. One of the difficulties is to know how many tasks need to be aggregated into a super-task. Another difficulty is the actual development of the super-task, which may require a nested scheduler (for example, in XiTAO). Hence, we will develop technologies to aid in the generation of super-tasks. These technologies are collectively called *static kernel identification*.

We will consider two methods, a loosely coupled method based on profiling and a strongly coupled method based on compiler analysis. The loosely coupled method will run a single-threaded version of the LEGaTO program and record an execution trace of the execution. An independent tool will then scan the trace and identify sets of tasks that have strong communication and consistently execute as a single unit. The tool will then output a suggestion to generate a single static kernel from these tasks.

The second method attempts to automatize this procedure. Using compiler technology we will identify compute-intensive sets of tasks that consistently execute as a single unit. The compiler will then reorganize the code and instruct the FPGA HLS toolchain to generate a single kernel from these tasks, or rearrange the tasks into a TAO object, following the XiTAO API, so that the XiTAO runtime can execute the kernel as a single object, implementing the elastic scheduler described in Section 5.5.10.
4.4.13. Trusted Tasks

To ensure the confidentiality and integrity of the application runtime in LEGaTO, we design and implement a security mechanism based on hardware-assisted memory protection using Intel SGX technology. One of our design goals is generality which means that our proposed mechanism can also support alternative hardware-based solutions (e.g., AMD SEV [47]). In LEGaTO, an application may consist of multiple tasks. The tasks protected by the proposed security mechanism are called trusted tasks. Next, we give a background about Intel SGX, thereafter we describe in detail our design in the next section 4.4.14.

Intel SGX: is an ISA extension which is a set of CPU special instructions for Trusted Execution Environments (TEE). These instructions enable applications to create enclaves – protected areas in the applications address space to provide strong confidentiality and integrity guarantees against adversaries with privileged root accesses (see Figure 4.16). Intel SGX enables trusted computing by isolating the environment of each enclave from the untrusted applications outside the enclave. In addition, by offering the remote attestation mechanism, Intel SGX allows a remote party to attest the application executing inside an enclave [24].

The enclave memory is acquired from Enclave Page Cache (EPC) — a dedicated memory region protected by an on-chip Memory Encryption Engine (MEE). The MEE transparently encrypts cache lines with writes and decrypts and verifies cache lines with reads. The EPC cannot be directly accessed by non-enclave applications including operating systems. To support multiple enclaves on a system, the EPC is partitioned into 4KB pages which can be assigned to various enclaves. Currently, the size of EPC is limited from 64MB to 128MB in which only 94MB can be used for user applications and the rest is used to store SGX metadata. Fortunately, SGX supports a secure paging mechanism to an unprotected memory region even though the paging mechanism incurs significant overheads depending on the memory access pattern (from $2 \times$ to $2000 \times$) [52].

The EPC is managed as the rest of the physical memory by an operating system (i.e., a hypervisor or an OS kernel). The operating system makes use of SGX instructions to allocate and free EPC pages for enclaves. In addition, the operating system is supposed to expose the enclave services (creating and managing) to applications. Since the operating system cannot be trusted, the SGX hardware (processors) verifies the correctness of EPC pages allocations and denies to perform any operations that would violate the security guarantees. For example, the SGX hardware will not allow the operating system to allocate the same EPC page for different enclaves.

To perform the verification, SGX makes use of the Enclave Page Cache Map (EPCM) to keep track of the information about EPC pages allocation operations of the operating system. EPCM is a table in which each row is used for each EPC page. EPCM keeps track the ownership of each EPC page.

Enclave Life Cycle: An enclave is created when the operating system calls the ECREATE instruction. This initializes an SGX enclave control structure (SECS) in the EPC (SECS is used to store the metadata associated with the enclave). The ECREATE marks the SECS as uninitialized. In this stage, the instruction EADD is
used to load the initial code and data into the enclave. The instruction EEXTEND is used to measure the content of the new added EPC page and update the cryptographic hash of the enclave (MRENCLAVE). Thereafter, the instruction EINIT is called to initialize the enclave. This instruction requires an INIT token which is created by a privileged enclave provided by Intel (Lunch Enclave) \[24\]. When having the INIT token, the instruction EINIT marks the SECS as initialized. After this point, no more pages can be added to the enclave, i.e., the instruction EADD cannot be executed in the enclave anymore. MRENCLAVE can now be reported to other enclaves for local attestation via the EREPORT instruction. When all computations within the enclave are done, the instruction EREMOVE is used to deallocate EPC pages allocated for the enclave. The instruction EREMOVE marks EPC pages as available by setting the VALID field in the EPCM table. The enclave is completely tear-downed when the EPC page keeping its SPECS is freed.

**Attestation:** After initializing an enclave, it is possible to create an attestation by using the EREPORT instruction. The report can be used by the enclave to prove its identity to other enclaves running on the same machine. This process is called *Local Attestation*. This process makes use of a symmetric key called *report key* which is only shared among the other enclaves and the SGX implementation. The report binds the identity of the enclave using a Message Authentication Code (MAC) with the symmetric key (see details in \[24\]). Note that the Local Attestation mechanism can only be used to verify enclaves running on the same platform. To allow a remote party attest the enclave, SGX utilizes an architecture Platform Service Enclave (PSE) namely *Quoting Enclave* to sign the report using Intel Enhanced Privacy ID (EPID) group signature method. The process is called Remote Attestation, please refer \[24\] for the details.

### 4.4.14. Compiler-based Protection Against Security Threats

To protect the integrity and confidentiality of applications and make them more resilient against security attacks, we make use of trusted execution environments (TEEs), more specifically, Intel SGX enclaves (so far, we not yet support other hardware-based solutions, e.g., AMD’s SEV). The proposed approach al-
allows us to reduce the size of the trusted computing base by at least an order of magnitude. The security mechanism can protect user-level applications against potential attacks from the privileged system software. However, note that user-level applications can not “just run” inside Intel SGX enclaves since Intel SGX imposes a number of restrictions on applications running inside enclaves, e.g., not allowing system calls directly called inside of enclaves, they need to be sanitized by a shielding process in the enclave before being used. However, system calls are the standard way for any user-space application to request service from the privileged operating system kernel. To transparently support this, we propose a compiler-based protection mechanism which prepares and builds the applications for execution inside enclaves. With our approach, no manual source-code changes are required to run applications inside enclaves. Currently, we support applications written in C/C++, Rust, Go, Python, Fortran, and Java).

To protect the syscall interface against security attacks, we need to design wrappers to copy all memory-based system call arguments between the enclave and non-enclave memory instead of having enclave code operate on pointers to non-enclave memory. In our mechanism, we support system call shields similar to Haven [10], SCONE [4], and Panoply [81] to protect the integrity and confidentiality of applications running inside enclaves. Typically, in our approach, the process of compiling an application to run inside of an enclave includes the following steps. First, the source code of the application is compiled as position independent code since the enclave hash does not depend on the absolute position of the enclave. Then, the object files are linked with specially crafted language bindings and starting routines into a library. Meantime, the language bindings have been compiled with a standard C library (we use Intel’s SGX driver for this) adapted to run legacy applications inside SGX enclaves. Thereafter, the generated binary is wrapped in a starter program that puts the binary in the enclave, measures it and then invokes the code. We implement the proposed mechanism in a toolchain by extending our previous work called SCONE [4]. Basically, SCONE is a shielded execution framework that enables unmodified legacy applications to take advantage of the security guarantees offered by Intel SGX. In SCONE platform, the source code of an application is recompiled against a modified standard C library (SCONE libc) to facilitate the execution of system calls. The address space of the application stays within an enclave, and the application only can access the untrusted memory via the system call interface.

We make use of the GNU Compiler Collection (GCC) to compile application written in C/C++, Fortran and Go. More specifically, we apply our toolchain into GCC to change the compiling process such that we can build position independent, statically linked code, and eventually linked with the starter program. This natively supports C/C++ applications. For Fortran and Go applications, we need to change the compilation of their respective runtime libraries to produce position independent code. For Rust application, since it uses its own toolchain, we compile the Rust standard libraries to use our enclave-enabled C library and instruct the cargo build system to build a static library instead of an executable binary. For Python applications, since Python is an interpreted language, we compile the CPython/PyPy interpreter with our C toolchain to run Python application inside an enclave. Finally, for Java applications, we recompile the Java virtual machine (JVM) with our C toolchain, to run Java applications inside enclaves. Note that,
since Python and Java codes are executed using the interpreter and Virtual machine, respectively, we protect their integrity and confidentiality by designing a file system shield.

While Intel SGX protects against accesses to the enclave memory, it does not prevent an application access and write its secrets and other sensitive data to non-enclave memory. To handle this problem, we implement runtime checks to disallow functions accessing non-enclave memory, and we activate this functionality by default in our toolchain. We use the MPX CPU extension for these runtime checks which incurs less than 10% overhead. Restricting the accesses to non-enclave memory means that even if an attacker is able to hijack the enclave’s control flow, he/she would need to use a function permitted to write data inside the enclave into non-enclave memory. To further improve the security of applications, we use additional techniques such as bounds checking inside enclaves to decrease the likelihood of control flow hijacks. The bounds checking technique provides stronger guarantees than region-read-write-integrity as well as lower performance overheads (only 18% [52] compared to 24% [78]).

Hitherto, there is no report regarding successful direct attacks on Intel SGX. However, there are several reports about successful side channel attacks (SCAs) on Intel SGX. In details, the traditional cache timing and page table SCAs expose the page-level memory access patterns [12, 14, 79, 35, 90, 40, 89]. In addition, recently, speculative attacks [49, 17] that use the side channels as a way of retrieving information.

Although Intel explicitly excludes SCAs from the SGX threat model, SCAs effectively break the SGX confidentiality guarantees and prevent the SGX adoption in real-world applications. More seriously, an adversary with the privileged right can deploy more powerful SCAs compared to the unprivileged one in many canonical variants of the attacks. For example, a malicious OS is able to significantly reduce the noise levels in cache timing attacks by using single-stepping [40] or via slowing down the victim. To solve this problem, we propose a mechanism called Varys [62] to protect unmodified applications running in SGX enclaves from cache timing and page table side-channel attacks. Varys takes a pragmatic approach of strict reservation of physical cores to security-sensitive threads, thereby preventing the attacker from accessing shared CPU resources during enclave execution. The key challenge that we are addressing is that of maintaining the core reservation in the presence of an untrusted OS. Varys fully protects against all L1/L2 cache timing attacks and significantly raises the bar for page table SCAs—all with only 15% overhead on average [62]. Currently, we implement Varys as an LLVM compiler pass that inserts periodic calls to a runtime library. We use our C toolchain to provide us with asynchronous system calls as well as in-enclave threading such that we minimize the need for an application to exit the enclave. In the next step, we need to integrate Varys into our toolchain (not using LLVM) to enhance the security guarantees for applications running inside enclaves.

Last but not least, another issue we want to tackle in the LEGaTO project is to securely deploy and bootstrap applications with Intel SGX. Deployment and bootstrapping applications to run inside enclaves is not trivial, in fact, challenging. Securely transferring secrets such as certificates, encryption keys and passwords...
to start applications inside enclaves is complicated since the fact that they need to be protected on the network as well as securely moved in the enclaves. To solve this problem, we design a configuration and attestation service (CAS) that releases security sensitive data only to applications that have authenticated themselves successfully against it.

### 4.4.15. Compiler-based Protection Against Software Faults

Unpredictable hardware/software faults can cause arbitrary state corruptions during computation, thus lead to application crashes. In the LEGaTO project, we introduce a compiler-based protection mechanism against software faults. Our goal is to achieve a $5 \times$ decrease in Mean Time to Failure. At the beginning phase of the project, we focus on handling transient faults which occur irregularly and randomly for a short of time. These faults affect various part in the application stage, i.e., they may activate errors during the executions. There is a wide range of causes for transient faults. For example, transient faults in CPUs may occur due to overheating, hardware/software incompatibility, power supply faults, manufacturing problems, etc. A transient fault can lead to process state corruption, data loss, and even outage of the entire application. Therefore, applications running on top of the LEGaTO platform need to be adapted to tolerate these transient faults. In the literature, there are plenty of mechanisms already proposed to protect applications against transient faults [92, 93, 75, 82, 53]. In general, these approaches first add redundancy at the instruction level, threads, or whole processes; then insert periodic comparisons of redundant copies to detect transient faults. Unfortunately, these approaches target only sequential applications, and they become impractical for multi-threaded applications.

Recently, a few hardening approaches for multi-threaded applications have been proposed [28, 11]. However, these approaches still contain at least one of the following limitations: (1) requiring to modify the source code of applications, (2) requiring operating system support, deterministic multi-threading and/or spare core for redundant executions, (3) relying on application-specific checks leveraging the high-level programming languages such as Apache Pig [64], (4) targeting restrictive programming models, e.g., assuming only event-based applications, and (5) providing only fail-stop semantics without providing recovery from faults.

To overcome the aforementioned limitations, we make use of a Hardware-Assisted Fault Tolerance (HAFT) technique — a compiler-based fault tolerance mechanism using hardware extensions of commodity CPUs to protect unmodified applications against transient hardware/software faults [51]. HAFT has the following advantages compared to the state-of-the-art mechanisms. First, HAFT applies to unmodified applications on the existing operating systems running on commodity hardware. Second, HAFT targets the general shared-memory multi-threaded programming model supporting the full range of synchronization primitives. Moreover, HAFT neither enforces deterministic execution nor requires spare cores, and thereby, it does not limit the available application parallelism, which is crucial for imposing low performance overheads. Finally, HAFT achieves high availability by providing fault detection as well as recovery from faults. The idea behind HAFT is that it utilizes two main techniques: (i) instruction-level redundancy (ILR) for fault detection and (ii) hardware transactional memory (HTM) for fault recovery.
(a) Native

1. z = add x, y
2. z2 = add x2, y2
3. d = cmp neq z, z2
4. br d, crash
5. ret z

(b) ILR

1. z = add x, y
2. z2 = add x2, y2
3. d = cmp neq z, z2
4. br d, crash
5. ret z

(c) HAFT

1. xbegin
2. z = add x, y
3. z2 = add x2, y2
4. d = cmp neq z, z2
5. br d, xabort
6. xend
7. ret z

Figure 4.17. Our compiler-based protection mechanism transforms original code: (a) replicating original instructions with ILR for fault detection (b) and covering the code in transactions with Tx for fault recovery (c). Green lines highlight instructions inserted by our compiler using ILR and Tx.

To achieve fault tolerance, our compiler-based protection mechanism utilizes HAFT to transform an application as follows. First, we replicate the instructions of the application and add periodic integrity checks. In this way, the replicated instructions create a separate data flow existing in parallel to the original one, and the instruction-level parallelism of modern CPUs efficiently schedule both data flows. Next, to provide fault recovery, we cover the whole execution of the application using HTM-based transactions. Whenever a fault is detected by ILR, the transaction is automatically rolled back and re-executed. Note that the HTM implementation we employed is best-effort which also renders HAFT’s recovery guarantees best-effort. However, our experiments show that a clever placement of transactions allows our fault tolerance mechanism to achieve high availability even in the presence of frequent faults (see [51]). Figure 4.17 illustrates an example of our compiler-based protection mechanism transforming a simple code snippet. We apply ILR first by replicating all instructions except control flow ones (Figure 4.17 (b)). To detect faults, we use ILR to insert a check before returning the result. A fault is detected if two copies of data differ then an error is reported by enforcing application termination. To achieve fault recovery, we next apply Tx to cover the code in transactions and substitute crashes by transactions aborts (Figure 4.17 (c)). If a fault is detected at runtime, the current transaction is rolled back and re-executed. Our mechanism attempts to rerun aborted transactions for a certain number of times (three times in our current implementation), after that the code is executed non-transitionally until a new transaction is encountered. If a fault is detected during the non-transactional part of the code, the ILR-based mechanism has no choice but terminate the application. In other words, our compiler-based protection mechanism offers best-effort fault recovery. It falls back to the fail-stop model in rare cases the bounded-number of re-executions (three times) is reached.

Currently, we implemented HAFT as an extension of the LLVM compiler framework to transform unmodified application code. In the next phase, we will integrate HAFT with our security toolchain (see 4.4.14), so that our compiler-based protection mechanism can protect user-level applications not only against soft faults but also security threats, including side-channel attacks, external exploits, and malicious execution environment.
4.5. OpenStack Extensions for a Task-based Programming Model

Some extensions to OpenStack will be required in order to fully support LEGaTO’s task-based programming model. As described below, the fault tolerance techniques developed in the project will benefit from elastic allocations (so that the application can restart using on-the-fly added nodes) and migration among heterogeneous nodes. In addition, the unique ability of the RECS|Box system to support node composition and dynamic (re-)configuration of the communication infrastructure requires the ability to pass the current configuration information to the topology-aware distributed task scheduler (see Section 5.5.9). This is done as described in Section 5.5.1. In addition, the ability for the runtime system to request dynamic reconfiguration using the Redfish API is under consideration.

4.5.1. Logical-level Orchestration of Hardware Resources

Many of the fault tolerance techniques developed in this project allow for the efficient recovery of the application without the need of re-launching an entire job again. In fact, most of the non-failed processes on a failed application can restart on-place using the fault tolerance techniques integrated in the LEGaTO framework. However, these capabilities are not exploited if the scheduler is not capable of dynamically adapting to such scenarios. Therefore, the following features should be supported by the job scheduler.

Elastic Allocations

The hardware resources orchestrator should support the elastic allocation of resources that allow for a failed application to continue its execution in a sub-set of previously allocated nodes and a sub-set of on-the-fly added nodes to replace the failed ones. Given that applications run on hybrid systems, the scheduler must take the heterogeneity into account at the moment of reallocating new resources to replace broken ones.

On-demand Heterogeneous Migration

Heterogeneity is at the core of LEGaTO, hence executions on different architectures is necessary. In particular, some executions might need to be interrupted and migrated to continue executing in a different type of architecture. Such type of migration involves multiple challenges, some related to the classic overheads involved in migration, others related to the fact that the target architecture might be different from the original one.

4.5.2. Node Composition and Dynamic (Re-)configuration of the Communication Infrastructure

OpenStack will be the managing software for the environment setup phase. Specifically, all tasks that are necessary for the resource management, node provisioning and setup of the communication infrastructure will be done through OpenStack.

The node composition and the dynamic reconfiguration of communication infrastructure at runtime of the RECS|Box system is a mechanism to connect multiple resources (nodes and PCIe devices) together to logical units utilizing the
high-speed-low-latency communication infrastructure and to change the communication infrastructure within a composed node during runtime. Although it is a vital part and needs to be taken into account for the design of the frontend systems, it is less obvious for users for direct interaction and thus is described in the backend-chapter 5.4.2 to keep it in one place.

4.6. Conclusion

This chapter presented the definition and the design of the front-end tool-chain, to be developed as part of LEGaTO’s Work Package 4 (WP4). The tools developed in WP4 offer an application programming interface, and will be used by the applications developed in Work Package 5. The same tools are in turn supported by a corresponding run-time system, developed in Work Package 3.

The front-end toolbox specification was presented as a number of extensions to the OmpSs programming model, including distributed dynamic scheduling of tasks across multiple nodes, energy efficiency, fault-tolerance and support for both Dfiant and Maxeler dataflows. The chapter contains a comprehensive description of the extensions to the OmpSs annotations, an Integrated Development Environment, efficient support for irregular data structures, directive-based checkpointing, static kernel identification, trusted tasks and compiler-based protection against security threats and software faults. Finally, it presents how LEGaTO will extend OpenStack extensions to support its task-based programming model and to orchestrate hardware resources at logical level, compose nodes and reconfigure the communication infrastructure.
5. Definition/Design of back-end Runtime System

5.1. Executive Summary

This chapter describes the back-end technologies of the LEGaTO toolchain that will be developed in the context of Work Package 3. This report comprises the deliverable D3.1 "Definition/Design of back-end Runtime System (M9)" of the original plan.

5.2. Introduction

This chapter describes the back-end runtime system of the LEGaTO project that is currently being implemented in Workpackage 3. Figure 5.1 shows a high level view of the components that are being implemented in LEGaTO. The figure highlights the components that are the responsibility of WP3 and its relation to the other parts of the LEGaTO toolchain and hardware. A more detailed diagram of the LEGaTO runtime back-end is depicted on the right side of Figure 5.2. The LEGaTO runtime system consists of those components that are operative during the execution of the LEGaTO application. The following sections describe these components. We begin in Section 5.3 by detailing an abstract task-based energy model that could potentially be useful in developing the task-based energy optimizations in WP3. We continue by describing the hardware management middleware in Section 5.4. Next, Section 5.5 describes the runtime library. Development and performance tools are then described in Section 5.6, while fault-tolerance schemes are finally described in Section 5.7.
5.3. Task-based Energy Model

This section introduces several concepts used throughout the document. First, metrics related with energy consumption are discussed, followed by a discussion on the coarse grain analytical energy mode for task-based programming models.

5.3.1. Energy-related Metrics

We will consider two main metrics for energy: Energy and Energy-Delay Product. The energy $E$ needed to execute an application is defined by the execution time $T$ and the power $P$ of the system.

\[ E = P \times T \]

Thus, we can aim to reduce energy by reducing execution time, power or both. Many well-known techniques to reduce execution time typically incorporate more resources to the system that imply an increase of power dissipation. For example, adding processors can speed up the application; however if the speed-up is not sufficiently high the additional power dissipation of the added processors may increase the system energy consumption.

Energy-Delay Product (EDP) weights execution time more than power. This metric is valuable even for energy-constrained systems because performance is still a concern. Although the rest of this document mentions Energy as the optimization metric, most of the LEGaTO optimizations target power-savings with minimal performance impact, therefore the EDP gains will be comparatively more significant than Energy savings. Thus, wherever appropriate, EDP results will be reported in addition to Energy.
Energy savings typically incur costs on other parts of the system. For example, if we lower the voltage, we can gain energy, with the cost of a higher failure probability. To handle the higher number of failures, we have to spend energy on reliability measures. Therefore, there is a tradeoff between the energy cost of reliability (and security) and energy savings, the LEGaTO aim is to maximize energy savings while minimizing the energy cost of reliability and security optimizations.

We now define the factors that influence the energy consumption of an application. Basically the energy consumption of a LEGaTO technology ($E_{\text{LEGaTO\_Technology}}$) is the difference of energy gains ($E_{\text{gain}}$) and costs ($E_{\text{cost}}$) in comparison to the energy consumption of the baseline ($E_{\text{baseline}}$). There can be several different sources of gains and costs.

$$E_{\text{LEGaTO\_Technology}} = E_{\text{baseline}} - \sum E_{\text{gain}} + \sum E_{\text{cost}}$$

Depending on the requirements of the resources, the application and the developer; one could define weighting factors for the different costs and/or gains. This would allow us to define individual energy profiles.

### 5.3.2. Energy Model Customized for LEGaTO

We first introduce customized analytical models for the task-based applications running on the heterogeneous hardware substrates considered in the LEGaTO project. The analytical models described below will be used as a basis for calculating the theoretical limits of the researched LEGaTO tools. These limits depend on many parameters and characteristics that may not be defined or known yet. Moreover, it is likely that the models will need to be refined in order to bind the theoretical limits in a more fine-grained way.

Energy savings will be with respect to a sequential application running on a typical high-power server-class Intel processor. Then, we compare it with the task-based version of the application running on the heterogeneous substrate.

To create the task-based version, the sequential program is annotated with tasks, so the sequential style of legacy applications could be preserved (in the style of OpenMP). After taskification, the programmer sees a number of static tasks, each of which could be mapped into multiple dynamic instantiations at runtime.

We assume perfect power-gating so there is no static power consumption if a unit is not used. We assume a given application to be composed of $N$ static tasks. We further assume each dynamic instantiation of a particular task to take the same execution time $T_{\text{cpu}}$, $T_{\text{gpu}}$ and $T_{\text{fpga}}$ on the particular hardware substrate they are executing (CPU, GPU or FPGA respectively). Assume each task $i$ is instantiated $k_{i\text{cpu}}$, $k_{i\text{gpu}}$ and $k_{i\text{fpga}}$ times on the CPU, GPU and FPGA respectively. Then, total energy is:
The energy consumption related to off chip memory access, where \( M_{cpu}^j \), \( M_{gpu}^j \) and \( M_{fpga}^j \) are the total number of main memory accesses for the \( j \)-th instantiation of a task for the CPU, GPU and FPGA substrates respectively, is given by:

\[
E_{mem} = \sum_{i=1}^{N} \left[ (\sum_{j=1}^{k_{cpu}^i} P_{cpu}^j \times M_{cpu}^j) + (\sum_{j=1}^{k_{gpu}^i} P_{gpu}^j \times M_{gpu}^j) + (\sum_{j=1}^{k_{fpga}^i} P_{fpga}^j \times M_{fpga}^j) \right]
\]

And for communication, where \( M_{cpu}^j \), \( M_{gpu}^j \) and \( M_{fpga}^j \) are the total communication cost (in bytes transmitted or received) for the \( j \)-th instantiation of a task for the CPU, GPU and FPGA substrates respectively, is given by:

\[
E_{comm} = \sum_{i=1}^{N} \left[ (\sum_{j=1}^{k_{cpu}^i} P_{cpu}^j \times C_{cpu}^j) + (\sum_{j=1}^{k_{gpu}^i} P_{gpu}^j \times C_{gpu}^j) + (\sum_{j=1}^{k_{fpga}^i} P_{fpga}^j \times C_{fpga}^j) \right]
\]

We will leverage the above model to perform early what-if studies for the various energy-saving techniques that will be developed during the project.

### 5.4. Resource Management

The LEGaTO runtime relies on a heterogeneous hardware platform with CPU, GPU and FPGA capabilities. The specification of the hardware is described in chapter 6. With the goal to easily make the hardware accessible to the user as well as to the runtime, OpenStack will be established as a middleware layer on top of the integrated management software (RECS_Master) and the firmware of the hardware platform.

OpenStack will be manually utilized by the cluster administrator during the phase of the environment setup. As first setup step the user will specify the necessary resources from the inventory list, comprising all nodes and PCIe devices of the hardware platform. By doing this he/she also has the possibility to utilize the high-speed-low-latency communication infrastructure by defining com-
posed nodes, which will be described in section 5.4.2. After gathering the resources and configuring the Ethernet network, the CPU and FPGA nodes can be provisioned from within OpenStack, which completes the setup process. Afterwards, the applications and LEGaTO runtime will interact with the resources independently from OpenStack. An optional dynamic reconfiguration of the high-speed-low-latency communication infrastructure at runtime will then be done by directly triggering the Redfish API. See sections 5.4.1 and 5.4.3 for a description of the Redfish API and the dynamic reconfiguration respectively.

The middleware development in LEGaTO relies on results of the Horizon 2020 project M2DC which is running until mid of 2019. At this point of time it is not completely foreseeable in which state the middleware will be at the end of M2DC. In any case, it is foreseen to be adapted and enhanced within the LEGaTO project to match the needs of the runtime and applications. Besides stabilizing the middleware and installing it on the project’s testbeds, the main enhancements will be to enable it to make use of the dynamic reconfigurability options of the hardware/firmware that will also be developed within this project and are described in section 6.3.4.

In this chapter the components and mechanisms of this layer will be described from bottom to top. For the sake of a comprehensive view of the complete middleware stack, this description covers the work which is part of the M2DC project as well as the work that is planned to be realized in the LEGaTO project. Within the latter the RECS_Master and its Redfish API as well as the underlying firmware directly running on the infrastructure hardware platform will be extended and adapted to refine the process of node composition, which will only have basic functionality after the end of the M2DC project. Especially the functionality for the dynamic reconfiguration of the high-speed-low-latency communication infrastructure at runtime is part of the LEGaTO project. This will improve the possibilities of the resource usage. In addition to that, the OpenStack components have to be adapted to match the modified node composition process and their usability will also be enhanced along the way.

Figure 5.3 shows all components of the software stack for managing the resources of the heterogeneous hardware infrastructure.

The firmware of the RECS_Box (which consists of very low-level firmware parts and the RECS_Master above that) is the low-level responsible part of the software stack to manage the heterogeneous resources of the hardware platform. This hardware-related part is integrated within the server’s chassis and thus described in section 6.3.4.

5.4.1. Redfish API

The management and composition of resources is one main task of the middleware layer. A Redfish API was defined in the M2DC project to enable the RECS_Master to provide detailed information about the hardware system and allow operation calls for managing the available resources via OpenStack directly. Although the Redfish API is also part of the hardware’s firmware (see Section 6.3.4), it is described in this chapter to give a better overview of the management possibilities.
The Redfish Scalable Platforms Management API from the Distributed Management Task Force, Inc (DMTF) [26] is designed to perform out-of-band management of multiple systems at once and therefore suits the needs of our hardware architecture quite well. It describes a RESTful interface on top of a data model [27] specified in both CSDL (Common Schema Definition Language) and JSON Schema. This model is capable of expressing the relationships between components in modern systems and is designed to be extensible. The payloads of HTTP requests to and responses from an implementation of a Redfish API are expressed in JSON following OData JSON conventions and can therefore easily be interpreted by clients.

In the M2DC project the schema Redfish API was extended to reflect the characteristics of the hardware platform and allow the composition of resources (described in the following section). The complete documentation of the adapted Redfish API is online available [20] and might be further extended during the project to meet the application’s needs. The schema consists of entities describing the corresponding hardware resources, such as chassis, computer systems, PCIe devices, switches, ports and other components. For the task of node composition also schemas for composed nodes and high-speed-low-latency connections are available. As this API is still under development in the M2DC project and will be further enhanced in the LEGaTO project, the documentation will also evolve over time.

Figure 5.3. LEGaTO middleware stack for resource management.
5.4.2. Node Composition

The idea and general process of node composition is derived from and loosely based on Intel’s Pod Manager API specification [46]. In our case, node composition is a mechanism to connect multiple resources (nodes and PCIe devices) together to logical units utilizing the high-speed-low-latency communication infrastructure. By controlling the switches within this communication network (transparent for the software layers above) different basic communication scenarios and thus node compositions are possible:

Node to Node communication (Host to Host)
This scenario allows building a communication network, similar to Ethernet, based on the PCIe infrastructure. Unlike conventional PCIe that is used to communicate between a host (root complex) and peripherals (endpoints), a communication between multiple hosts is possible.

Host to PCIe peripheral communication
This might be one of the most common scenarios, which allows the connection of hosts to PCIe peripherals.

Multi-Root I/O Virtualization
PCle devices, that support single-root I/O virtualization can be shared across multiple nodes, extending the single-root to multi-root I/O virtualization. In some cases this can enable an efficient usage of PCIe devices by sharing them within a composed node.

Dual-Socket functionality
One of the ARMv8 COM Express based microservers that is available for the RECS|Box allows to be operated in a dual-socket manner by connecting two of them on one carrier-blade within a RECS|Box.

It is also possible to combine these scenarios which allows the implementation of comprehensive or complex use cases.

The process of node composition will be done from within the OpenStack component Valence. As the first step, the user provides a set of requirements describing the node he wants to compose and tells Valence to allocate corresponding resources. These requirements can either be concrete hardware entities such as microservers or value sets specifying the desired properties of the composed node such as processor cores or communication capabilities. A best match based on these requirements is then calculated and the resources are reserved by creating a composed node entity out of them. After that, this allocated composed node is presented to the user as a proposition, who can either accept the offered composition by ordering Valence to assemble it or reject it and provide adjusted requirements. After this last step, the composed node is ready to be used. Figure 5.4 describes the state changes of a composed node to illustrate the process of node composition.

5.4.3. Dynamic Reconfiguration of Communication Infrastructure at Runtime

During the runtime of the application it will be possible to reconfigure the high-speed-low-latency communication infrastructure within a composed node. One possible use case for this feature is the change of the communication topology...
within a pure multi-FPGA environment. For the sake of an efficient communication it can be beneficial to start the computation of an algorithm with a simple ring topology and then switch to a full mesh topology in later phases of the algorithm. Allowing this dynamic reconfiguration at runtime can improve the efficiency of algorithms. Figure 5.5 illustrates the described use case.

In order to use this feature, the user has to specify all envisioned communication configuration sets at the composed node setup while setting up the environment in OpenStack. In addition to that he has to let the computation algorithm trigger the topology changes by directly calling the Redfish API and therefore bypassing OpenStack for a faster process. The RECS_Master then instructs the low-level parts of the firmware to dynamically reconfigure the communication infrastructure to the new topology. After the necessary reinitialization of the communication links between the involved communication partners is done, the algorithm can continue the computation work with the new communication topology.

**5.4.4. OpenStack Middleware**

As already introduced in the frontend chapter, OpenStack will be the managing software for the environment setup phase. Specifically, all tasks that are necessary for the resource management, node provisioning and setup of the communication infrastructure will be done through OpenStack. The only exception is the triggering of the dynamic reconfiguration (c.f. section 5.5) of the communication infrastructure at runtime, which will be done by directly calling the Redfish API.

OpenStack has a modular architecture and clearly separates tasks to different
components. Therefore, various components will be utilized for the setup tasks involved in the LEGaTO project, which are described in this section. It should be noted, that the integration between these components is still ongoing within the OpenStack work groups. Furthermore, the basic adaptation work for these components will be done within the M2DC project, which is still under development. The LEGaTO project will do the adaptation of the OpenStack environment based on the intermediate and final results of the other project. LEGaTO plans to upstream especially the contribution to Valence, which is dependent on the M2DC work.

The main components of OpenStack that are used within the LEGaTO project are shortly described in the following paragraphs. These descriptions are summarized from the more detailed explanations in the confidential deliverables 4.8 [38] and 6.1 [39] of the M2DC project.

**Relevant OpenStack Components**

**Horizon** provides a web based user interface for all other OpenStack services.

**Nova** is used to organize the deployment of applications or images to compute nodes. Amongst other features, it supports creating virtual machines, provisioning of baremetal servers (by utilizing Ironic) and has limited support for system containers. Nova also integrates its own scheduler to select target nodes based on different parameters. It is not yet decided, if this functionality will be utilized within the LEGaTO project. Nova also provides an external usable API.

**Ironic** enables the dynamic bare-metal provisioning of x86 or ARM nodes. Baremetal deployment reduces the computational overhead compared to virtualization and is therefore ideal for the microservers within the hardware platform. Furthermore, it eases the use of the special capabilities of the RECS|Box like PCIe switching, attaching accelerators etc.

**Neutron** is the OpenStack component for managing the network connectivity between the nodes and switches. It also provides an API.

**Valence** is the basis to handle and interface composable architectures. In the projects M2DC and LEGaTO it is utilized to manage composable nodes, like it is described in section 5.4.2. In order to accomplish this, Valence communicates with the RECS_Master through the Redfish API.

**Cyborg** is used to manage an inventory of accelerators in OpenStack managed systems and to deploy software to them. In LEGaTO it will be used to provision FPGA nodes.

This list is not complete as it only covers the primary components, which are involved in the setup process within LEGaTO. Secondary components, e.g., for authentication, are also part of the middleware solution but not important to mention here as they are used in their typical way.
5.5. The LEGaTO Runtime Library

The runtime library involves the LEGaTO components that are active during the execution of the application. Its main tasks are 1) handling the offloading of tasks to LEGaTO devices (GPU, FPGA, Nanos and XiTAO), and 2) implementing energy-efficient scheduling strategies to decide at runtime on which type of device and on which instance to map a particular task. In addition to this, it will also implement techniques for efficient power management.

5.5.1. Hardware Topology Discovery

The hardware topology discovery component runs during the application startup phase. The module queries the node management software for all present hardware and builds an in-memory database of components. By using information obtained from SLURM and/or OpenStack, the component can retrieve information concerning the other nodes assigned to it for execution and build a global view of the available hardware. The database of components represents the topology of the hardware partition on which the LEGaTO application is running. The topology should include costs for network components, so that the cost of data communication can be estimated and taken into account during scheduling. If the underlying topology changes due to dynamic node reconfiguration, a callback function needs to be called to update the in-memory database of components and network costs. The topology needs to clearly identify which CPUs are available, how much memory is ready to be used, which CPU components share memory, which accelerator devices are available on which nodes, and which CPUs belong to nodes that pertain to different physical memory domains.

5.5.2. GPU Components

The LEGaTO project will focus on GPUs developed by Nvidia kernels executed by these devices will not be generated by the LEGaTO toolchain, instead they need to be provided externally and linked to the LEGaTO application by using the OmpSs target device(cuda) or target device(opencl) directives. To execute GPU kernels, LEGaTO will utilize the CUDA libraries and drivers which are preloaded on the boards of the RECS|Box platform nodes that have GPU hardware, or the corresponding OpenCL libraries and drivers for GPU hardware supporting OpenCL.

The hardware description will include:

- Vendor.
- Model.
- Amount of internal shared memory.
- Amount of global memory.
- Directory where to search for available kernels.

5.5.3. FPGA Components

The LEGaTO project will work with FPGA accelerators from various vendors. They will be used from OmpSs by providing the code targeting the FPGA annotated with the target device fpga directive. In the FPGA case, when the FPGA component has a High-Level Synthesis tool provided by the vendor, such tool
will be used to compile the code annotated to VHDL/Verilog and a bitstream will be automatically generated during the compilation phase of the application.

Regarding the FPGA components, the hardware description will provide a description of the specific FPGA device, including:

- Vendor.
- Model.
- Part number of the specific FPGA chip.
- Amount of local memory.
- Amount of global memory.
- Directory with the available bitstreams.

If the FPGA vendor supports OpenCL for a specific FPGA, that FPGA will be also available as an OpenCL platform and device from OmpSs.

### 5.5.4. CPU components

The libraries needed for executing on CPU components are the same as needed for any other application, with the exception of the runtime components themselves which run on the CPU. The two runtime components are the Nanos runtime and the XiTAO runtime. Besides these runtimes, we assume that the standard C and C++ runtime libraries (libc and libstdc++, or libc++ -if LLVM is used-) are installed on the system. Finally, the systems also need to provide an implementation of the pthread library and support for C++ threads (≥C++11).

The hardware description will include:

- Vendor.
- Model.
- Amount of cores and, potentially, the type of each core.
- Amount of NUMA nodes and the mapping of each core.
- Amount of shared cache memory.
- Amount of DRAM memory.

### 5.5.5. Maxeler Run-time Components

Maxeler MaxCompiler compiles MaxJ kernels and manager code into a binary that can be loaded and run on the DFEs. This DFE binary needs to interface with a CPU host application. Therefore, Maxeler also provides several software components for seamless integration with CPU host applications as well as runtime management functionality. This platform consists of the following:

1. Simple Live CPU interface (SLiC) API to seamlessly integrate calls to DFEs into CPU host applications;

2. MaxelerOS, a runtime layer between the SLiC API, the Linux operating system and the DFE hardware, which manages CPU-DFE interactions in a transparent way.

The SLiC API provides the interface between a host application and DFE. SLiC allows CPU applications to configure and load a variable number of DFEs as well as to subsequently schedule and run actions across those DFEs using simple function calls.
MaxelerOS is software layer and runtime sitting between the SLiC interface, the Linux operating system and the hardware, which manages DFE hardware and CPU-DFE interactions in a way transparent to the user. MaxelerOS consists of the following four components:

- Driver: it provides low-level access to local cards;
- Daemon: supports configuring and monitoring local cards by using Utilities;
- Utilities: command-line tools for managing local cards and MPC-X nodes;
- libmaxeleros: a shared library used by the MaxCompiler SLiC interface to communicate with local or remote DFE cards.

MaxelerOS is available in several RPM versions for CentOS or Red Hat Enterprise Linux 6/7, targeting different host configurations. The basic RPM supports a host node with locally installed DFEs and does not provide a dependency on Infini-band libraries. For a host with access to remote DFEs on MPC-X nodes the RPM with InfiniBand dependency is also provided.

Maxeler provides several tools for DFE monitoring and management. **Maxtop** is part of the MaxelerOS utilities and it is similar to the Unix command “top”. It allows users to view both static information about the DFEs present in the system such as board model, memory capacity and local interconnects. It also monitors dynamic DFE parameters such as utilisation and power consumption. Maxtop can show the status of local DFEs, or remote DFEs in an MPC-X node. The same information can also be retrieved through a JSON interface.

**Maxdashboard** is a web browser-based GUI for MaxelerOS and Maxtop and presents DFE information in a graphical environment.

**Mpcxtool** provides a range of monitoring and management functionalities for MPC-X node. It enables remote powering on and off of MPC-X nodes, log retrieval, reading internal sensors such as voltages and current power draw at the mains socket level, as well as a number of DFE-specific operations. Unlike maxtop which manages DFE via the dataflow plane, mpcxtool manages the entire node via a separate Ethernet connection which is typically connected to a dedicated management network.

The Maxeler **Orchestrator** manages the entire pool of remote DFEs across multiple MPC-X nodes. In large-scale systems with many remote DFEs, congestion or race conditions can arise if the same DFE resource is accessed by multiple applications. This is addressed by the Orchestrator. It automatically discovers DFE resources on the Infiniband network, and applications can request single DFEs, or multiple DFEs in a range of topologies via a reservation mechanism. The purpose of the reservation mechanism is to allocate and block a DFEs until explicitly released. The Orchestrator manages which physical DFEs will be allocated for a request.

### 5.5.6. Heterogeneous Scheduling

The LEGaTO system will include a heterogeneous scheduler that will search for the optimal combination of resources to be used when CPUs, GPUs, DFEs and
FPGAs are available to execute a collection of given tasks. The implementation will extend the task-based scheduler currently available in OmpSs to a heterogeneity-aware scheduler.

Currently, tasks can be provided in several flavors, each one providing an implementation of a particular algorithm on the resources available on the platform (CPU, GPU, FPGA). The programmer uses the `target device(dev) implements(funcname)` clause to indicate that the annotated function implements the same algorithm as `funcname` in the specified device (dev). The referenced function (`funcname`) can be targeting the same or another of the accelerator types.

With this information, the compiler builds a table of algorithm availability for each particular task. When executing, the scheduling policy can use this information and apply various techniques, for example:

- Execute the task in the first type of device that becomes available.
- Measure execution times in the available devices during the very first executions of the task, record them into a performance tracking table, and then select the one better fitting the algorithm and the input data.
- Select the best device based on the profile of previously executed tasks and use the current informations to update those profiles.

Once this first scheduling phase is done and the type of device on which to execute the task is known, the final step is to decide on which device of the given type the task should be executed considering the availability of multiple devices of the same type.

In the context of the LEGaTO project, we will provide a tuned version of these scheduling techniques for the hardware and accelerators available in the project.

5.5.7. Topology-aware Scheduling

The topology-aware scheduler chooses mappings based on communication-costs. There are two mappings that can be done: 1) Global mappings, which choose the node on which to execute a task, and 2) Local mappings, which choose the core, GPU or FPGA within a particular node. The location on which a task is executed is called its “place”.

In Nanos Clusters, global scheduling is done based on data locality, load balancing and data affinity hints, as described in Section 5.5.9. In order for the runtime system to estimate the communication costs, needed for data locality, it will require (a) a concise, hierarchical representation of the inter-node communication latencies and/or bandwidths and (b) a means to map from the MPI process number or node hostname to the physical location in the hierarchy.

For components under control of the XiTAO runtime, the topology annotations described in Section 4.4.5 will be used to devise topology-aware schedules, in combination with information extracted from the performance tracking table (see Section 5.6.2).
5.5.8. ML-support for Heterogeneous Scheduling

Optimal scheduling of tasks on heterogeneous hardware is a challenging task due to many factors, most notably the lack of a priori knowledge about the execution time of tasks. Machine learning (ML) can contribute to improved scheduling and reduced energy consumption and execution time. We plan to carry out two iterations of ML-based scheduling algorithms. In the first iteration (A) a ML system is trained off-line to predict energy consumption and execution time on heterogeneous hardware. This system will complement the simple performance table suggested in Section 5.6.2. In the second iteration also the placement of tasks will be handled by the trained ML system. The training procedure differs significantly between the two different systems. In iteration (A) the system is trained as a function approximator on logged data, i.e., supervised learning. In iteration (B) the system learns to place tasks by actually placing them and then analyse the outcome to self improve, i.e., reinforcement learning. The former requires logged data to be used for training, while the latter requires a simulation environment to learn by doing.

Prerequisites

1. Analyse what features in the task definitions can be used as inputs to the ML system.
2. Analyse how informative and general these features are to predict execution time of tasks.
3. Select features to be used.

Iteration A

A4. Collect data and statistics of energy consumption and execution time from a pool of sample programs to be used as training data.

A5. Train a ML system that predicts the execution time and energy consumption of tasks on the available hardware based on the features selected in step (3).

A6. Plug in these estimations in the current infrastructure to further improve the scheduling based on simple performance tables.

Iteration B

B4. Create a hook-in in the runtime so that it can be used as a simulation environment to train the ML algorithm via reinforcement learning.

B5. Train a system that, based on a pool of tasks, schedules tasks on a heterogeneous hardware testbed in an energy aware way.

One of the major challenges that needs to be taken into account when designing these systems is the overhead of the algorithms themselves. Another important aspect is fault tolerance, which can be simulated and therefore learned to handle gracefully in iteration (B). A risk of a centralised scheduler is that it itself becomes a bottleneck. Approaches to mitigate this limitation will be explored.
5.5.9. Distributed Scheduling

Nanos Clusters makes use of a distributed scheduler that operates at two levels: global scheduling and local scheduling. The global scheduler’s task is to distribute tasks among the available nodes. In particular for Nanos Clusters, global scheduling is based on (a) data locality, (b) load balancing and (c) data affinity. Scheduling tasks onto nodes that already store a task’s required data reduces the latency to access the data and lowers the overhead created by transferring the data from another node. Avoiding data transfer is crucial for performance, especially since the network bisection bandwidth becomes a bottleneck. Minimising the movement of data across a network maximizes the use of most precious and costly resources (i.e., the network) by increasing efficiency and reliability of the nodes both on a cost as well as a processing basis. Node selection is currently approximated by choosing the node with the greatest amount of required data. However, placing tasks close to their data can conflict with load balancing requirements, if other tasks cannot be scheduled onto the same node. Hence, there is a trade-off between load balancing and minimising communication. The load generated by the tasks is balanced by avoiding idle processing resources. Idle processing resources query work queues of other processing resources and steal their tasks if possible. Applying this technique efficiently distributes the scheduling work over the processing resources. Finally, the global scheduler is making use of hints provided by the programmer with respect to data affinity. In the long term, these hints should be eliminated, but for the moment, the scheduler has to rely on this additional information annotating the source code.

Local scheduling is using primarily data locality and load balancing to run tasks on a single node after being placed there by global scheduling policies. Data locality on a single node refers to the allocation of data on NUMA nodes and/or device memory such as for graphic cards or FPGAs. Global location information and potentially local location information is passed by the runtime system within inter-node control messages that signal task completion. To make the most efficient use of the devices on the machine a task will be split into sub-tasks by the local scheduler to balance the load. Making effective use of this information in the scheduler, without introducing excessive overhead, taking into account both the computational cost of determining where to execute the task and the communication cost of transferring the necessary metadata, is the subject of ongoing research.

5.5.10. Elastic Place Management

The elastic place manager is the scheduling component internal to the XiTAO runtime. It may be considered at a later stage for integration into the Nanos runtime as well. This module chooses the place “granularity” for a task assembly object (TAO in XiTAO terminology) and assigns a partition of resources, which is called a “place”. The place can be chosen with multiple goals in mind: for example, to minimize execution time, to minimize cache misses or to minimize energy. A runtime setting will determine the target scheduling objective. Elasticity is a XiTAO mechanism that is designed to balance parallelism, overheads and locality, and to eliminate inter-task interference. Elasticity has recently been shown to be useful in combination with online performance tracking to generate...
efficient schedules [70, 32].

LEGaTO will extend the Elastic Place manager implemented in the XiTAO prototype for single-ISA heterogeneous platforms such as as ARMs big.LITTLE. The current architecture of the XiTAO prototype requires all cores to share the same memory. The ARM big.LITTLE platform is hence a good candidate for extending XiTAO and for evaluating the effects of interference-free scheduling for both energy efficiency and performance.

The elastic place manager exists as a component of the XiTAO runtime. In the context of LEGaTO, XiTAO will receive tasks to execute as part of the offload mechanism in which tasks are offloaded from a master thread via the cluster scheduler. For better flexibility, these tasks should be task assembly objects (TAO). Simple (static, single-threaded) tasks can also be executed via XiTAO, but the lack of flexibility limits the optimizations for such types of tasks. Here we discuss only the implementation for TAOs.

TAOs are moldable collections of work. Moldable means that a variable number of resources can be assigned at runtime. Collection of work means that the TAO encapsulates some amount of work that can potentially be distributed over multiple workers. The runtime itself is unaware of the internal structure of a TAO. It is only concerned with assigning resources and starting/finishing the execution of the TAO. TAOs are internally self-scheduled. A common idea is to think of a TAO as a nested parallel runtime. The structure of a TAO is shown in Figure 5.6.

The task of the elastic place manager is to assign resources to and to execute a TAO. The problem of assigning resource and scheduling the TAO for execution needs to be handled with care. Assigning resources has implications in terms of parallelism, overheads and locality. Scheduling, on the other hand, has implications on idleness and overheads. Choosing the appropriate resource partitions for execution will be one of the main research topics within the LEGaTO project. In terms of scheduling, we plan to leverage the Dynamic Place Allocation (DPA) algorithm that is implemented in the XiTAO prototype and extend it for LEGaTO. DPA performs a pipelined, overlapped and deadlock-free scheduler for executing multiple concurrent TAOs with high performance. The DPA algorithm is explained in detail in [70]. Figure 5.7 shows the structure of the DPA algorithm in the context of a homogeneous architecture.

The extension of DPA for heterogeneous systems requires considering multiple types of cores. This is usually handled in a consistent interface to the operating
system (Linux) by selecting hardware threads. In LEGaTO, this selection will be done either based on high level information provided either by the compiler or the programmer, or it will be done by querying a performance table. Details on runtime-aware scheduling are discussed in Section 5.6.2.

5.5.11. Runtime Power Management

The runtime power management module will utilize hardware performance and power counters developed in Chapter 6 in order to establish an instantaneous power and energy map of the LEGaTO hardware/software stack. The power monitoring utility will utilize Running Average Power Limit (RAPL) feature for x86 processors or ACPI states for ARM and Intel processors and XADC IP or PMbus for Xilinx FPGAs. This power information will be propagated to the OmpSs and XiTAO runtime which will take action in the Heterogeneous Scheduling model to select the best energy fitting hardware substrate for incoming jobs. Runtime power management will combine the hardware power information with the programmer annotations for energy and other task-specific runtime information. Note that the nanos++ runtime and XiTAO libraries are built on top of POSIX thread support and as such do not support thread suspend/resume because of the problem of suspending a thread when it holds a lock.

5.6. Performance and Correctness

In addition to the runtime components in charge of application execution, several components will be designed that aid in obtaining higher performance and better quality of implementation. These components include a resource monitor that selects the hardware partition provided to the LEGaTO application, a performance monitor that provides online feedback to the runtime components for better scheduling, and finally a debugging component to aid in the implementation of correct LEGaTO/OmpSs applications.
5.6.1. Resource Allocation and Monitoring

In a standard version, SLURM comes with support for managing additional resources like accelerators. Out of the box SLURM supports Graphics Processing Units (GPUs) and Intel Many Integrated Core (MIC) processors. These devices are named in SLURM Generic Resources (GRES) and can be assigned to the nodes using the SLURM configuration. SLURM monitors whether the devices defined in the configuration are present on the nodes and gives users of the computational cluster the possibility to reserve and use these resources. Moreover, interfaces for plugins is provided that supports more specific resources in the cluster. Within the M2DC project, this plugin was used to properly initialize the devices and, e.g., reconfigure them in order to fulfil user-specific requirements. Doing this, FPGAs can be used and deployed with user-specific code.

The LEGaTO job scheduler will be implemented with SLURM, and will perform resource allocation and monitoring. After the cluster setup phase, the SLURM resource database will be populated with the information obtained from the OpenStack database, regarding the cluster nodes and resources available in the system [57]. This information will be provided to running applications, so that they can reconfigure communications among nodes, and know about the devices attached to them (see also Section 5.5.1).

Additionally, SLURM will be provided with a specific plug-in to monitor the status of the resources available in the system. When a resource is detected to be offline, SLURM will try to replace it with an equivalent resource, if available. This way, new applications starting in the system will be allocated to the new node (or nodes), in a transparent way.

5.6.2. Performance Monitoring

Runtime performance monitoring is one of the main mechanisms to implement efficient scheduling in LEGaTO. The performance monitor will collect statistics on the execution of the different application components (such as tasks) and make this information available to the runtime schedulers. The statistics to be collected include execution time, cache misses and/or energy per task. The runtime schedulers can then use this information to take scheduling decisions with certain targets in mind. The targets that will be considered in LEGaTO are 1) minimum execution time, 2) minimum energy, and 3) minimum energy-delay product.

XiTAO implements a simple performance tracking table that can be used to make intelligent selections of cores at runtime [32]. The performance tracking table (PTT) contains a row for each possible initial assembly worker in a TAO and one column for each size of partition. Every time that a TAO finishes execution, the value in the PTT is updated by using averaging over a number of executions. A representation of this table is shown in Figure 5.8.

Initial research has found the PTT to be a simple yet powerful way to track both performance scalability and system interference. The latter is not measured directly but observed indirectly. In scenarios of high concurrency, assigning small resource partitions leads to higher parallelism, which can lead to higher contention on the memory subsystem. Although the PTT cannot measure parallelism and interference, it observes an increase in the execution time due to the extra
interference. Hence, by scanning the values in the PTT it is possible to improve decision making in scheduling.

In LEGaTO, the PTT will be extended to track heterogeneous systems and to support all the runtimes in the system, including the global scheduler. We will research methods to provide richer information to the runtime, including information on energy consumption and cache misses, which have not been implemented in the current approach. Using this additional information LEGaTO will target more energy efficient scheduling.

5.6.3. Debugging support

We will integrate the ability to execute LEGaTO applications under the control of the GDB debugger into the Eclipse/OmpSs plugin. With this tool, OmpSs environment variables will be automatically set before the execution with the debugger, and the proper, debugger-enabled binaries will be selected for execution.

Following the same approach that we use to trace the behaviour of the internal execution of the IPs in the FPGA, we will provide capabilities to obtain the values of variables from inside the instrumented IPs, to be able to debug the internal implementation on the code running on the FPGAs.

5.7. Fault Tolerance

Although hardware based fault tolerance mechanisms such as ECC (for memory) and CRC (for interconnect) incorporate strong fault detection and correction properties, they do not have sufficient fault coverage, i.e., they do not provide protection from a large class of faults such as those that impact the combinational logic that are dominant in current computing substrates. Therefore, software approaches are essential for energy efficient reliability. One such classic approach, termed checkpointing, takes regular snapshots of current state of the computation to memory or disk and restarts the computation from the latest such checkpoint when a fault is detected. The LEGaTO project will develop a consistent standardized checkpointing tool for the heterogeneous substrate of CPUs, GPUs and FPGAs. This tool will be multi-level: based on user annotations, it will take checkpoints to main memory or the file system to maximize reliability while minimizing the energy footprint of checkpointing.

For certain fault classes such as silent data corruption, it is necessary to run replicated copies of tasks and then compare their results to detect and correct any errors. However, complete task replication is not energy efficient; requiring
partial replication schemes.

The combination of the above methods will increase reliability by $10 \times$ while minimizing the energy overheads.

### 5.7.1. Checkpointing of GPUs and FPGAs

Heterogeneous systems need to protect executions regardless of where they are taking place. This is challenging due to multiple limitations coming from vendors and from the low maturity level of these devices. It has been shown [58] that GPUs are two orders of magnitude more propitious to experience uncorrectable errors than the host in which they are installed. This makes the resilience aspect on these devices even more important.

In addition, low-power heterogeneous systems are growingly exploiting FPGAs, expected to be in 30% of supercomputers, according to Top500 news [33]. Similar to other digital devices in nano-scale technology nodes, FPGAs are also susceptible to errors stemming from aggressive undervolting, SEUs, manufacturing defects and radiation-induced faults. State-of-the-art FPGAs are composed of a wide set of components such as DSPs, SRAMs, and LUTs, and also hardware processor cores such as ARM CPUs in the SOC architecture. Errors in these components may cause applications to produce wrong results or even to system crashes. Understanding the behavior of errors in different FPGA components and efficiently mitigating them is an approach in LEGaTO to achieve ultra low-power and fault-resilient design.

#### GPU checkpointing

Most efforts on GPU checkpointing try to bypass the driver’s limitations but such implementations suffer from low portability as most of the times the next generations render the previously developed techniques obsolete. Recently, important advancements have been done to integrate fault tolerance techniques such as checkpointing in GPUs. Yet, there is no standard library to transparently checkpoint GPUs. LEGaTO will integrate the declarative checkpointing API with internal features to transparently checkpoint GPUs. Checkpoints can be taken in between two GPU kernels execution or they could be taken inside the kernel if its execution was too long.

#### Checkpointing FPGAs

Comprehensive checkpointing of FPGA designs is still an open issue in the literature; however, there are multiple works aiming to checkpoint and rollback the execution of the application from a safe state. Previous works are mostly limited to software simulated designs and injected faults, which can obviously limit the reproducibility of the results for real faults. In LEGaTO, we aim to extensively extend these studies for commercial FPGAs, real errors such as aggressive undervolting error, and integrate it with run-time tools such as OmpSs.

### 5.7.2. Task Replication

As hardware technology scaling gets closer to the end of Moore’s Law, the number of undetected faults increase leading to Silent Data Corruption (SDC). One way to mitigate SDC is through replication of computation. The computation could be duplicated (called duplex redundancy) or triplicated (called triplex re-
dundancy). While it is relatively more straightforward to replicate tasks in task-based programming models, complete replication of the computation is energy inefficient. Therefore, it is needed to develop partial replication schemes, which replicate the most reliability critical tasks while being energy efficient.

5.7.3. Energy-efficient Multi-level Checkpointing

Recent large scale machines have new storage devices incorporated into the compute nodes increasing the storage hierarchy in the system. This positively impacts the reliability of applications as the deep memory hierarchy allows for a relatively easy way of leveraging multilevel checkpointing in the system. Writing in local storage can save substantial power consumption compared to transfer to the file system. Moreover, non-volatile memories introduce a new level of energy efficiency that can be leveraged. Smart data movement across the storage hierarchy allows for substantial energy waste reduction. Parameters such as the checkpoint size, the failure rate, the application write patterns can influence dramatically the energy consumption of writing checkpoint files at large scale. Taking all those parameters into account allows for large improvements in efficiency.

5.7.4. FPGAs Undervolting

The power consumption of digital circuits, e.g., FPGAs, is directly related to their operating supply voltages. On the other hand, usually, chip vendors introduce a conservative voltage guardband below the standard nominal level to ensure the correct functionality of the design in the worst-case process and environmental scenarios. For instance, this voltage guardband is empirically measured to be 12%, 20%, and 16% of the nominal level in commercial CPUs [5], GPUs [56], and DRAMs [16], respectively. However, in many real-world applications, this guardband is extremely conservative and eliminating it can result in significant power savings without any overhead. Motivated by these studies, we extended the undervolting technique to commercial FPGAs, with a preliminary concentration on on-chip memories, or Block RAMs (BRAMs). Our experiments cover several representative FPGA platforms from Xilinx, a main vendor including a VC707, two identical samples of KC705, and a ZC705. The experimental results show the voltage guardband to be on average 39% of the nominal level ($V_{\text{nom}} = 1V, V_{\text{min}} = 0.61V$), which in turn, directly delivers an order of magnitude BRAM power savings. Further undervolting below the minimum safe voltage, i.e., $V_{\text{min}}$ delivers more power savings up to 40%; however, causes faults occurrence in some locations of some of BRAMs. These faults are the consequence of timing violations since the circuit delay increases by further undervolting. Note that simultaneously downscaling the frequency is a promising approach to prevent the generation of these faults; however, it can limit the energy reduction achievement. Alternatively, our aim is to understand the behavior of these faults, through which customized and low-overhead fault mitigation techniques can be deployed to achieve power saving gains.

The overall methodology is shown in Figure 5.9. Our FPGA design includes raw Read/Write accesses to BRAMs, while their supply voltage, i.e., $V_{\text{CCBRAM}}$ is controlled in the host through the Power Management Bus (PMBus) interface. The onboard voltage regulator with the part number of UCD9248 has the respon-
Figure 5.9. The overall methodology for FPGA undervolting experimental study

Figure 5.10. Power saving gain and reliability costs of FPGA BRAMs through undervolting

...sibility to handle these PMBus commands and set the appropriate voltage to different components, e.g., BRAMs. Note that other FPGA components, e.g., LUTs, DSPs, operate at their default nominal voltage levels. Through experiments on this setup, the overall power and reliability trade-off is summarized in Figure 5.10 for VC707, when $V_{CCBRAM}$ is downscaled from the nominal level $V_{nom} = 1V$ to the minimum level that the FPGA practically operates, $V_{crash} = 0.54V$. As can be seen, BRAMs start experiencing faults in regions below $V_{min} = 0.61V$ with an exponentially increasing behavior up to 653 faults per 1Mbit equals to 0.06%.

Major observed properties of these faults are summarized as follows:

- There is significant variability of fault rate among different BRAMs, which is the consequence of the inherent process variation. Through our experiments, we observed that more than 38.9% of BRAMs never experience faults. Also, among BRAMs the maximum, minimum, and average fault rate are 2.84%, 0%, and 0.06%.

---

1Fault maps are published to the public and available in github. https://github.com/behzadsalami/FPGA-BRAMs-Undervoltig-Study
• The faults locations and rate do not change over time. Also, by experimentally evaluating different data patterns, we observed that the fault rate directly depends on the number of "1" bits since a vast majority of generated faults are '1' to '0' bit-flips.

• More than 90% of undervolting faults are single-bit, and a further 7% are double-bit faults. Due to this observation on the behavior of faults, the built-in ECC of BRAMs can be effective to mitigate these faults. Note that the built-in ECC of BRAMs has the type of Single-Error Correction and Double-Error Detection (SECDED) capability, potentially with good efficiency to mitigate BRAMs undervolting faults.

• Faulty bitcells in a certain voltage stay faulty in lower voltages, as well, and potentially, expand to other bitcells. This property is called Fault Inclusion Property (FIP). Our work experimentally confirms that FIP exists in FPGAs, under aggressive low-voltage operations. FIP can be potentially used to build efficient fault mitigation techniques.

5.8. Next Steps

To achieve the goal of completing the LEGaTO toolchain in the projected timeframe, work in the coming months will focus on producing first versions of the XiTAO and Nanos runtimes that can run on the LEGaTO hardware platform. This work will be driven by selected use cases. In parallel to this work we will work on a first version of the multi-level checkpointing and we will produce a middleware scheme to allow administrators and users to manage the resources of the hardware platform. After these steps have been completed WP3 will work on developing a first integrated Nanos/XiTAO implementation that can run LEGaTO tasks with high resilience on any of the target devices considered in the project.

5.9. Conclusion

Work Package 3 has defined a back-end runtime system for LEGaTO, in which multiple heterogeneous devices and multiple resource management libraries will be targeted using task offloading. The next steps will consist of implementing the LEGaTO runtime and developing scheduling technologies targeting low energy consumption. In addition, we have defined tools for improving the performance and correctness of LEGaTO applications. These tools will provide online monitoring of application execution and provide access to debugging facilities across the heterogeneous hardware stack. We have also defined functionality to support fault tolerance. While such functionality has been researched extensively in the context of CPUs, LEGaTO will implement fault tolerance for FPGAs and GPUs via checkpointing and task replication. Finally, we have defined a novel scheme in which communication topologies are to be defined at the application level. These topologies will be used to minimize energy due to communication and to drive the reconfiguration at the hardware level.
6. Hardware and Firmware Specification

6.1. Executive Summary

This chapter describes the underlying hardware used for the applications and use-cases in LEGaTO. It covers the contents of what was originally described as deliverable D2.1 "Hardware and Firmware Specification" in the project work plan. Figure 6.1 illustrates the interaction of WP2 with the other Work Packages from a global project perspective.

6.2. Introduction

Apart from the hardware architecture itself, this chapter also focuses on the firmware architecture and the interfaces towards the run-time management layer (see Section 5). While the architecture of the hardware used in the data center part of LEGaTO is fixed and will only experience minor changes throughout the project, e.g., during TCO optimization (see Section 6.3.5), the hardware used for the edge components within LEGaTO will evolve during the project lifetime. For this reason, the two cloud platforms in LEGaTO, the RECS|Box (see Section 6.3) and Maxeler’s DFE (see Section 6.5) are already described in more detail, while the specification of the edge platform (see Section 6.4) is still in progress.

6.3. RECS|Box Architecture

The RECS|Box is a heterogeneous cluster server prototype which gives users the possibility to select between multiple compute architectures, network systems as well as network topologies and microserver sizes. The term microserver,
in this context, refers to a self-contained Computer/Server-on-Module (CoM), which integrates all components (e.g., CPU, memory, IO, power), in a small, compact form factor for integration in a server or embedded environment. While existing microserver platforms mostly support a single, homogeneous microserver architecture, RECS|Box supports the full range of heterogeneous microserver technology from CPUs to FPGAs, which can be seamlessly combined into a single chassis. The RECS|Box server architecture supports microservers based on x86 (e.g., Intel Xeon), 64-bit ARM mobile/embedded SoCs, 64-bit ARM server processors, FPGAs, GPUs as well as other PCIe-based acceleration units. This heterogeneity can be used to configure and build the optimal processing platform based on the needs of the application. All the main microserver architectures (CPU, GPU and FPGA) are available in a low-power as well as in a high-performance variant. Like the big-little approach in today’s mobile processors, this feature allows to increase the energy efficiency even more by dynamically switching, e.g., between 64-bit ARM server processors and 64-bit ARM mobile SoCs depending on the current load situation of the application. In the following sections, the modular hardware architecture, the multilayer communication infrastructure, an overview of available microserver types and the firmware-based management options are described.

The RECS|Box has been designed and prototypes are under development within the H2020 project M2DC, therefore some parts of the following descriptions are based on M2DC documents. Due to the extensive monitoring capabilities and its heterogeneous nature (including CPU/FPGA/GPU coupling), the RECS|Box will be used for evaluation of the project’s appliances. Additionally, it will be stabilized and enhanced within this project to support dynamic reconfiguration of the communication infrastructure. Furthermore, new concepts for TCO-optimized variants of the RECS|Box will be developed based on the use case requirements within LEGaTO.

6.3.1. Hardware Architecture

As depicted in Figure 6.2, the RECS|Box server features a modular approach [63]. This modularity ensures flexibility and reusability, thereby ensuring high levels of maintainability. Microservers are grouped on carriers (or baseboard), which support hot-swapping and hot-plugging of microservers, similar to a blade-style server. There are different carriers available, e.g., one for low-power microservers supporting a population of 16 microservers, and one for high-performance microservers, supporting a population of 3 microservers. The microservers are designed based on existing computer-on-module or server-on-module form factors which are established in the industry. This eases integration of third-party microserver modules, providing a broad set of commercial of the shelf microserver modules readily available for usage in the RECS|Box server.

Within the RECS|Box, one or several microservers can be coupled among each other or with PCIe extension cards using a flexible high-speed, low-latency (HSLL) communication infrastructure, supporting raw serial links as well as PCIe-based communication (see Section 5.4.2). Two different chassis sizes are available so far, a 2 RU chassis with three slots for blade-carriers and a 3 RU chassis with nine slots for blade-carriers, see Figure 6.3. Both chassis share common parts and a blade-style approach.
Each carrier-blade is slid into the RECS|Box from the front, connecting 12 V DC power as well as several management signals, 4 x 10 Gbit/s Ethernet signals and PCIe signals to the corresponding backplane. Once a carrier-blade is inserted, it gets detected by the firmware and initialized, detecting what microservers are plugged in and what was their last state. From there on, they can be regularly used, directly and individual, or composed to a bigger virtual node on the PCIe level. Furthermore, they can be deployed bare-metal using OpenStack’s Ironic with some Linux distribution. There will be at least one working Linux version available for each microserver type which is preconfigured and brings the integration software for the microservers to be monitored from the RECS_Master as described in Chapter 6.3.4.

The RECS|Box Durin has one backplane which hosts both major communication infrastructures, 10/40 Gbit/s Ethernet as well as the PCIe based HSLL while the RECS|Box Deneb has six backplanes. Three identical ones provide all management and monitoring functionalities as well as the 10/40 Gbit/s Ethernet switching, three more realize the HSLL communication. The first ones are always necessary, the latter ones are optional and required only for HSLL communication.

All major components of the RECS|Box have been designed to be hot-swappable, namely the carrier-blades (which of course powers down all containing microservers), the fans and the redundant power supplies.

![Figure 6.2. Overview of the RECS|Box server architecture [63](image)](image)
6.3.2. Microserver

The carriers of the RECS|Box server can be equipped with different microservers. As listed in Table 6.2, the COM Express [71] form factor is used as the basis for all

|                                | RECS|Box Deneb | RECS|Box Durin |
|--------------------------------|----------|----------|
| Chassis size                   | 19-inch 3 RU | 19-inch 2 RU |
| Maximum number of low-power microservers (using Jetson form factor) | 144 | 48 |
| Maximum number of high-performance microservers (using COM Express form factor) | 27 | 9 |
| Extension carriers, each can host 1 PCIe card like GPGPU/PCIe-SSD or up to 6 2.5-inch HDDs | 3 | N/A |
| Ethernet infrastructure       | Mult. 1 + 10 Gbit/s Ethernet per microserver, ≥ 40 Gbit/s internal backbone |
| Ethernet connectors (trunkable) | 3 x QSFP+ (40 GbE) or 6 x RJ45 (10 GbE) | 1 x QSFP+ (40 GbE) or 2 x RJ45 (10 GbE) |
| High-Speed infrastructure      | 63/80 Gbit/s high-speed, low-latency (PCIe or HSSL) |
| High-Speed connectors (4 x 4 HSSL lanes per connector) | 1 x Quad iPass+ HD (SFF-8644) | 3 x Quad iPass+ HD (SFF-8644) |
| Power supply capability (shareable across chassis) | 6000 W redundant (3 + 1) | 2000 W redundant (1 + 1) |
| Management features            | • iKVM connection for each microserver  
|                                | • OLED status display  
|                                | • Management Web GUI with XML REST API and IPMI  
|                                | • Dedicated Ethernet management port |

Table 6.1. RECS|Box 4 model feature comparison
<table>
<thead>
<tr>
<th>Low-Power Microserver (Jetson/Apalis)</th>
<th>CPU</th>
<th>GPU</th>
<th>FPGA</th>
</tr>
</thead>
<tbody>
<tr>
<td>NVIDIA Tegra X2</td>
<td>2 Core Denver2 + 4 Core A57 +</td>
<td>Xilinx Zynq 7020 85 kLC</td>
<td></td>
</tr>
<tr>
<td>ARMv8 Server SoC</td>
<td>Intel Xeon E3</td>
<td>NVIDIA Tesla V100 Volta GPGPU@1.3 GHz</td>
<td></td>
</tr>
<tr>
<td>32 Core A72@2.4 GHz</td>
<td>4 Core Kaby Lake@3.0 GHz</td>
<td>Intel Stratix 10 SoC 1,092 kLC</td>
<td></td>
</tr>
<tr>
<td>Intel Xeon D</td>
<td>16 Core Broadwell@1.3 GHz</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>High-Performance Microserver (COM Express)</th>
<th>CPU</th>
<th>GPU</th>
<th>FPGA</th>
</tr>
</thead>
<tbody>
<tr>
<td>Intel Xeon E3</td>
<td>4 Core Kaby Lake@3.0 GHz</td>
<td></td>
<td></td>
</tr>
<tr>
<td>16 Core Broadwell@1.3 GHz</td>
<td>Intel Xeon D</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**Table 6.2. Selected RECS|Box microservers**

High-performance microservers, while the Jetson standard [60] from NVIDIA or the Apalis standard from Toradex [88] is used for the low-power microservers. The table lists some examples of possible modules, many others are available. The COM Express form factor supports a compact size of just 125 mm x 95 mm. COM Express Type 6 and Type 7 modules are supported, enabling direct integration of commercial off-the-shelf modules, e.g., x86 modules based on Intel’s Kabylake/Coffee lake architecture. Apart from x86-based modules (Intel Xeon, AMD Epyc), microservers integrating ARM 64-bit (Cortex-A72) and FPGAs (Intel Stratix 10 and Xilinx Virtex Ultrascale+) are available.

The high-performance ARMv8 microserver is based on an ARM 64-bit SoC, integrating 32 Cortex-A72 cores running at up to 2.4 GHz. The memory controller supports four memory channels populated with DDR4 SO-DIMMs running at 1866 MHz. In total, each microserver integrates up to 128 GByte RAM. For connectivity, the microserver provides two 10G GbE ports with RoCE support, in addition to a 1 GbE port which is mainly used for management purposes. Up to 24 high-speed serial lanes are available for connection of peripherals or for high-speed, low-latency communication to other microservers in the RECS|Box server. Using these high-speed links, also multi-socket configurations of the ARMv8 microserver are supported. Additionally, a wide variety of fixed-function units are integrated into the SoC, providing highly resource-efficient acceleration of compression/decompression or security algorithms like asymmetric encryption.

FPGAs are becoming more and more attractive in HPC and cloud computing due to their potentially very high performance combined with moderate power requirements. High-level synthesis and OpenCL support, which is becoming more and more mature, enable additional application scenarios since programming is no longer limited to hardware specialists. The FPGA-based high-performance microserver is a full featured COM Express module, comprising an Altera Stratix 10 SoC with an integrated 64 bit quad-core ARM Cortex-A53 processor. Dedicated DDR4 memory is provided for the CPU as well as for the FPGA fabric, supporting up to four memory channels and up to 64 GByte. The high-speed transceivers integrated in the FPGAs are used for PCIe interfacing to communi-
cate with other processor modules, and for low-latency high-bandwidth communication between high-performance FPGA-based microserver modules.

SoCs targeting the mobile market are promising platforms for data centers when focusing on energy efficiency, especially due to a large amount of integrated accelerators, including GPGPUs, fixed function units, e.g., for video transcoding, or even FPGAs. The RECS|Box server enables integration of modules based on the Jetson standard from NVIDIA [60], which is used for the currently available Tegra SoCs (Tegra-X1) and the upcoming generations from NVIDIA. Additionally, the Apalis standard from Toradex [88] is supported. With its small form factor of just 82 x 45 mm, it allows a very high density of microservers. In addition to commercially available Apalis modules from Toradex, modules have been developed at Bielefeld University integrating Samsung Exynos5250 SoCs and Xilinx Zynq7020, respectively.

6.3.3. Communication Infrastructure

As described above, all microservers have access to several 1G/10G Ethernet ports as well as to the high-speed, low-latency communication infrastructure (HSLL). In addition to the monitoring infrastructure described in Section 6.3.4, the Ethernet as well as the HSLL infrastructure are presented in this section.

The Ethernet-based network infrastructure features at least one 1 Gbit/s and one 10 Gbit/s network link per microserver, independently of the microserver type. Most high-performance microservers have two 10 Gbit/s links available. The Ethernet network is internally switched by a hierarchical, multilevel switching infrastructure which supports typical data center features like, e.g., VxLAN, RoCE or iWARP. All carriers are connected via 40 Gbit/s links to the Ethernet switching backbone of the backplanes. The external connectors can be 2 x 10 Gbit/s or 1 x 40 Gbit/s links per backplane. The overall number of connectors depends on the chassis size as described in Table 6.1. The resulting upstream bandwidth towards the top of the rack (ToR) switch is up to 120 Gbit/s, combining three 40 Gbit/s links.

The Ethernet-based network is one of the main communication infrastructures within the RECS|Box, used for several communication links:

- Between the microcontrollers on the backplanes and the carrier blades/baseboards for low-level communication like controlling and monitoring of attached sensors and microservers
- Between the backplanes/baseboards microcontrollers and the chassis controller (respectively the RECS_Master that actually runs on one of the backplane microcontrollers)
- Providing a dedicated external RJ45 Ethernet port at the back side of the chassis as the server management port
- Connecting the available 1/10 Gbit ports of all microservers between each other and to the outside of the server

Within the RECS|Box every baseboard (there is one respectively two in each carrier-blade) has an integrated Ethernet switch, just as every backplane. These multiple Ethernet switches are configured through a PCIe management interface
which is accomplished through a combination of a specialized switch management software and the RECS_Master. To provide straightforward management interface to the user, support for the management of multiple switches has been added to the RECS_Master, hiding the complexity of the multi-node chassis with many internal switches from the user. This is done by showing a microserver-centric view to the user. Through this view, the user is able to e.g., configure that a specific microserver should be available in a certain VLAN and that this VLAN should be reachable from a specific external 10 GbE RJ45 connector. The RECS_Master then configures all switches that are part of the network path between the microserver and the external connector to route the VLAN accordingly.

The RECS|Box HSSL communication infrastructure is based on a dedicated high-speed low-latency communication network. It connects to the CPU-/GPU-based microservers via PCIe and to the FPGA-based microservers via their high-speed serial interfaces. Depending on the involved communication partners, it features either Host-2-Host PCIe-based packet switching or direct, circuit-switched switching. In addition to direct communication between the different microservers, it also supports connection to storage or I/O-extensions. This possibility allows easy integration of PCIe-based extension cards like GPGPUs or storage subsystems, which can also be shared across multiple microservers via the multi-root I/O virtualization (MR-IOV) feature of the PCIe communication infrastructure.

In contrast to the majority of today’s implementations, in which a hardware accelerator is typically attached physically to the PCIe lanes of a CPU node, the RECS|Box HSSL communication infrastructure can be used not only to connect CPUs to hardware accelerators but also CPUs to CPUs or accelerators to other accelerators. Furthermore, it is possible to divide the SEE link of a certain CPU/accelerator, e.g., connecting an accelerator to both a CPU and another accelerator. Thus, accelerators can be combined into one large virtual SEE unit. At run-time, the communication topology can be reconfigured and adapted to changing application requirements.

The high-speed, low-latency communication infrastructure allows a number of different application scenarios, which are illustrated in Figure 6.4 to Figure 6.6. The communication infrastructure could be used to connect any microserver of a baseboard to a PCIe-based peripheral. This allows attaching accelerators like GPGPUs to a microserver, as shown in Figure 6.4(left). While all microservers of a baseboard are supported with PCIe x8, one microserver slot per baseboard also

---

**Figure 6.4. PCIe communication within the RECS|Box Server**

please provide the necessary details for the diagram described in the text.
supports PCIe x16 for maximum bandwidth.

Apart from just attaching a PCIe peripheral to just one microserver, the PCIe infrastructure supports also multi-root scenarios as depicted in Figure 6.4 (middle). If this feature is supported by the peripheral, this can be a useful feature to create MR-IOV (Multi-Root I/O virtualization) setups, which are commonly used, e.g., for PCIe-based SSD storage systems.

The PCIe communication infrastructure integrated into the RECS\Box server also support direct Host-2-Host communication, as shown in Figure 6.4(right). In contrast to setups using the traditional non-transparent endpoint feature, which typically leads to limited burst-lengths as well as limited scalability, the RECS\Box server uses PCIe switches that directly support scalable Multi-Host communication. This results in a packet-based network style communication that can be used like Ethernet or Infiniband. Two main communication modes are supported, either DMA-based data transfer that can be used like RDMA over Converged Ethernet (RoCE), or low latency message passing, supporting MPI style communication.

Apart from communication based on PCIe, the RECS\Box server communication infrastructure also supports direct links between microservers, which are switched based on asynchronous crosspoint switches, independent from the used protocol. This feature is of particular interest for FPGAs, as these devices support low-level point-to-point communication schemes not involving the overhead of protocols like PCIe. Example use cases are shown in Figure 6.5, which depicts a direct communication infrastructure between the FPGAs as well as multiple PCIe endpoints implemented on the FPGA, used for communication towards the processor-based (e.g., ARM\textsuperscript{v8}) microservers. As shown in Figure 6.6, such a scheme could also be realized using PCIe-based Host-2-Host communication, leading to additional advantages such as network-based communication, as already discussed before.

**6.3.4. Management and Monitoring**

This section gives an overview of the different management/monitoring mechanisms of the RECS\Box. It covers mainly the low-level software (firmware) part, while the higher levels are described in Section 5.4. Although the higher levels of the firmware and management software are described elsewhere, the sub-

![Figure 6.5. PCIe communication combined with direct links between the FPGAs](image-url)
Figure 6.6. Direct communication between FPGA and Host-2-Host PCIe communication

Substantial development will be done within the hardware Work Package.

As described above, each RECS|Box contains different baseboards and at least one backplane. Each baseboard and backplane is equipped with a quite powerful microcontroller running Linux as a base system and, in addition, some low-level and firmware components, if required by the respective baseboard/backplane component.

The baseboard firmware is a custom Linux program that communicates with the management software and controls all peripherals on the baseboard. It encapsulates the basic Kernel interfaces (e.g., for GPIOs, ADCs, and I²C) into objects and then realizes the management features on top of these. As this low-level hardware access will largely be the same between different baseboards and also the backplane, there will only be one firmware program that, during runtime, determines on which board it runs and which tasks are to be performed. Using C++ as programming language and object-oriented design makes this adoption easily possible.

On one of the backplanes, a central management software runs that takes control and monitors all further components, the RECS_Master. It is responsible for all low-level management functionalities but also hosts the external software APIs and interfaces as described in Section 5.4. It offers detailed monitoring information of the hardware itself like power usage and temperatures of individual components, as well as the possibility to control all microservers with respect to their power state, boot mode etc. Additionally, it manages the Ethernet and high-speed, low-latency communication infrastructure. The latter allows the connection of resources like microservers and PCIe devices with each other in a very flexible manner. This mechanism, called node composition, is described in Section 5.4.2 as it is directly used by LEGaTO’s OpenStack middleware.

The capabilities of the RECS_Master are accessible through different interface. For regular manual administrative usage a WebGUI[18] is provided that allows access to all user-changeable settings and all monitoring and control functionality. In addition to that, a Nagios Remote Plugin Executor (NRPE) interface for integration with monitoring and alerting software as well as an IPMI interface are offered by the RECS_Master. Furthermore, there is a self-defined RESTful API[19],

---

**Document Information**

**Contract Number**: 780681

**Project Website**: https://legato-project.eu/

**Contractual Deadline**: [To be filled]

**Dissemination Level**: [To be filled]

**Nature**: [To be filled]

**Author**: [To be filled]

**Contributors**: [To be filled]

**Reviewers**: [To be filled]

---

The LEGaTO project has received funding from the European Union’s Horizon 2020 research and innovation programme under the Grant Agreement No 780681.
which allows retrieval of all measured sensor values and basic control of the nodes. Finally, the extended Redfish API is available for managing certain special functions like node composition, described in Section 5.4.1. The Redfish API is utilized by the OpenStack Valence component to compose heterogeneous resources to more complex entities, whose communication infrastructure can be changed directly through the Redfish API during runtime.

**Oscilloscope Mode**

The monitoring system of the RECS|Box server normally works with a sensor update rate in the order of 1 Hz. While this is certainly fast enough for monitoring the health status of the system, it does not allow characterizing the energy consumption of computational kernels running on the microservers. To be able to do this a much higher sampling frequency for certain signals in a given timeframe is realized via a so-called 'Oscilloscope mode': When it is enabled by the user, the baseboard hosting the selected microserver samples the respective ADC channels with the maximum available sample rate. Instead of just computing average, minimum and maximum values of the data, it is completely stored in a ring buffer where it can later be retrieved for analysis. The sample rate available to the user is hereby increased to up to 1 MSPS.

Oscilloscope mode also allows setting up triggering either on rising or falling edge (e.g., rising current consumption) with a settable threshold or manual triggering, e.g., before running the computational kernel. The user can also define the split between pre- and post-trigger data that should be stored (effectively setting the trigger position)[37].

**6.3.5. TCO Optimization**

One major goal of the LEGaTO project is to enable users to optimize their applications towards heterogeneous hardware and thus energy efficiency. Within WP2, we try to optimize the (already existing or newly developed) hardware towards the applications, so we have the same goal but come a different way. Optimizing the hardware has always many possible goals: one could optimize a given hardware for performance, for flexibility, for heterogeneity, for space efficiency or for TCO. We chose the latter one as it combines almost all other possible metrics in a certain way.

To come up with a reasonable definition of a hardware optimization, we chose a typical hardware scenario which could fulfill some of the application needs, within or outside LEGaTO, but is not tailored to one specific as it would limit the broad usage of the hardware. The scenario is limited to x86 hardware for easy comparison to commercially available 3rd party solutions. But as one of the main advantages of the RECS|Box hardware is its flexibility, it should be taken into account that this comparison is just one of many possible scenarios. The RECS|Box platform would most likely outmatch other solutions even more in heterogeneous scenarios, e.g., when incorporating GPUs, FPGAs, or ARM-based hardware. The chosen scenario contains an x86 server with 8 cores at approximately 2 GHz, 32 GB memory, 2 x 10 Gbit/s Ethernet and a local 256 GB SSD for OS bootup. For the LEGaTO project, such a server can be packed as a COM Express microserver into a RECS|Box, 9 into a RECS|Box Durin and 27 into a RECS|Box Deneb. Of course the RECS|Box is quite costly, on the one hand because it is built
<table>
<thead>
<tr>
<th>Server type</th>
<th>RECS</th>
<th>Box Deneb</th>
<th>RECS</th>
<th>Box Deneb</th>
<th>Intel Quad</th>
<th>Intel Std</th>
</tr>
</thead>
<tbody>
<tr>
<td>Switching description</td>
<td>internal</td>
<td>internal</td>
<td>internal</td>
<td>external std. switches</td>
<td>external std. switches</td>
<td></td>
</tr>
<tr>
<td>RU per server</td>
<td>3</td>
<td>3</td>
<td>2</td>
<td>2</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>Nodes per server</td>
<td>27</td>
<td>27</td>
<td>9</td>
<td>4</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>Costs per chassis</td>
<td>61,061 €</td>
<td>51,111 €</td>
<td>21,760 €</td>
<td>11,754 €</td>
<td>2,219 €</td>
<td></td>
</tr>
<tr>
<td>Costs per node</td>
<td>2,280 €</td>
<td>1,993 €</td>
<td>2,415 €</td>
<td>2,938 €</td>
<td>2,219 €</td>
<td></td>
</tr>
</tbody>
</table>

**Switch costs**

Switch port costs per chassis | 713 € | 713 € | 238 € | 2,401 € | 292 € |

Costs per chassis incl. switch ports | 61,754 € | 51,834 € | 21,978 € | 14,155 € | 2,511 € |

Costs per node incl. switch ports | 2,287 € | 1,991 € | 2,422 € | 3,539 € | 2,511 € |

### 42 RU rack

| Chassis per rack | (13) | (13) | 20 (18) | 15 | 29 |
| Switches RU per rack | 1 | 1 | 1 | 12 | 3 |
| Nodes per rack | 351 (162) | 351 (162) | 180 (162) | 60 | 39 |
| Cores per rack | 2802 (1098) | 2808 (1122) | 1460 (1199) | 960 | 312 |
| Nodes per RU | 8.8 (8.8) | 8.8 (8.8) | 4.4 | 1.8 | 0.9 |
| Cores per RU | 70.2 (68.2) | 70.2 (68.2) | 35.1 (35.0) | 22.9 | 7.4 |
| Costs chassis | 793,533 € | 664,443 € | 434,800 € | 176,310 € | 86,541 € |
| Costs switches | 7,610 € | 7,600 € | 7,600 € | 39,792 € | 14,013 € |
| Total costs fully equipped rack | 8,017,373 € | 672,043 € (373,846 €) | 420,200 € (309,976 €) | 216,102 € | 100,055 € |

### Power usage

| Max power usage per chassis | 2,241 W | 2,241 W | 871 W | 700 W | 151 W |
| Max power usage servers per rack | 31,473 W (14,526 W) | 29,733 W (16,487 W) | 17,560 W (17,786 W) | 10,500 W | 5,889 W |
| Max power usage switches per rack | 310 W | 310 W | 710 W | 1,104 W | 756 W |
| Total max power usage kW/Rack | 32 kW (15 kW) | 29 kW (16 kW) | 18 kW (16 kW) | 12,0 kW | 7,0 kW |

### Data Centre

| # racks for 1000 servers | 3 (1) | 3 (1) | 6 (1) | 17 | 26 |
| DC size (m²), using 3.5 m² per rack | 20,5 (34.5) | 20,5 (34.5) | 31,0 (34.5) | 69.5 | 101 |
| Heat load (kW/m²) | 4.4 (2.2) | 4.3 (2.2) | 3.2 (2.3) | 2.8 | 1.7 |
| Total heat load (kW) | 91 | 84 | 99 | 193 | 170 |
| Construction costs €/m² for above heat load | 11,000 € (9,100 €) | 11,000 € (9,100 €) | 10,200 € (9,900 €) | 9,800 € | 8,200 € |

### CAPEX

| Total construction costs DC | 233,700 € (334,600 €) | 227,730 € (309,700 €) | 316,200 € (341,950 €) | 681,500 € | 818,200 € |
| 1000 servers + switches | 2,282,000 € (2,303,000 €) | 1,916,000 € (1,973,000 €) | 2,458,000 € (2,526,000 €) | 3,602,000 € | 2,518,000 € |
| Total costs for DC + 1000 servers + switches | 2,516,000 € (2,643,000 €) | 2,262,000 € (2,474,000 €) | 2,776,000 € (2,884,000 €) | 4,283,000 € | 3,459,000 € |

### OPEX for PsE: 15, €/kWh: 0.14, a load on servers: 0.6

| Power costs / year | 99,905 € (101,080 €) | 92,239 € (93,420 €) | 109,460 € (109,670 €) | 273,470 € | 182,210 € |
| Total power costs for 5 years | 500,000 € (505,000 €) | 483,000 € (485,000 €) | 547,000 € (548,000 €) | 1,067,000 € | 936,000 € |

| TCO for a DC, 1000 servers, switches and power for 5 years | 3,016,000 € (3,116,000 €) | 2,605,000 € (2,701,000 €) | 3,221,000 € (3,332,000 €) | 5,350,000 € | 4,293,000 € |

**Table 6.3. TCO calculation of different server types**
in small (prototype) quantities, and is not produced in larger volumes yet. On the other hand as it provides many unique features not available in a standard x86 server, e.g., the PCIe based communication (high-speed low-latency, HSLL) infrastructure, support for a wide range of heterogeneous compute modules as well as a flexible Ethernet communication infrastructure, e.g., the possibility to use COM Express Type 6 and Type 7 modules, including separate 10 Gbit/s Ethernet adaptors for Type 6 modules. Removing those features limits the universal approach of the RECS|Box, however, also lowers the hardware costs and therefore improves the TCO. Such optimizations leads to a redesign of the COM Express carrier blade (used for HP microservers) for the RECS|Box chassis. Nevertheless, due to the modular design of the RECS|Box server, the full-featured version of the HP microserver can still be deployed if required, as both versions of the carrier blade are supported by the hardware and firmware architecture.

An estimation of TCO of the RECS|Box compared to alternative approaches has been calculated as seen in the condensed Table 6.3. It provides a first general estimate that will be updated within the project based on specific use cases. The table compares the purchasing costs of standard servers with the production costs of RECS|Box servers, which makes it only applicable for project partners having access to RECS|Box servers at the production costs. For an customer view, a profit margin would have to be added, making the advantages of the RECS|Box servers smaller, but still significant, especially if comparing the TCO values that include the Data Centre and power costs. The design of the RECS|Box server is optimized for hyperscale data centers targeting a high compute density, but also required quite high power and cooling requirements per rack. These power and cooling requirements might not be supported by every data center, therefore, the RECS|Box Deneb and Durin compositions in Table 6.3 contain additional values in brackets for a comparison within an data center environment with a limit of 16 kW per rack.

The RECS|Box Deneb and RECS|Box Durin represent the full-featured version of the RECS|Box servers. As explained before, the idea for the RECS|Box Deneb TCO optimized is to have a modified COM Express carrier-blade and not equipping the HSLL backplane. This will remove all high-speed, low-latency communication features, e.g., the usage of PCIe for host-to-host communication or direct FPGA communication. Still, two neighboring microservers could be directly coupled and using PCIe extension cards will also be possible. Intel Quad is a 2 RU server that has 4 nodes, each dual-CPU ready, but only equipped with one CPU. Although this server is more compact than the typical 1 RU “pizza box” Intel Std (as it can be seen in the line # racks for 1000 servers, it costs more because of the unused resources of the 2nd CPU slot.

When looking at TCO, the biggest difference is between the RECS|Box Deneb TCO optimized and the Intel Quad which cost about twice as much in total. The more obvious choice would maybe be the Intel Std servers which still are 68 % (62 % for a datacenter limited to 16 kW per rack) more expensive than the RECS|Box Deneb TCO optimized.

A direct comparison between the non-optimized RECS|Box Deneb and the TCO optimized version shows CAPEX savings of about 15 % while OPEX savings are about 7.5 % due to reduce energy usage overhead of the removed chips and in-
frastructure. As these savings result from just replacing a single component of the modular RECS|Box architecture, this is a viable option and could be considered during the LEGaTO project for applications matching the reduced feature set. As most of the applications are not yet fully taskified and analyzed, further application analysis is required in order to determine the efficiency of the suggested TCO optimization. One example of a potential use case that would profit from the TCO optimized RECS|Box version could be, e.g., the Secure IoT Gateway (see Section 3.7), using the RECS|Box as Gateway Cluster and/or Network Gateway, replacing the virtualized setup by bare-metal installations on the microservers.

6.4. Edge Server Architecture

Apart from optimizations of the architecture of the cloud platform used in LEGaTO, the development of an edge server platform is foreseen within the project. While the cloud hardware is focused on the data center, the edge server architecture supports applications with local (pre)processing requirement, e.g., for data-reduction, direct user interaction, faster response time or safety/security/dependability reasons. Main application use cases within LEGaTO so far include smart home (see Section 3.3) and machine learning (see Section 3.5), with possible extensions to smart city (see Section 3.4) and IoT gateway (see Section 3.7).

The general architecture requirements include a modular hardware design and support for all three major compute architectures, i.e., CPU, GPU, and FPGA. It should be based on the existing microservers and form factors, which are also present in the RECS|Box architecture. In addition to the existing form factors, also new developments like Jetson AGX Xavier from NVIDIA or new PICMG standards are considered for integration. The overall idea is to support a distributed setup, i.e., a reasonably fast, low-latency communication link between the different units is mandatory. In addition, it should be possible to seamlessly interface to common peripherals like cameras, displays, LIDAR/RADAR or microphones. The whole unit shall support rich monitoring as well as a small form factor, allowing deployment in an embedded, rough environment.

The following items explain the architecture of the proposed edge server platform, which is shown in Figure 6.7, based on the features of the architecture. It also correlates the requirement from the different applications like smart home or machine learning with these features to judge the different design decisions.

- **Support for all architectures:** The platform shall support all major compute architectures, namely CPU, GPU, and FPGA. As for the RECS|Box platform, low-power, as well as high-performance, compute units should be supported. Furthermore, different variants of a particular compute architecture, like x86 vs. ARMv8 for CPU, or Xilinx vs. Intel for FPGA, should be possible. Having such a wide variety of architectures allows to choose the best platform for a given task or application, e.g., image processing required in machine learning or smart home applications performs best on FPGA, while the used deep learning algorithms are easily implemented on a GPU. More specialized neural networks could be again more suitable for implementation on an FPGA, especially concerning energy efficiency.
Using the computer-on-module standards which are already used in the RECS|Box server allows easy reuse of the already existing microservers in both low-power and high-performance variants. In addition to the supported microservers, a PCIe Gen.3 x16 Slot is also available in the architecture, supporting the full range of available PCIe peripherals. The PCIe device is not permanently assigned to a specific microserver; it can be flexibly assigned to any microserver at run-time, and also shared between the different microservers, which is particularly useful for storage and network devices. The platform itself integrates an NVMe-SSD and a dual-port 10G Ethernet NIC, which can both be shared across the different microservers.

- **Modular, scalable design:** The platform shall support a modular approach. Both types of microservers, low-power, and high-performance ones shall be supported. The platform shall be designed to support scalability, which is of particular importance for a number of applications, especially the machine learning application, which targets autonomous driving, but also for the smart home application. Furthermore, the platform should support multiple low-power microservers on a single edge platform, in order to support scalability not only for multiple edge servers but in a single one as well. By integration of up to three low-power and one high-performance microservers, the proposed edge server fulfills these requirements.

- **Distributed architecture, low-latency communication:** In order to support distributed setups, the edge server shall integrate a fast, low-latency communication infrastructure, which does not only support communication between the different microservers on one edge server platform but also between different edge server units. This is of particular importance for the autonomous driving use case in the machine learning application.

The proposed communication infrastructure of the edge server architec-
ture is based on two main pillars, on the one hand, the Ethernet-based communication infrastructure, and on the other hand the PCIe-based communication infrastructure. Both support communication between the different microservers on one edge server platform as well as between multiple edge servers. The main difference between them is the communication latency, which is about one order of magnitude lower for PCIe. The Ethernet communication infrastructure provides multiple 1/10 Gbit/s ports to all microservers, low-power microservers can access the 10 Gbit/s NIC via the PCIe infrastructure. The proposed edge server design provides eight external Ethernet ports, sufficient to support larger installations without the need of an external switch using a mesh-style topology, which is useful in large distributed edge server use cases. The PCIe-based communication infrastructure provides two external ports providing the possibility to create a ring-style communication topology.

- **Peripheral support:** As an edge server platform, easy interfacing to peripherals is vital. The different microservers offer various interfaces like USB, CSI or even arbitrary direct interfacing in case of an FPGA-based microserver. This can be used to seamlessly interface to common peripherals like cameras, displays, LIDAR/RADAR or microphones. Due to its embedded nature, the LP microserver form factors support a wide range of peripheral interfaces. Integrating multiple LP microservers in one edge server gives the possibility to combine multiple sensors in one system, e.g., multiple cameras or camera and LIDAR sensors. All peripherals interfaces capable of supporting sensors or actuators are connected straight to external connectors of the edge server.

- **Monitoring/Management:** In order to manage the edge server platform, for housekeeping as well as easy integration into different environments, it is mandatory to include a management interface into the server architecture. Apart from simple management, the proposed solution also allows continuous monitoring of the different temperatures as well as voltage and current rails of the system. An integrated iKVM solution enables remote KVM access to every microserver in the system. As an option, a special, fast sampling mode can be integrated which allows to use the current monitoring to capture fast changes in power consumption, like in an oscilloscope connected to a current shunt, which gives to possibility to perform benchmarking, profiling and characterization of an application towards power or energy consumption in a fine-grained manner.

- **Compact form factor:** The system shall be realized in a small form factor, allowing deployment in an embedded, rough environment. In the current proposal, the system has a size of about a full-size PCIe card. This is sufficient space to accommodate the mentioned PCIe card, as well as three LP and one HP microserver, along with the required communication, management and power infrastructure. The form factor matches the outline of 1/3 of a standard 19 inch rack. It is powered from 12 V, making it possible to be powered in embedded as well as standard environments.

A cost-optimized version of the edge server architecture is shown in Figure 6.8. It does not fully match to the requirements discussed before, however, as the
TCO-optimized version of the RECS|Box cloud server (see Section 6.3.5), it might be sufficient for some use cases. Instead of three LP Microserver, it supports just one LP Microserver. Furthermore, the communication infrastructure is reduced significantly, as it has no PCIe communication infrastructure and no integrated 10G Ethernet switching. It does, however, still support all architectures in their LP and HP variants, as well as a PCIe peripheral, comparable to the full featured version depicted in Figure 6.7. Furthermore, both version integrate a comparable management and monitoring infrastructure.

### 6.5. Maxeler Dataflow Engines and Systems

Maxeler Dataflow Engines (DFEs) are PCIe cards that combine a large reconfigurable device with large amounts of DDR memory. The high-level architecture of a DFE is illustrated in figure 6.9. The current MAX5 DFE generation uses a Xilinx UltraScale+ VU9P FPGA device to provide the reconfigurable substrate for the
dataflow computations. FPGAs contain programmable logic resources in form of general-purpose logic look-up tables, programmable interconnect, on-chip memory and programmable DSPs. On-chip memories are exposed to the developer as Fast Memory (FMem) as they can be accessed with an aggregated bandwidth of several tens of terabytes/second. For large-scale storage the DFE provides 48-96 GB DDR4 DRAM which is called Large Memory (LMem). The DFE also provides a direct card-to-card communication link called MaxRing as well as a 100Gb Ethernet port.

DFEs can be integrated into a range of different system configurations. In Maxeler MPC-C series systems, up to 4 DFEs are integrated into a high-end dual-socket server with Intel Xeon or AMD EPYC CPUs, creating a heterogeneous high-performance computing system. The architecture is shown illustrated in figure 6.10. An MPC-C node can be 1 or 2U in height, depending on the host server model that is used. The MPC-C model can also be customised in RAM, HDD or SSD storage, and networking options.

However, a far more flexible system architecture is illustrated in figure 6.11. In Maxeler MPC-X series systems, 8 DFE cards are incorporated into a dense, industry standard 1U chassis, creating a pure dataflow appliance. This system is connected to conventional CPU servers via an Infiniband network. Inside the MPC-X, the DFEs are also directly connected through MaxRing. Such a system can dynamically allocate large numbers of DFEs to multiple applications, providing high scalability and flexibility for large-scale systems.

An MPC-X based pilot system equipped with the latest generation MAX5 DFEs is available at the Jülich Supercomputing Center in Germany. It was delivered as part of the PRACE PCP project. The system consists of a Supermicro server equipped with 2x AMD 7601 EPYC CPUs, 1 TB RAM, 9.6 TB SSD data storage, 10Gb Ethernet and Infiniband. Over Infiniband, this server connects to a Maxeler MPC-X 3000 with 8x MAX5 DFEs. The system also contains an additional head and login node. This system is available to research partners on request.

Maxeler MAX5 DFEs are fully compatible with Amazon EC2 F1 instances meaning that applications developed for this latest DFE generation can also run on the AWS Cloud. As explained in Section 4.3.6, MaxCompiler supports compiling application for Amazon EC2 F1 instances. The Amazon Cloud can therefore also be

![Figure 6.10. Architecture of an MPC-C series system. DFEs are directly integrated into a server.](image-url)
used to run applications that explore MaxJ programming techniques.

**6.6. Conclusion**

The hardware architecture of LEGaTO, defined, specified and implemented within WP2, has been detailed in this chapter. The LEGaTO hardware enables the deployment of the LEGaTO stack for cloud as well as edge use cases. Two cloud platforms are used in LEGaTO, the RECS|Box (see Section 6.3) as well as Maxeler’s DFE (see Section 6.5). In addition, an edge platform (see Section 6.4) is developed with the project. The platforms support the full range of heterogeneous microserver technology from CPUs to FPGAs, which can be seamlessly combined to provide the optimal platform for a given use case or application. The integrated high-speed, low-latency communication infrastructure, described in Section 6.3.3 provides another powerful optimization option. The dynamic node composition feature, which is controlled by the management functionality in combination with the Refish and Openstack middleware described in Section 5.4, allows to adapt the communication topology at run-time, dynamically adapting to changing application requirements.

---

*Figure 6.11. Architecture of an MPC-X series system with 8 DFEs. Multiple MPC-X nodes and CPU nodes are connected through an Infiniband network.*
7. SD1 Conclusion

Superdeliverable SD1 puts together all technical deliverables of month 9 in Work Packages 2 to 5, in order to offer a homogeneous and integrated presentation of the results of the design phase of the project. The joint preparation of the deliverable allowed for more internal exchanges between partners, improving the quality of the document, increasing the overall understanding of the components by all partners and, finally, allowing to reduce the efforts for integration in the upcoming months.

Work Package 2 goals for the month 9 deliverable were the specification and optimization of the LEGaTO hardware platform, which has been described extensively in this document. Additionally, another important Work Package 2 goal was to start to deploy the LEGaTO hardware platform very early in the project timeline so that the use cases (Work Package 5) and the energy-efficient programming environment (Work Package 4 and Work Package 3) could be integrated, tested and optimized in this platform. Towards these goals, we have concluded the definition of LEGaTO hardware architectures, drafted a proposal for possible TCO optimizations of RECS|Box components and started to deploy the selected hardware. Work has begun on testbed deployment and maintenance, design of edge server architecture, as well as node composition and related aspects on the high-speed low-latency communication infrastructure.

Work Package 3 has defined a back-end runtime system for LEGaTO, in which multiple heterogeneous devices and multiple resource management libraries will be targeted using task offloading. The next steps will consist of implementing the LEGaTO runtime and developing scheduling technologies targeting low energy consumption. In addition, we have defined tools for improving the performance and correctness of LEGaTO applications. These tools will provide online monitoring of application execution and provide access to debugging facilities across the heterogeneous hardware stack. We have also defined functionality to support fault tolerance. While such functionality has been researched extensively in the context of CPUs, LEGaTO will implement checkpointing functionality for FPGAs and GPUs, as well as task replication. Finally, we have defined a novel scheme in which communication topologies are to be defined at the application level. These software topologies will be used to drive the reconfiguration at the hardware level.

In Work Package 4, we presented OmpSs as the main programming model for LEGaTO, and we proposed a number of new types of annotations for mapping, fault-tolerance, security and reconfiguration, as well as programming support for applications using XiTAO, DFiant and Maxeler extensions. We proposed several extensions for the front-end side of the runtime system supporting the execution of applications written with OmpSs, as efficient support for irregular data structures, directive-based checkpointing, static kernel identification and trusted tasks. We also designed techniques for compiler-based protection against security threats and protection against software faults. Checkpointing techniques are leveraged to allow for dynamic node addition and migration during execution. To improve application programmability, we proposed a set of
plug-ins for Eclipse IDE, providing directive hints and execution and debugging support. The toolchain proposed also improves OmpSs with a distributed memory version, supporting dynamic scheduling of tasks across multiple computing nodes.

In Work Package 5, we have analyzed all the LEGaTO use cases. This includes the code and data structure analysis of each application, as well as identification and characterization of computational characteristics of the applications. Based on these analyses, we developed optimization and evaluation plans for each application including measurable metrics. Work Package 5 will use the result of this analysis and pathfinding stage for all applications to drive the application integration stage for integrating the LEGaTO software toolset with the LEGaTO hardware.

Table 7.1. Summary of LEGaTO’s components contributions to the project’s objectives

<table>
<thead>
<tr>
<th>Component</th>
<th>Energy-Efficiency</th>
<th>Security</th>
<th>Reliability</th>
<th>Programmability</th>
<th>Related WPs</th>
</tr>
</thead>
<tbody>
<tr>
<td>Undervolting</td>
<td></td>
<td>X</td>
<td></td>
<td></td>
<td>WP2, WP3</td>
</tr>
<tr>
<td>GPU-FPGA checkpointing</td>
<td></td>
<td>X</td>
<td></td>
<td></td>
<td>WP3, WP4</td>
</tr>
<tr>
<td>OmpSs annotations</td>
<td></td>
<td>X</td>
<td>X</td>
<td></td>
<td>WP3, WP4</td>
</tr>
<tr>
<td>OmpSs-2 cluster runtime</td>
<td></td>
<td>X</td>
<td></td>
<td></td>
<td>WP3</td>
</tr>
<tr>
<td>XiTAO runtime</td>
<td></td>
<td>X</td>
<td></td>
<td></td>
<td>WP3</td>
</tr>
<tr>
<td>Dynamic reconfiguration</td>
<td></td>
<td>X</td>
<td></td>
<td></td>
<td>WP2, WP3</td>
</tr>
<tr>
<td>Hardware &amp; software topologies</td>
<td></td>
<td>X</td>
<td></td>
<td></td>
<td>WP3</td>
</tr>
<tr>
<td>Offloading CPU-FPGA-GPU LEGaTO runtime</td>
<td></td>
<td>X</td>
<td></td>
<td></td>
<td>WP3, WP4</td>
</tr>
<tr>
<td>DFiant language</td>
<td></td>
<td>X</td>
<td></td>
<td></td>
<td>WP4</td>
</tr>
<tr>
<td>MaxJ compiler</td>
<td></td>
<td>X</td>
<td></td>
<td></td>
<td>WP4</td>
</tr>
<tr>
<td>Secure IoT gateway</td>
<td></td>
<td>X</td>
<td></td>
<td></td>
<td>WP5</td>
</tr>
<tr>
<td>Integrated development environment</td>
<td></td>
<td>X</td>
<td></td>
<td></td>
<td>WP4</td>
</tr>
<tr>
<td>Performance and debug tools</td>
<td></td>
<td>X</td>
<td></td>
<td></td>
<td>WP3</td>
</tr>
</tbody>
</table>

Superdeliverable 1 completes the definition of the architecture, the application requirements, and the optimization goals for LEGaTO. Table 7.1 summarises how all components in the architecture contribute to LEGaTO’s objectives. The project enters now a phase devoted to implementation and integration. As seen in the technical sections of this deliverable, the work requires substantial cross-WP efforts. We foresee this to continue until the end of the project. In addition to the project timeline in the DoW, we also established a more tight timeline internally. According to this timeline, we plan to release OmpSs@FPGA, OmpSs-
CLuster, XiTao, Maxcompiler and Dfiant at Month 20, integrate OmpSs with XiTao, Maxcompiler and Dfiant at Month 24, apply runtime reconfigurability and OmpSs Cluster on ARM/X86 on Month 30, and finally integrate LEGaTO HW, programming model and use cases by Month 36.
8. References


[9] Valeria Bartsch, Carsten Lojewski, Antoniu Pop, Paul Carpenter, Babis Chalios, Antonio J. Peña, Kyunghun Kim, Andrea Bartolini, and Francesco Conti. ExaNoDe D3.2, Runtime systems (OmpSs, OpenStream) and communication libraries (GPI, MPI): Advanced implementation customized for ExaNoDe architecture, interconnect, operating system, September 2017.


OData. OData - the Best Way to REST. http://www.odata.org/. (Online; Last access: 22.05.2017).


OPNsense. OPNsense Hardware requirements. https://wiki.opnsense.org/manual/hardware.html#hardware-requirements. (Online; Last access: 01.08.2018).


