PUCRS

ESCOLA POLITÉCNICA
PROGRAMA DE PÓS-GRADUAÇÃO EM CIÊNCIA DA COMPUTAÇÃO
MESTRADO EM CIÊNCIA DA COMPUTAÇÃO

ANDERSON ROBERTO PINHEIRO DOMINGUES

ORCA: A SELF-ADAPTIVE, MULTIPROCESSOR SYSTEM-ON-CHIP PLATFORM

Porto Alegre
2020

PÓS-GRADUAÇÃO - *STRICTO SENSU*

Pontifícia Universidade Católica
do Rio Grande do Sul

PONTIFICAL CATHOLIC UNIVERSITY OF RIO GRANDE DO SUL
SCHOOL OF TECHNOLOGY
COMPUTER SCIENCE GRADUATE PROGRAM

ORCA: A SELF-ADAPTIVE, MULTIPROCESSOR SYSTEM-ON-CHIP PLATFORM

ANDERSON ROBERTO PINHEIRO DOMINGUES

Master's Thesis submitted to the Pontifical Catholic University of Rio Grande do Sul in partial fulfillment of the requirements for the degree of Master in Computer Science.

Advisor: Prof. Dr. Alexandre de Morais Amory

Porto Alegre
2020

Ficha Catalográfica

D671o Domingues, Anderson Roberto Pinheiro

ORCA, A Self-Adaptive, Multiprocessor, System-on-Chip Platform / Anderson Roberto Pinheiro Domingues . – 2020.
110 p.
Dissertação (Mestrado) – Programa de Pós-Graduação em Ciência da Computação, PUCRS.

Orientador: Prof. Dr. Alexandre de Morais Amory.

1. multiprocessor system-on-chip. 2. self-adaptive systems. 3. computing system simulation. I. Amory, Alexandre de Morais. II. Título.

Elaborada pelo Sistema de Geração Automática de Ficha Catalográfica da PUCRS
com os dados fornecidos pelo(a) autor(a).
Bibliotecária responsável: Clarissa Jesinska Selbach CRB-10/2051

Anderson Roberto Pinheiro Domingues

ORCA, A Self-Adaptive, Multiprocessor System-on-Chip Platform

This Master Thesis/Doctoral Thesis has been submitted in partial fulfillment of the requirements for the degree of Doctor/Master of Computer Science, of the Graduate Program in Computer Science, School of Technology of the Pontifícia Universidade Católica do Rio Grande do Sul.

Sanctioned on February 28, 2020.

COMMITTEE MEMBERS:

Prof. Dr. Antonio Carlos Schneider Beck Filho (PGCC/UFRGS)

Prof. Dr. Cesar Augusto Missio Marcon (PPGCC/PUCRS)

Dr. Alexandre de Morais Amory (PPGCC/PUCRS - Advisor)

“Philosophy is a battle against the bewitchment of our intelligence by means of language.”
(Ludwig Wittgenstein, philosopher)

ORCA: A SELF-ADAPTIVE, MULTIPROCESSOR SYSTEM-ON-CHIP PLATFORM

ABSTRACT

The complex organization of multiprocessor systems-on-chips (MPSoCs) requires the smart management of systems' resources during runtime and the capability of systems to optimize their operation under abnormal situations such as temporary hardware unavailability. One of the approaches for resource management in MPSoCs is self-adaptation, which augment these systems with sensors, actuators, and decision logic components. In this thesis, we propose ORCA, a development platform to aid in designing self-adaptive systems. The platform provides abstractions to deal with self-adaptation complexity, based on previous work in the field, including a configurable hardware architecture, operating system, software libraries, and simulation environment. The hardware architecture consists of open-source hardware modules and implements a many-core approach based on a RISC-V compatible processor core. That architecture can be emulated and simulated through ORCA-SIM, a simulation tool, also part of this work. The tool uses discrete-event simulation to speed up the simulation process, based on the URSA application programming interface, also part of this work. Software components are also discussed, including a library for hardware monitoring and energy consumption estimation, and a library for designing publish-subscribe systems. We present a compilation of results achieved in previous work and new experiments to cover the validation of the entire platform. For the latter, we focus the discussion on the design of a task reallocation mechanism based on self-adaptive components.

Keywords: multiprocessor system-on-chip, self-adaptative systems, computing system simulation.

ORCA: UMA PLATAFORMA MULTIPROCESSADA INTRA-CHIP AUTO-ADAPTATIVA

RESUMO

A complexa organização dos sistemas multiprocessador intra-chip (MPSoCs) demanda a organização inteligente dos recursos destes sistemas em tempo de execução, assim como a capacidade destes sistemas de otimizarem sua operação em situações atípicas, como a indisponibilidade temporária de hardware. Uma das abordagem utilizadas para gerência de recursos em MPSoCs é a da auto-adaptação, que aprimora estes sistemas através de componentes como sensores, atuadores, e lógica de decisão. Nesta dissertação, nós propomos ORCA, uma plataforma de desenvolvimento para auxiliar no projeto de sistemas auto-adaptativos. A plataforma provê abstrações para contornar a complexidade da auto-adaptação, baseado em trabalhos anteriores da área, incluindo uma arquitetura de hardware configurável, sistema operacional, bibliotecas de software, e ambiente de simulação. A arquitetura de hardware consiste de módulos de hardware de código-aberto e implementa uma arquitetura multiprocessada baseada em um processador compatível com o padrão RISC-V. A arquitetura também pode ser simulada e emulada através da ferramenta de simulação ORCA-SIM, parte desde trabalho. Esta ferramenta utiliza simulação de eventos discretos para acelerar o processo de simulação, e utiliza da interface de programação de aplicação (API) URSA, também parte deste trabalho. Componentes de software também são discutidos, incluindo uma biblioteca para o monitoramento de hardware e consumo de energia, e uma biblioteca para o projeto de sistemas publish-subscribe. Nós apresentamos um compilado dos resultados obitidos em trabalhos anteriores e novos experimentos para cobrir a validação de toda a plataforma. Para este último, focamos a discussão no projeto de um mecanismo de realocação de tarefas baseado em componentes auto-adaptativos.

Palavras-Chave: sistemas multiprocessados intra-chip, sistemas auto-adaptativos, simulação de sistemas computacionais.

LIST OF FIGURES

LIST OF TABLES

LIST OF ACRONYMS

ABB – Adaptive Body Biasing

ALU – Arithmetic and Logic Unit

AMBA – Advanced Microcontroller Bus Architecture

API – Application Programming Interface

ASOC – Autonomic System-on-Chip

BTI – Bias Temperature Instability

CPSOC – Cyber-Physical System-on-Chip

DES – Discrete-event Simulation

DMA – Direct Memory Access

DSL – Domain-Specific Language

DSP – Digital Signal Processor

DVFS – Dynamic Voltage-Frequency Scaling

EKF – Extended Kalman Filter

FIFO – First-In, First-Out

FPGA – Field-Programmable Gate Array

FSM – Finite State Machine

GMP – Global Manager PE

GPL – General-Purpose Language

HAL – Hardware Abstraction Layer

HAMSOC – Hierarchical Agent Monitored SoC

HEMPS – Hermes Multiprocessor System-on-chip

HPDC – High-Performance Distributed Computing

I/O – Input/Output

IOT – Internet of Things

IP – Intellectual Property

LMP – Local Manager PE

MAPE – Monitoring-Analysis-Planning-Execute

MAPE-K – Monitoring-Analysis-Planning-Execute-Knowledge

MPSOC – Multiprocessor System-on-Chip

NBTI – Negative Bias Temperature Instatibility

NI – Network Interface

NOC – Network-on-Chip

NORMA – No-Remote Memory Access

OC – Organic Computing

ODA – Observe-Decide-Act

ORCA – Self-adaptive System-on-Chip Platform

OVP – Open Virtual Platforms

PE – Processing Elements

PID – Proportional-Integral Derivative

RAM – Random-Access Memory

RISC – Reduced Instruction Set Computer

RM – Resource Management

RSP – Remote Serial Protocol

RTL – Register-Transfer Level

SAL – Software Abstraction Layer

SAS – Self-Adaptive System

SOC – System-on-Chip

SP – Slave PE

STL – Standard Template Library

SUS – System Under Simulation

TDDB – Time-dependent Dieletric Breakdow

VLSI – Very Large Scale Integration

CONTENTS

1. INTRODUCTION

Multiprocessor system-on-chip (MPSoC) is an important class of very large scale integration (VLSI) devices, emerged in the ’2000s, in which all (or most of) the components necessary for a multiprocessor application reside in the same chip [83]. The goal of the MPSoC technology is to fill a gap in the market for low-power/energy, high-performance platforms, which matches the requirements of embedded application domains such as robotics and autonomous vehicles, Internet-of-Things, and mobile communication and gaming. Nowadays, the market for MPSoCs targets many products, including vehicles [14, 54], smartphones, digital televisions, video games [58], and other specialized telecommunications and networking devices [62].

The ever-decreasing size of technology node has forced up the number of transistors per area, which increases year after year [69]. However, Moore’s law is bending [15, 55], and computing power for single-core chips have been limited by physical constraints that lead to power dissipation and frequency walls [61]. In this situation, MPSoC technology appears as an alternative to keep pace with the increasing demand for computing power for modern embedded systems by exploring massively parallel architectures together with smart resource management (RM) [16, 39, 68, 88]. RM allows for the runtime configuration of MPSoCs, dynamically adapting the system in the presence of atypical situations such as high workload and temporary hardware unavailability [48, 49, 53].

One notorious approach for RM in MPSoCs is self-adaptation. Systems with self-adaptive traits, called self-adaptive systems (SaS), have capabilities to manage themselves in the face of a changing environment [8]. In the domain of MPSoCs, self-adaptation plays an essential role in RM, whose studies encompass security [17, 76], fault-tolerance [63], energy management [48], and performance [87]. Since the late 90’s, many attempts to provide RM through self-adaptation were made, including those of organic and autonomic computing [85, 86, 87], passing through cyber-physical systems [72, 74, 73] and hierarchical management [30], ending up with modern ad hoc solutions [9, 24]. These attempts included modeling elements whose purpose makes sense only within their paradigm [46], rarely being able to be reused anywhere else.

In MPSoCs, self-adaptation permeates both software and hardware components. A self-adaptive technique represents the implementation of a self-adaptive trait and can include from the physical on-chip temperature sensor to software-level, application-specific counters in the design. For instance, an arbitrary self-adaptive technique may combine hardware-only actuation, e.g., dynamic voltage-frequency scaling (DVFS), with software-driven decision mechanisms. The number of available self-adaptive techniques is vast, and as far as we are concerned, there is no comprehensive bibliographic review on the matter. Nevertheless, from the studies on specific techniques, we can conclude that these tech-

niques share some specific features such as sensors (both software and hardware ones), actuators, and decision logic. Sensing, interchangeably called monitoring, provides the necessary information for the system to be aware of environmental (including itself) status. Then, a decision mechanism must use of sensing data to anticipate possible hazards to the operation of the system, or even explore possibilities for performance gains. Lastly, the system must have access to actuation points through components that allow for runtime configuration. The community translated this thought into the so-called control-loops, accompanied by one or another paradigm in the past. Examples of such control loops include MAPE [36], MAPE with knowledge (MAPE-K) [2], and ODA [47], all they emerged within the autonomic computing paradigm. For a comprehensive survey on self-adaptive systems engineering, see Krupitzer et al. [45].

1.1 Research Problem and Scope

MPSoC platforms with support for self-adaptation were developed in the past [3, 7, 74]. However, these platforms assume specific frameworks and are limited to support a particular set of techniques. As different paradigms have different assumptions about systems' functioning, porting techniques between paradigms can be challenging. One way to decouple platform-specific features and application programming interface (API) from self-adaptive techniques is to describe these techniques using well-defined building blocks, whose functioning rely on generic system components, usually implemented through a software abstraction layer (SAL). Since some techniques may require specialized hardware, there must be support for abstracting hardware through a hardware abstraction layer (HAL). Given the complex, non-trivial, and time-consuming activity of developing self-adaptive techniques, a platform to support the design of such techniques is of interest.

In this thesis, we present ORCA, a platform for developing self-adaptive techniques targeting MPSoCs. ORCA provides the tooling for developers to build self-adaptive techniques for MPSoC hardware organization, comprising a configurable number of programmable processor cores with private, no-remote memory access (NORMA) hierarchy and network-on-chip (NoC) communication infrastructure. We provide two communication models within the platform: the message-passing model, which is the most primitive mechanism for point-to-point communication, and the publish-subscribe model, which provides decentralized communication. We also provide libraries containing data structures for sensors, actuators, and decision-logic components. All software is build up on top of HellfireOS [40], a fully-preemptive, real-time operating system. Finally, URSA, a simulation API, provides the basis for building cycle-accurate simulators, also allowing hardware models to be modified to include new sensors, e.g., counters, and actuators, e.g., clock gating.

1.2 Goals

The goal of this work is to provide a platform for the development of self-adaptive techniques targeting MPSoCs. For this purpose, we present ORCA, which we claim to provide the necessary support for the development of these techniques. We present features that we designed during the work and the motivation to include them in the platform for the remain of the thesis. The following secondary goals apply to this thesis.

1. Propose an architecture for an MPSoC, and provide a reference hardware organization to aid the development of self-adaptive software. Such an organization would rely on existing and favorably open-source modules (Chapter 5).
2. Provide software abstractions to control self-adaptive aspects of the MPSoC from the software-level, incorporating features from other libraries and middleware, if available (Chapter 6).
3. Suggest an alternative to canonical register-transfer level (RTL) simulation to accelerate simulations of the system, preferably with minimum loss of simulation precision (Chapter 4).
4. Provide a taxonomy on the components for self-adaptation, as well as data structures to represent these components in the platform (Chapter 3).
5. Validate the platform by demonstrating the implemented features through minimal working examples, also focusing on the participation of these features in self-adaptation (Chapter 7).

1.3 Contributions

The main contribution of this work refers to the organization for the components of the proposed platform, specifically those related to self-adaptation. That organization is inspired by the paradigms that ruled the domain over the years. These paradigms represent years of development in several fields and encompass a massive amount of studies, reported in many conference proceedings, journals, and books. We summarize the contributions of this thesis as follows.

1. A cycle-accurate simulator, named ORCA-SIM, to simulate models written in C++, capable of estimating the energy consumption of the system under simulation (SUS). The simulator is made on top of URSA, an API for simulating computing systems. Both are novel contributions of this thesis (Chapter 4).

2. A hardware organization for an NoC-based, mesh-topological MPSoC, consisting of a router, network interface, processor core, memory modules, and auxiliary modules for off-chip communication. The provided hardware models are based on existing open-source register RTL projects. All hardware models, with exception to the processor core, are novel contributions of this thesis (Chapter 5).

3. Three software libraries: The first library provides monitoring of the system through hardware and software sensing. The second library supports the development of publish-subscribe applications, mainly targeting the dissemination of data within the system. The third library permits the system to communicate with peripheral and external modules. The three libraries are novel contributions of this thesis (Chapter 6).

4. Demonstrations on some features of the proposed platform, including functionalities such as task reallocation, energy estimation, and off-chip communication (Chapter 7).

1.4 Publications

During the master's period, the author submitted four conference papers and one journal article; all them have been accepted for publication [33, 21, 19, 20, 79]. We summarize the contributions presented in each publications as follows.

- 2018
  - Broker fault recovery for a multiprocessor system-an-chip middleware (SBCCI [19]). In the paper, the authors present a protocol to backup sensitive data from brokers in the MPQSoC middleware [32], along with a recovery protocol to use the backed up data to reestablish the publish-subscribe service in the system. The authors validated the work in MPSoC platform in Open Virtual Platforms (OVP).
  - Evaluating serialization for a publish-subscribe based middleware for MPSoCs (ICECS [33]). In this work, the authors performed experiments to measure the performance, code size, and memory usage of several serialization libraries. The experiments were performed on the same OVP platform from previous work [32].
- 2019
  - Integrating an MPSoC to a Robotics Environment (LARS [21]). In this work, the authors develop two nodes for a ROS [59] system that permit the ORCA MPSoC to communicate with external systems. The goal was to integrate the MPSoC with Gazebo [60], a robot simulator. As a demonstration, the authors present a synthetic random-walk application running in the MPSoC. The application reads

data from the robot's sensors (laser range and odometry) and actuates on the wheels of the robot to move it around an example room.

- 2020
  - Towards an integrated software development environment for robotic applications in MPSoCs (ISCAS [79]). In this work, the authors use the ROS nodes developed in the previous work [21] to provide a development environment. In the environment, Gazebo simulates the world and robot's physics, while the software runs in the ORCA MPSoC. The environment provides several hardware counters that enable energy estimation of the system through hardware characterization using the technique proposed by Martins [48, 49]. The authors validated the environment for a control application in a quadrotor vehicle.
  - A Fault Recovery Protocol for Brokers in Centralized Publish-Subscribe Systems targeting Multiprocessor Systems-on-Chips (ANALOG [20]. In this work, the authors extend a fault recovery protocol [19], making it modular so that parts of the protocol can be replaced by other existing solutions, contemplating a broader range of systems. The authors validated the protocol in two MPSoC platforms: (i) the OVP platform from the previous work [32], and (ii) HeMPS [56].

1.5 Thesis Organization

We organize the rest of the thesis as follows. In Chapter 2, we discuss the related work, including studies that resemble the one presented in this thesis. We center the discussion around the features of six system-on-chip (SoC) platforms, representing the paradigms that conducted the development of self-adaptive techniques for MPSoCs in the last two decades. We close that chapter by presenting the requirements that lead to the design decisions in ORCA. We present ORCA, the proposed platform, in four chapters. First, Chapter 3 presents an overview of the platform, organized in terms of software, hardware, and tools. In Chapter 4, we discuss the simulation environment, consisting of URSA, a simulation API, and ORCA-SIM, a simulation tool. Next, Chapter 5 presents the hardware modules that we integrated into the platform, followed by the software components, presented in Chapter 6. Chapter 7 brings demonstrations on several features of the platform, including task reallocation and software sensing. Lastly, Chapter 8 presents the final consideration of this thesis, including lessons learned during the development of this work, a list of features to be included in upcoming versions of the platform, and research directions encompassing self-adaptive techniques and resource management for MPSoCs.

2. RELATED WORK

Self-adaptation certainly encompasses several aspects of MPSoC design, such as sensing and actuation, decision logic, software-hardware logic partitioning, and task allocation. These concepts are generally developed aside from each other, although the literature presents many attempts to explore them altogether. In this chapter, we endeavor to summarize the progress of a couple of projects in the exploration of MPSoC design regarding self-adaptation. The goal is to identify which features make certain MPSoC platforms self-adaptive, as well as to identify the requirements that lead to some of the design decisions discussed in related studies. With that end, we report the results of a non-systematic review of the literature while discussing projects that, in our opinion, present remarkable contributions to the state-of-the-art in the field, in Section 2.1. Later in the chapter, in Section 2.2, we present the requirements for the platform proposed in this thesis, based on features that we identified in the discussed platforms.

2.1 A Non-Systematic Review on Self-Adaptive SoC Design

In this section, we discuss six MPSoC projects: Autonomic System-on-Chip (ASoC), Connective Autonomic Real-time Ultra-low-power System on Chip (CARUSO), Hierarchical Agent Monitored SoC (HaMSoC), Cyber-Physical System-on-Chip (CPSoC), A Self-Adaptive Organic Many-Core Architecture (DodOrg), and Hermes Multiprocessor System-on-Chip (HeMPS). The goal is to present an overview of each project, identifying components that enable self-adaptation in these platforms. For a comprehensive discussion on the features of each particular platform, please refer to the related publications.

2.1.1 Autonomic System-on-Chip (ASoC)

In 2005, Lipsa et al. [47] presented ASoC, a proposal for a framework to aid in the development of SoC platforms with self-adaptive features. They argued that some of the capacity of transistors in SoCs should be used to add self-adaptive features to these platforms, as the number of transistors was dramatically increasing year after year. These features would include higher fault tolerance, increased performance, power/energy efficiency, and a more straightforward system diagnosis.

In the ASoC framework, the hardware is separated into two layers. The first, the functional layer, might include all hardware of the system that is not related to self-adaptation, that is, the hardware that implements the functional requirements of the system. For the

non-functional requirements, the autonomic layer would group all the hardware that explicitly implements the self-adaptive features of the system, comprising monitors, actuators, evaluators, and a communication interface with the functional layer. Figure 2.1 (left) illustrates the organization of layers adopted in ASoC.

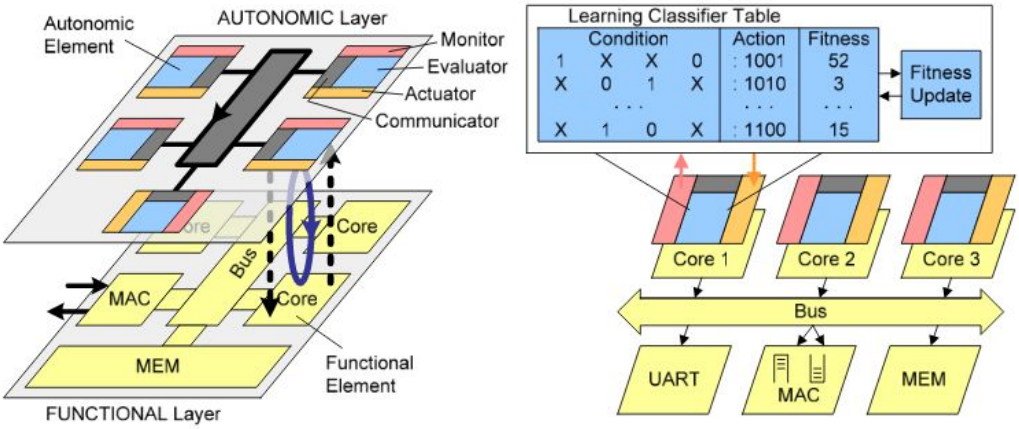

Figure 2.1 – ASoC's architecture with Advanced Microcontroller Bus Architecture (AMBA™) interconnection (left), and an learning classifier tables (LCT) serving as an evaluator (right) (adapted from [47]).

It is important to note that the authors presented no case study or validation of the architecture or the design methodology at that moment. However, ASoC appeared as the basis for other studies. In 2006, Bouajila et al. [5] studied error detection techniques and the feasibility of implementing them as monitors for the ASoC framework. The authors provide a fault classification and a survey on previous work but do not provide any results, as the work was reportedly in progress. Years later, in 2010, Zeppendfeld et al. [85] reported the successful implementation of a fault-tolerant CPU data path with the ASoC framework, as well as the application of reinforcement learning to fine-tuning the actuation of the system through learning classifier tables (LCT) [86]. They point scalability and reliability as future challenges while providing experimental results for an adaptive Ethernet MAC module.

Zeppenfeld et al. [87], in 2011, adopted the co-design of autonomic elements for the automated load-balancing of a task-based system. The software part of the system was responsible for task migration, while the hardware monitors the workload by intercepting data directly on the CPU. The system workload was calculated by reading the frequency and utilization of the CPUs. They used actuators to configure Leon3 cores to operate at the desired frequency following their workload. Regarding task migration, the autonomic elements implement a binding system for deciding whose core would run the migrated task. The platform was evaluated against a version using the dynamic voltage-frequency scaling (DVFS) technique and a static version of the same system. As a result, the authors report that the autonomic approach surpasses both the DVFS and static versions of the platform for all evaluated metrics.

From studies related to ASoC, we extracted components that we believe that closely relate to the self-adaptation, as shown in Table 2.1. We included only features explicitly mentioned in the papers to avoid misleading our review, performing the same analysis for all the platforms discussed in this chapter. From now on, by the end of each section, a table will display the extracted features for each platform. We distribute the extracted components in classes that group them according to their role in the system. We explain these classes later, in Section 3.2, as they guide the organization of self-adaptation in ORCA.

Table 2.1 – Summary on the features mentioned in studies for ASoC platform.

Label	Feature	Class
F-ASOC-1	Functional and autonomic layers for hardware components	Architectural
F-ASOC-2	Frequency Monitor	Physical Sensor
F-ASOC-3	Cycle Counter	Physical Sensor
F-ASOC-4	Workload Monitor	Logical Sensor
F-ASOC-5	Frequency Scaling (Clock Gating)	Physical Actuator
F-ASOC-6	Task Migration	Logical Actuator
F-ASOC-7	Voltage Scaling	Physical Actuator

2.1.2 Connective Autonomic Real-time Ultra-low-power System on Chip (CARUSO)

CARUSO is an approach for designing SoCs with a focus on autonomic computing, low energy consumption, connectivity, and real-time support, in addition to the existing requirements of embedded platforms [7]. Most of the proposed architecture relies on software and middleware for regulating networking and self-* properties[1] along with a helper thread. CARUSO uses the helper thread to treat monitoring and decision-making in the system. One of the most notable benefits of using an auxiliary thread to control the autonomic part of the system includes the support for real-time applications, where a fixed slice of time is reserved for the auxiliary thread to run using guaranteed percentage scheduling [44]. Finally, the helper thread would run in a multithreaded processor, separated from the reconfigurable parts of the system, deployed to an FPGA.

As validation, the authors claim to have applied the CARUSO approach to a swarm of robots application, whose organization is as shown in Figure 2.2. However, the paper does not provide any evidence of such a claim, nor a reference to previous work, future, or related work. Still, Herkersdorf et al. [34] argue that their work is complementary to CARUSO, as they invest their efforts exclusively to provide self-adaptation in hardware, contrary to

[1] Groups of properties that include self-management, self-adaptation, self-healing, and other properties with the “self-” prefix are commonly denoted as “self-*” or “self-x”.

CARUSO's, which relies exclusively on software. We present a summary of the features mentioned in single study involving CARUSO in Table 2.2.

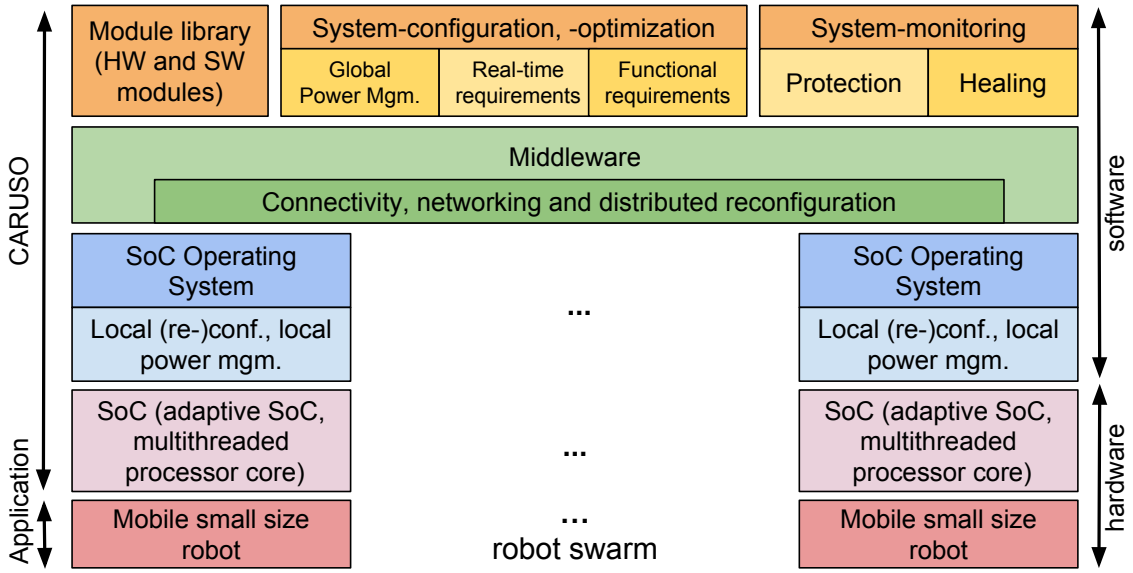

Figure 2.2 – CARUSO's architecture (adapted from [7]).

Table 2.2 – Summary on the features mentioned in studies for CARUSO platform.

Label	Feature	Class
F-CARU-1	Monitoring via auxiliary threads	Architectural
F-CARU-2	Energy Monitoring	Logical Sensor
F-CARU-3	Clock Frequency and Voltage Scaling	Physical Actuator
F-CARU-4	Dark Silicon	Physical Actuator
F-CARU-5	Real-time Constraints	Logical Sensor
F-CARU-6	Off-chip Communication Interface	Architectural

2.1.3 Hierarchical Agent Monitored SoC (HaMSoC)

HAMSoC [30] uses an hierarchical organization of agents for monitoring and actuating on a self-adaptive MPSoC. There are three levels of agents in HAMSoC: platform agent, cluster agent, and cell agent. The platform agent implements general monitoring and configuration of the platform, aiming for global optimization of the system, including network configuration and voltage island partition. The cluster agents apply the configuration to the associated clusters, while the platform agent decides on the configuration of the whole platform. However, the application of the configuration is performed per cluster, individually, since modern MPSoC may run applications with different resources requirements. On

the cell level, which corresponds to a processing element (PE) in ordinary MPSoCs, the cell agent traces the status of the local circuit, monitoring for faults, failures, and workload issues. Figure 2.3 shows the different agents of HAMSoC architecture.

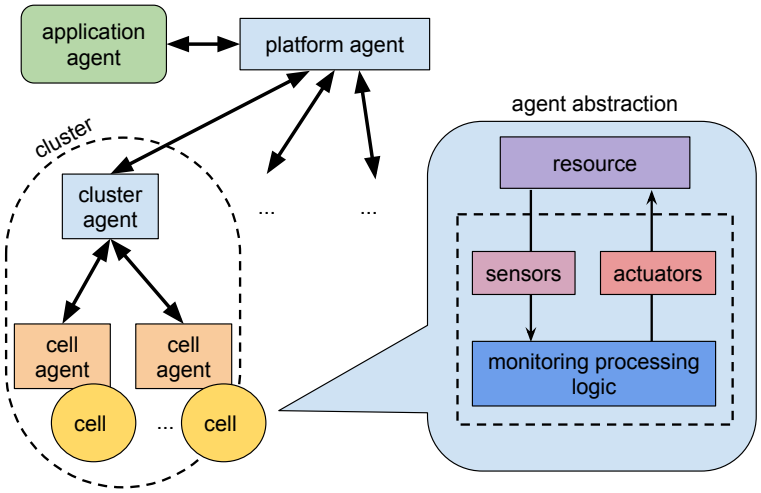

Figure 2.3 – HAMSoC Hierarchical Architecture (adapted from [30]).

HAMSoC provides a framework for formal modeling of agents, resources, and monitoring operations. Parameters of agents and resources can be specified so that they are visible to monitoring operations. Tuples specify the properties for agents and resources, while state machines specify the behavior of agents. They provide a design example of a hierarchical power monitoring system of a clustered MPSoC, containing clusters of four PE each. The many PEs were interconnected using on-chip routers in a mesh-based topology network-on-chip, called HAMNoC. It is important to note that one PE per cluster was dedicated to run the cluster agent, while the other three cell agents run in the remaining PE. In the provided example, application agents inform application requirements to cluster agents, which apply DVFS to the cluster to keep communication power as lower as possible without degradation of service's quality. Although the authors mention the role of the platform agent in the example system, it was not possible to determine the PE in which the platform agent is running. Also, the authors discuss no benefits of using the formal modeling framework. We summarize the features found for HaMSoC's studies in Table 2.3.

Table 2.3 – Summary on the features mentioned in studies for HaMSoC platform.

Label	Feature	Class
F-HAMS-1	Hierarchical Monitoring	Architectural
F-HAMS-2	Clock Frequency and Voltage Scaling	Physical Actuator
F-HAMS-3	Clock Gating	Physical Actuator
F-HAMS-4	Boundary Requirements	Logical Sensor

2.1.4 Cyber-Physical System-on-Chip (CPSoC)

Sarma et al. [74] proposed a self-aware and adaptive MPSoC platform with cross-level sensing and actuation named CPSoC. That platform relies on a middleware to provide self-adaptation based on the ODA (observe-decide-act) loop, where sensors are either physical or virtual, distributed over the layers of the CPSoC architecture. For instance, the platform can sense circuit delay, aging, leakage, temperature, and oxide breakdown at the circuit/device level. The key features of the platform include (i) physical and virtual sensing actuation, (ii) self-awareness and self-adaptation, (iii) cross-layer interaction between components, and (iv) predictive modeling and learning. CPSoC's adopts two networks-on-chips. The first, named cNoC (core-to-core NoC), treats packets for application purposes. The other one, named sNoC, treats the control packets. Recent work [71] suggests using hybrid NoCs to achieve real-time in CPSoC. Figure 2.4 shows an overview of CPSoC's architecture.

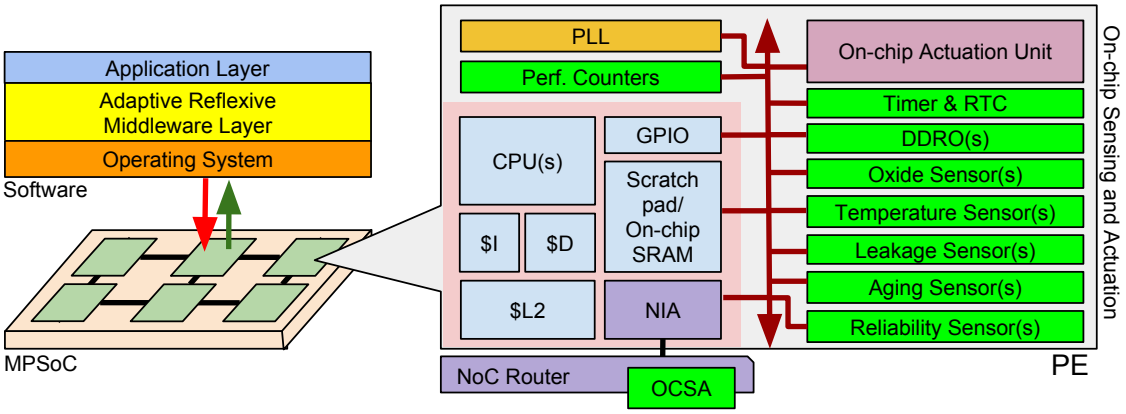

Figure 2.4 – CPSoC Architecture (adapted from [73]).

CPSoC permits the configuration of adaptive control policies, and to prioritize them according to their severity. For example, the system use data from aging, time-dependent dieletric breakdown (TDDB), bias temperature instability (BTI), and temperature sensors to detect impending failures. The implemented policy enforces the system to rest immediately, to heal itself by gating the clock and power of the block. This policy has severity four (4), which means their activities have priority on the system's resources against policies of severity three or less. Examples of such policies include negative bias temperature instability (NBTI) induced timing errors (3), thermal and power emergency (2), and thermal-induced short delays (1). Besides, the platform provides support for self-learning mechanisms, enabling the MPSoC to anticipate failures and predict vulnerabilities. For instance, the temperature control of CPSoC combines two hierarchical, cooperative ODA loops. The inner loops implement adaptive feedback control. This loop is encapsulated into another loop that uses online learning-based control. The thermal profiling of the system indicates that the peak of temperature is mitigated by adopting the control-based approach [73].

Table 2.4 – Summary on the features mentioned in studies for CPSoC platform.

Label	Feature	Class
F-CPSO-1	Layer organization	Architectural
F-CPSO-2	Application workload type	Logical Sensors
F-CPSO-3	Application power and energy consumption	Logical Sensors
F-CPSO-4	Application execution time	Logical Sensors
F-CPSO-5	System utilization	Logical Sensors
F-CPSO-6	Epoch length	Logical Sensors
F-CPSO-7	Context switch counter	Logical Sensors
F-CPSO-8	Thread load	Logical Sensors
F-CPSO-9	History	Logical Sensors
F-CPSO-10	Network bandwidth	Physical Sensors
F-CPSO-11	Packet/Flit statuses	Physical Sensors
F-CPSO-12	Channel status	Physical Sensors
F-CPSO-13	Congestion	Physical Sensors
F-CPSO-14	Latency	Physical Sensors
F-CPSO-15	Bus/Router power	Physical Sensors
F-CPSO-16	Branch miss	Physical Sensors
F-CPSO-17	Ckt delay	Physical Sensors
F-CPSO-18	Aging	Physical Sensors
F-CPSO-19	Leakage	Physical Sensors
F-CPSO-20	Power monitoring	Physical Sensors
F-CPSO-21	Temperature sensor	Physical Sensors
F-CPSO-22	Oxide breakdown	Physical Sensors
F-CPSO-23	Reliability	Physical Sensors
F-CPSO-24	Loop perforation	Logical Actuators
F-CPSO-25	Memoization algorithmic choice	Logical Actuators
F-CPSO-26	Degree of parallelism	Logical Actuators
F-CPSO-27	Code redundancy	Logical Actuators
F-CPSO-28	Task allocation	Logical Actuators
F-CPSO-29	Scheduling	Logical Actuators
F-CPSO-30	Task migration	Logical Actuators
F-CPSO-31	Offloading	Logical Actuators
F-CPSO-32	Duty cycling	Logical Actuators
F-CPSO-33	Adaptive routing	Physical Actuators
F-CPSO-34	Dynamic bandwidth allocation	Physical Actuators
F-CPSO-35	Channel number and direction	Physical Actuators
F-CPSO-36	Dynamic voltage-frequency scaling (DVFS)	Physical Actuators
F-CPSO-37	Adaptive body biasing (ABB)	Physical Actuators
F-CPSO-38	Reverse biasing	Physical Actuators
F-CPSO-39	Clock and power gating	Physical Actuators
F-CPSO-40	Multi-gate threshold	Physical Actuators

We show a summary of the features extracted from studies on CPSoC in Table 2.4. Please note that features for dynamic compilation, virtualization, and cache were intentionally omitted from the table as they do not relate to the runtime operation of the platform, nor they relate to the hardware organization of the proposed platform. For a comprehensive description of each feature, see Sarma et al. [72, 73, 74].

2.1.5 DodOrg – A Self-adaptive Organic Many-core Architecture

DodOrg [3, 22, 43] follows an organic computing approach by modeling the different elements of self-adaptive real-time embedded systems as organs from the human body. This approach relies on the multi-tier design of the system, including the brain, organs, and cell tiers. At the brain level, the system is goal-driven, and the parts of the system are coordinated by the brain to accomplish tasks (analogous to the human body). Next, the organ level coordinate subsystems, that is, self-contained smaller systems responsible for executing smaller but crucial tasks in the system (similar to organs, e.g., heart, stomach). At the lowest tier, the cells of the system organize themselves to form tissues.

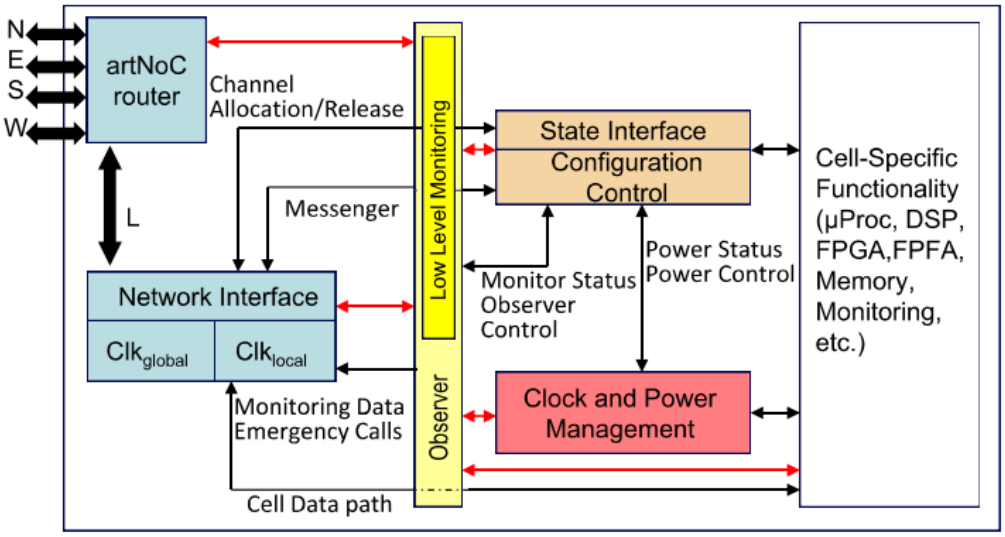

Figure 2.5 – Internal structure of a cell in DodOrg (adapted from [3]).

The hardware of Dodorg architecture relies on cells, which are similar to a PE in most MPSoCs. They argue that these cells can be placed in a mesh-based topology NoC, namely adaptive real-time network-on-chip (artNoC [75]), to form complex systems, supporting a couple of traffic classes: best effort, real-time, and broadcast. These cells can appear with digital signal processor (DSP), input and output (I/O), microprocessor, memory, and field-programmable gate array (FPGA) modules, as shown in Figure 2.5.

The thermal management of the system relies on dedicated clock domains to adapt the frequency to the current power budget of cells [22]. Regarding monitoring, the system

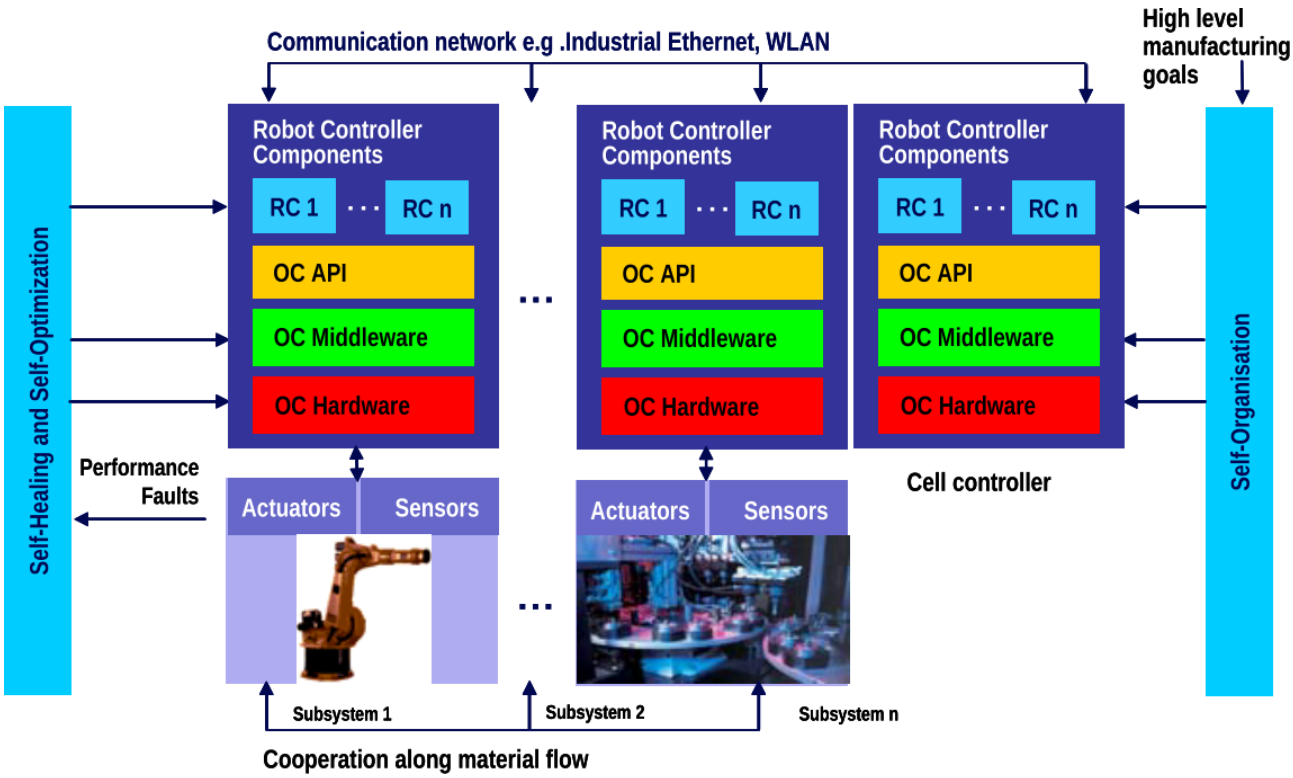

Figure 2.6 – Self-organization of an autonomous robot-based production cell [3].

uses the counters available in artNoC, e.g., link utilization, to detect abnormal situations such as deadlock in packets. Besides, their FPGA implementation of the system supports adding additional counters to the hardware, e.g., idle counter to specific datapaths.

The authors provide an example application applied to a robot swarm for an autonomous robot-based production cell. Each cell controller is attached to a subsystem and has an organic computing API, middleware, and hardware. The hardware interacts with sensors and actuators from the controlled subsystems, which cooperate to accomplish system goals. The middleware is responsible for abstracting the hardware from the cells and orchestrate the communication between them. In the application, the robot components use the API to interact with the rest of the system, as seen in Figure 2.6. We show a summary of the features extracted from studies on Dodorg in Table 2.5.

Table 2.5 – Summary on the features mentioned in studies for DodOrg platform.

Label	Feature	Class
F-DODO-1	Distributed management	Architectural
F-DODO-2	Thermal monitoring	Physical Sensing
F-DODO-3	Adaptive routing	Physical Actuation
F-DODO-4	Dynamic voltage-frequency scaling (DVFS)	Physical Actuation

2.1.6 Hermes Multiprocessor System-on-Chip (HeMPS)

HeMPS [10, 11] is not one but a family of MPSoCs that relies on Hermes NoC [56] as communication infrastructure, connecting several processing elements. The underlying network topology is usually mesh-based, but torus topology was also studied. Although HeMPS does not aim to provide the infrastructure for a self-adaptive system (at least not explicitly), many studies on SaS are validated using HeMPS as the platform, mainly focusing on the kernel and task migration [24], fault-tolerance [80], and security [9]. Newer updates on HeMPS were released under the name Memphis [66]; Hence, we consider HeMPS and Memphis as a single project.

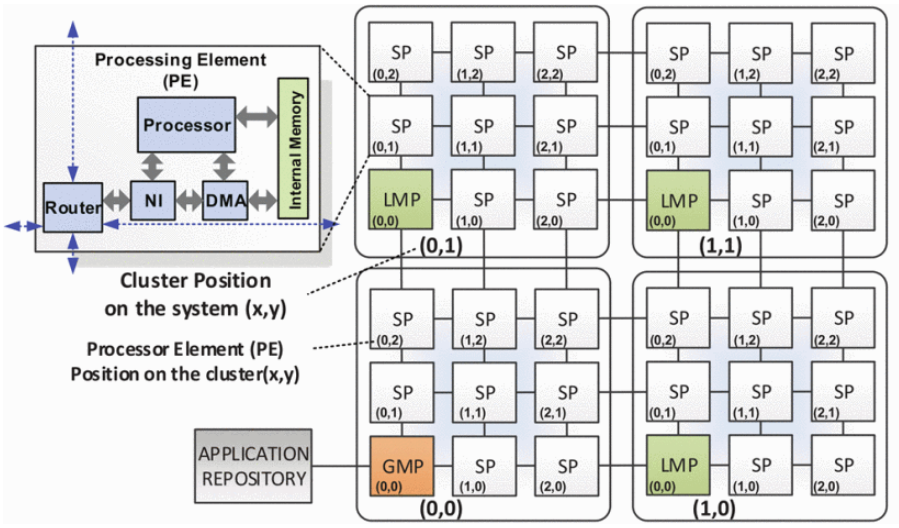

Figure 2.7 – HeMPS instance using a 6x6 mesh NoC [67]. The system is organized as 3x3 clusters. Node zero (0,0) is the global manager (GMP). Each cluster has a local manager node (LMP), with rest of nodes being slave nodes (SP). The GMP is connected to the application repository, from which the applications' tasks are loaded into the SPs.

In HeMPS, the processing elements comprise a network interface, direct memory access module (DMA), random-access memory (RAM) module, and a Plasma processor core. The Plasma processor is a 32-bit reduced instruction set computer (RISC) microprocessor that executes all MIPS I user-mode instructions. Recently, Ruaro et al. [67] reportedly merged the DMA and NI modules into a single module called DMNI, which presented performance gains when compared to the PE with uncoupled hardware. Figure 2.7 illustrates the organization of HeMPS.

HeMPS implements a protocol for dynamic loading applications into clusters of PEs. Applications' images (executable assembly and data) are stored in a task repository, and transferred to PEs at startup by the global manager, which is a particular PE responsible for processing administrative tasks for the whole system. They organize the system in homogeneous clusters. Each cluster has one of their PE dedicated to administrative tasks, called local manager PE, while the rest of the PE cares of applications' processing. A manager

PE enables the system to run task migration, load balance, and other distributed computing management, similar to a cluster of computers from the high-performance distributed computing (HPDC) domain. We show a summary of the features extracted from studies on HeMPS in Table 2.6.

Table 2.6 – Summary on the features mentioned in studies for the HeMPS platform.

Label	Feature	Class
F-HEMP-1	Hierarchical management	Architectural
F-HEMP-2	Thermal estimation	Physical/Logical Sensing
F-HEMP-3	Power estimation	Physical/Logical Sensing
F-HEMP-4	System utilization	Logical Sensing
F-HEMP-5	Dynamic voltage-frequency scaling (DVFS)	Physical Actuation
F-HEMP-6	Task Migration	Logical Actuation

2.2 Requirements for Self-Adaptation in MPSoCs

In the previous sections, we presented a couple of MPSoC projects that we believe to include the most relevant studies on self-adaptation in the domain. We focused on the architectural aspects of the platforms, i.e., the organization of the components and their role in self-adaptation. Although a couple of other studies treat self-adaptation in MPSoCs, we decided to omit them because we are more interested in the organization of the components involved in self-adaptation than in ad hoc techniques. For instance, it would be better to provide support to a broad number of sensors than to specific ones. Through the analysis of the included studies — including all the referenced papers in the previous sections —, we could determine the following requirements for the proposed platform.

2.2.1 Hardware Design

RQ1: Hardware Sensing. The platform must support adding (and removing) sensors from the architecture design. Sensing data must be accessible at the software level. Minimal support for sensors such as temperature, cycle, and heat must be observed, as these sensors partake in many techniques (F-ASOC-2, F-ASOC-3, F-CPSO-10, F-CPSO-11, F-CPSO-12, F-CPSO-13, F-CPSO-14, F-CPSO-15, F-CPSO-16, F-CPSO-17, F-CPSO-18, F-CPSO-19, F-CPSO-20, F-CPSO-21, F-CPSO-22, F-CPSO-23, F-DODO-2, F-HEMP-2, F-HEMP-3). We fullfil this requirement by providing a simulation

framework that permits including new hardware counters and sensors to the design (see Section 3.2), also providing an example set of hardware counters.

RQ2: Hardware Actuation. The platform must support the design of hardware-level actuation. Minimal support for DVFS and power gating must be observed, as these sensors partake in many techniques (F-ASOC-5, F-CARU-3, F-CARU-4, F-HAMS-2, F-HAMS-3, F-CPSO-33, F-CPSO-34, F-CPSO-35, F-CPSO-36, F-CPSO-37, F-CPSO-38, F-CPSO-39, F-CPSO-40, F-DODO-3, F-DODO-4, F-HEMP-5). We fullfil this requirement by providing support for the DVFS and dark silicon techniques (see Section 4.2.3).

RQ3: Support for Heterogeneity and Peripherals. The platform must support mixing multiple micro-architectures in the design since application in the domain commonly relies on hardware heterogeneity. Since SoCs are mostly designed to be part of larger systems, the platform must provide an interface that enables the MPSoC to communicate with external systems and peripherals (F-CARU-6). We fullfil this requirement by providing a modular architecture, based on tiles. We provide two example tiles (see Section 5.4 and Section 5.3) within our platform.

2.2.2 Kernel Features

RQ4: Real-Time Scheduling. The kernel included in the platform must support real-time scheduling, as many techniques use of auxiliary threads to process sensing data without impacting the quality-of-service of the running applications (F-CARU-1). We fulfill this requirement by adopting the HellfireOS kernel (see Section 6.1).

RQ5: Inter-Process Communication (IPC). Self-adaptation is an activity that inherently involves many processes. The kernel included in the platform must support IPC for either kernel and application tasks (F-CARU-1). We fulfill this requirement by adopting the HellfireOS kernel (see Section 6.1).

2.2.3 Software Design

RQ6: Software Sensing. The platform must provide the software basis for accessing sensors, including accessing registers from the hardware, in the case of hardware-implemented sensors, or system counters, in the case of software-implemented sensors (F-ASOC-4, F-CARU-2, F-CARU-5, F-HANS-4, F-CPSO-2, F-CPSO-3, F-CPSO-4, F-CPSO-5, F-CPSO-6, F-CPSO-7, F-CPSO-8, F-CPSO-9, F-HEMP-2, F-HEMP-3,

F-HEMP-4). We fulfill this requirement by using some counters provided by the HellfireOS kernel (6.1). Additional counters for using Martin's [48] technique for power estimation were included in the platform as well.

RQ7: Sensor Fusing and Composition. The platform must support the development of filters to treat data from sensors, as well as permit these sensors to be combined to generate new data on the system. We fulfill this requirement by providing software sensing within the platform. Software sensing permits treating raw data from hardware sensors, as well as perform sensor fusing (see Section 3.2).

RQ8: Software Actuation Support. The platform must provide the basis for implemented software actuation. Actuators such as task allocation and real-time parameters adjusting must be observed (F-ASOC-6, F-CPSO-24, F-CPSO-25, F-CPSO-26, F-CPSO-27, F-CPSO-28, F-CPSO-29, F-CPSO-30, F-CPSO-31, F-CPSO-32). We fulfill this requirement by providing task reallocation, which can be extended to implement task migration (7.1.3). The API for adjusting the real-time parameters of tasks is part of the HellfireOS kernel already.

2.2.4 Architectural Aspects

RQ9: Support for Centralized, Distributed and Mixed Software Architecture. As an inherently distributed system, the MPSoC must support the several architectural styles for resource management and system administration (F-ASOC-1, F-CARU-1, F-HANS-1, F-CPSO-1, F-DODO-1, F-HEMP-1). We fulfill this requirement by providing an organization for self-adaptation unbound of software architecture (see Section 3.2).

We provide an overview of the requirements implemented by the studied platforms in Table 2.7. Please note that some platforms may have intentionally missed the implementation of some requirements, while others may have implemented the requirement but not reported the implementation in the analyzed papers. The goal is not to compare platforms. Instead, our goals are to depict better what our platform is intended to achieve.

Although we could determine a minor set of components from the analysed studies, one cannot guarantee that these components comprehend all the features of the target platforms. We tried as much as possible to avoid our platform to be too restrictive on the implementation of the discovered features. Hence, we let the implementation of some components open to developers to decide the best architecture, mainly for software components. However, we provide a guidance on the organization of these components in Section 3.2, where we present the self-adaptive framework used within ORCA. Table 2.8 presents the support for the extracted features in ORCA.

Table 2.7 – Implementation of requirements per platform.

Platforms	Requirements								
	RQ1	RQ2	RQ3	RQ4	RQ5	RQ6	RQ7	RQ8	RQ9
ASoC	++	++	?	?	?	++	?	++	?
CARUSO	++	++	++	++	++	++	?	?	no
CPSoC	++	++	S	++	++	++	++	++	?
DodOrg	++	++	S	?	?	no	?	?	++
HamSoC	++	++	?	?	?	++	++	++	no
HeMPS	++	++	++	++	?	++	?	?	no
ORCA	++	++	++	++	++	++	++	++	++

(++) implemented, (S) partially implemented, (?) unknown, (no) explicitly not implemented

Table 2.8 – Features mentioned in related work how ORCA supports them.

Feature	Description / Note	Support	Feature	Description / Note	Support
F-ASOC-1	Architecture layering (HW)	+	F-CARU-1	Auxiliary threads[1]	+++
F-ASOC-2	Frequency monitor	++	F-CARU-2	*same as F-HEMP-3*	
F-ASOC-3	Cycle counter	+++	F-CARU-3	DVFS	+
F-ASOC-4	Workload monitor	++	F-CARU-4	Dark silicon	+
F-ASOC-5	Frequency scaling	+	F-CARU-5	Real-time constraints	+++
F-ASOC-6	Task migration	+	F-CARU-6	Off-chip comm.	+++
F-HAMS-1	Hierarchical monitoring	+	F-DODO-1	Distributed management	+++
F-HAMS-2	*same as F-CARU-3*		F-DODO-2	*same as F-HEMP-2*	
F-HAMS-3	*included in F-CPSO-39*	+	F-DODO-3	*same as F-CPSO-33*	
F-HAMS-4	Boundary requirements	+	F-DODO-4	*same as F-CARU-3*	
F-CPSO-1	Architecture layering (SW)	+	F-CPSO-21	*included in F-HEMP-2*	
F-CPSO-2	Workload type	+	F-CPSO-22	Oxide breakdown	d.n.a.
F-CPSO-3	App. pwr/energy consumption	++	F-CPSO-23	Reliability	d.n.a
F-CPSO-4	App. exec. time	++	F-CPSO-24	Loop perforation	d.n.a
F-CPSO-5	Sys. utilization	+++	F-CPSO-25	Memoization algorithmic choice	d.n.a
F-CPSO-6	Epoch length	++	F-CPSO-26	Degree of parallelism	d.n.a
F-CPSO-7	Context switch counter	++	F-CPSO-27	Code redundancy	d.n.a
F-CPSO-8	Thread load	d.n.a.	F-CPSO-28	Task allocation	++
F-CPSO-9	History	+	F-CPSO-29	Scheduling	++
F-CPSO-10	Network bandwidth	+	F-CPSO-30	*same as F-ASOC-6*	
F-CPSO-11	Packet/Flit status	+	F-CPSO-31	Offloading	d.n.a
F-CPSO-12	Channel status	d.n.a.	F-CPSO-32	Duty cycling	d.n.a
F-CPSO-13	Congestion	++	F-CPSO-33	Adaptive routing	+
F-CPSO-14	Latency	+	F-CPSO-34	Dynamic bandwidth alloc.	+
F-CPSO-15	Bus/Router pwr.	+++	F-CPSO-35	Channel num. and direction	d.n.a.
F-CPSO-16	Branch miss	+++	F-CPSO-36	*same as F-CARU-3*	
F-CPSO-17	Ckt delay	d.n.a.	F-CPSO-37	ABB	d.n.a
F-CPSO-18	Aging	d.n.a	F-CPSO-38	Reverse biasing	d.n.a.
F-CPSO-19	Leakage	++	F-CPSO-39	Clock and pwr. gating	+
F-CPSO-20	*same as F-HEMP-3*		F-CPSO-40	Multi-gate threshould	d.n.a.
F-HEMP-1	*same as F-HAMS-1*		F-HEMP-4	*same as F-CPSO-5*	
F-HEMP-2	Thermal estimation	++	F-HEMP-5	*same as F-CARU-3*	
F-HEMP-3	Energy and pwr. estimation	+++	F-HEMP-6	*same as F-ASOC-6*	

(+++) implemented (++) partially implemented (+) supported, but not implemented (d.n.a.) does not apply

[1]Implemented as real-time tasks

3. ORCA: A SELF-ADAPTIVE MPSOC PLATFORM

In this chapter, we present a short introduction to ORCA (self-adaptive multiprocess**or** system-on-**c**hip pl**a**tform), the proposed platform. The goal of this chapter is to explain the role of the parts of the platform superficially, serving as an introduction to the next three chapters. The first part of this chapter, Section 3.1, discusses the leading organization of the platform, which we define in terms of hardware, software, and tools parts. Chapter 5 (hardware) and Chapter 6 (software) present a more in-depth discussion on each of the first two parts. In the same section, the tools part is promptly explained, with Chapter 4 (simulation) presenting a more extended discussion on the simulation environment. The second part of the chapter, Section 3.2, discusses the support for self-adaptation in the platform.

3.1 Platform Organization

For this work, we assume a platform to be an operating environment that provides the necessary resources for one or more applications to run [25]. We consider as parts of a platform the related software stack, including support libraries, hardware drivers, and the operating system; the associated hardware architecture, including micro-architecture and memory organization; and tools, which permit developers to fast access the resources offered by the platform. Figure 3.1 presents a schematic depicting the organization of the platform in terms of hardware, software, and tools.

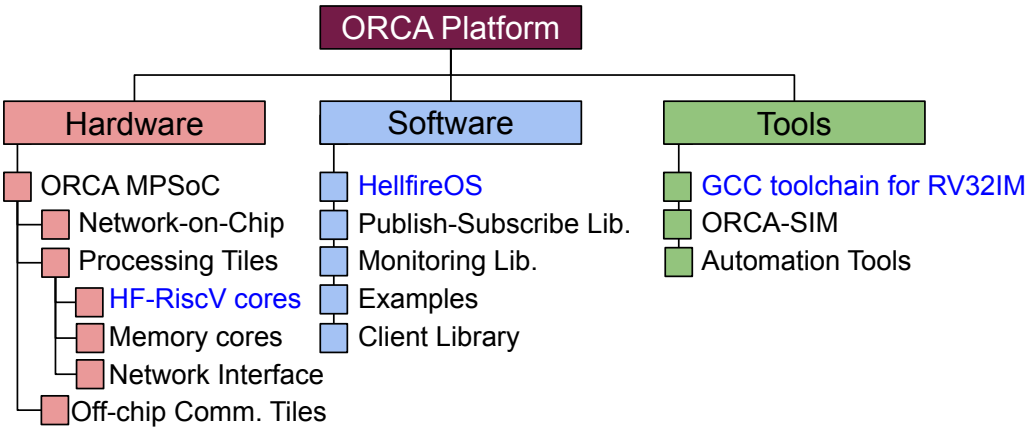

Figure 3.1 – The organization of ORCA platform, depicting software, hardware, and tools. Items highlighted in blue are third part work have included int this work with none or minimal modifications.

We refer to the hardware of the platform as to ORCA MPSoC. This MPSoC is based on existing open-source hardware, bringing components from other SoC projects. The first project is Hermes [56], a NoC whose design is parts of HeMPS and Memphis

platforms [10, 66]. Our hardware design is similar to the one in HeMPS in several aspects, including the NORMA organization and on-chip network topology. In contrast, ORCA has a different organization for peripherals. In HeMPs, peripherals are connected to border routers in non-local ports, while in ORCA, peripherals must necessarily connect to local ports of routers.

Nevertheless, HeMPs is a homogeneous MPSoC due to its computing nodes always implement the same ISA (MIPS2), while ORCA can have different cores with different ISA (e.g., RISCV32i and RISCV32im) because ORCA incorporates the HF-RiSCV processor core from the HFRISC SoC. We provide two tile designs for the MPSoC: (i) processing tiles, whose functioning corresponds to the processing elements in HeMPs, and (ii) off-chip communication tiles, whose hardware bridges the communication between the MPSoC and the rest of the system. We discuss all these components along with the rest of the hardware of the platform in Chapter 5.

Off-chip communication tiles cannot run any software, as they include no processor in their design. Thus, all the software run in processing tiles. An image containing a modified version of the HellfireOS — a fully-preemptive, real-time kernel —, applications and software libraries' code is loaded into the main memory of each processing tile at the startup. The modifications that we made for the kernel to work with the MPSoC hardware were minimal; thus, we omit them in this work. Two software libraries were developed on the top of HellfireOS: one provides support for the publish-subscribe communication model for applications, while the other one enables the access of hardware-level sensors and counters to software (memory-mapped I/O). Examples of applications that use both libraries are provided within the platform. We developed a third library for off-chip communication. This library can be deployed to other machines (e.g., a desktop computer) to enable that machine to use the NoC protocol for communicating with the MPSoC. We discuss HellfireOS and software libraries in Chapter 6.

The last part corresponds to the tooling of the platform. Generally speaking, tools include any software or artifact whose use applies only to the development stages of the project. The tools that we use in ORCA include the toolchain for the HF-RiscV core, automation scripts (written in bash and make languages), testing applications to validate off-chip communication, and simulator. The toolchain corresponds to the GNU Binutils 2.28, containing linker, assembler, and other facilities for manipulating binary files; and GCC 7.1.0, including front end for C and C++ languages. Automation scripts provide facilities for debugging and configuration and are intended to be platform-specific, tested with Debian (Stretch and Buster), and Ubuntu (16.04.6 LTS). Testing applications for off-chip communication consists of UDP/IP applications interacting with the simulator tool whose packets can be inspected using the provided plug-in for Wireshark [82]. The simulation tool used in this work, ORCA-SIM, and the correspondent simulation API, URSA, are discussed in Chapter 4. Technical information on the testing environment is presented in Chapter 7 (validation).

3.2 A Development Environment for Self-Adaptive Techniques

The platform presented in this work is intended to be used for both software and hardware development, as many self-adaptive techniques involve the co-design of software and hardware projects. We provide support for the development of several components for both hardware and software. First, our simulation API provides an environment for dealing with hardware-specific issues, fulfilling RQ1, and RQ2 (see Section 2.2). To ease the design and maintenance of hardware modules, we provide a minimal set of hardware modules that can be modified to incorporate more features, or even combined to form new platforms. For the software, we provide a lightweight framework for the design of self-adaptive techniques that control the hardware resources from the application level. The goal is to organize the resources of the platform to favor the development of these techniques. For instance, we do not enforce any mechanism on the decision logic of techniques. Instead, we provide a set of elements to facilitate the integration between components of the system, favoring the reuse of project artifacts, e.g., separation of concerns between evaluators and decision logic. In ORCA, self-adaptive techniques comprise several components. These components fall in one of the following categories: (i) physical sensors, (ii) logical sensors, (iii) evaluators, (iv) decision logic, (v) system state, (vi) policies, (vii) goals, (viii) procedures, (ix) actions, and (x) switches. We explain each of these components below. Please note that these elements may appear in others' work with different naming. Figure 3.2 displays how we organize these components in the system.

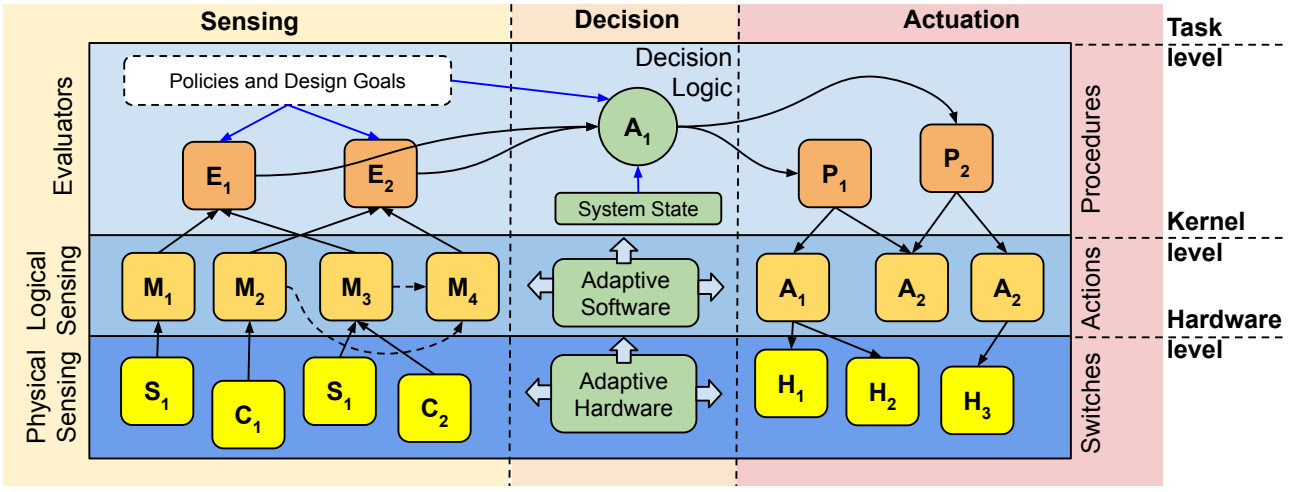

Figure 3.2 – Organization of components for self-adaptation in ORCA. Components are distributed in hardware, kernel, and application levels, grouped for their role in the system: sensing, decision, and actuation. Black arrows indicate the dataflow, and blue arrows indicate dependencies resolved at the design time. Adaptive hardware and software have the decision role, although they work separated from the rest of the system. The system state is an hypothetical set representing any data relevant for decision-making.

3.2.1 Physical Sensing

Physical sensing corresponds to the reading of hardware sensors and counters. Examples of counters include performance counters embedded in nowadays's general-purpose processors, while sensors correspond to specialized hardware to measure temperature, power, and aging of chips' components. In ORCA, we provide a couple of counters, embedded to hardware modules of the platform — a comprehensive list of the counters in the platform is provided within Chapter 5. We assume sensors to be mapped to the memory space, accessible at the software level. Since data from sensors may not represent useful data due to noise or invalid data, we delegate the treatment of data to logical sensors.

3.2.2 Logical Sensing

Logical sensing regards the treatment of raw data from physical sensors, software-level sensing, and sensor fusing. For the first, data is usually treated by peripheral drivers, while the other two are hardware independent. Software-level sensing includes readings from the kernel, e.g., the number of deadlines missed by one task and CPU utilization, and application-level counters such as response time and workload. Sensor fusing enables the complex organization of sensors. For instance, the pose estimation of a vehicle using a Kalman filter [79] is an example of sensor fusing. Another example can be calculating the average temperature of the system through the mean of two temperature sensors. Data from logical sensors is intended to be consumed by evaluators, the kernel, and other logical sensors (composition). We present a couple of logical sensors in Chapters 6 and 7.

3.2.3 Evaluators

Evaluators are entities, usually tasks that check on the current state of the system to identify abnormalities to the system's functioning or to detect opportunities for performance gains. These evaluators can calculate whether the value of a variable went over some threshold, or even use complex machine learning algorithms to detect abnormal states. In both cases, evaluators may decide if the system must adapt or not. In this work, we assume these evaluators to work by periodically polling on logical sensors, although their implementation at the kernel level could permit them to work reactively. Whether evaluators decide that the system must adapt, the decision logic is activated to plan on the next steps to be taken by the system. An example of an evaluator can be found in Chapter 7, which relies on the number of missed deadlines to activate the associated decision logic.

3.2.4 Decision Logic

Decision logic corresponds to any techniques, algorithms, or calculi for determining a sequence of actions for the system to go from one state, usually the current state, to another state, the desired state. Decision logic can only be activated by an evaluator, which sets the desired state of the system. Thus, decision logic has nothing to do with deciding on the goals of the system. Instead, decision logic has to build a plan for the system to reach the state set by the evaluator. Thus, a decision roughly corresponds to a planning activity. However, algorithms simpler than those from planning can be an option, since the number of possible actions in the system is conditioned to the number of available procedures.

A system can implement several decision logics. An evaluator can trigger one of these logics, taking into consideration the design goals and policies of the system. For instance, one decision logic can consume less energy than another, although providing a more accurate plan of action. If the convergence of one algorithm cannot be verified, multiple algorithms can execute in the hope of using the results from the first algorithm to converge. Also, multiple decision logics can act in different parts of the system if these parts do not depend on each other. Finally, the decision logic must always observe the state of the system since it can change during the decision-making process.

3.2.5 System State

The system state is a model consisting of sensors' readings and history. If the system state considers history [73], the source of the data becomes a sensor — any variable in the system is a sensor if considered in the self-adaptation model. For instance, the deadline-misses evaluator presented in Chapter 7 considers the number of deadline misses (logical sensor) for a task since the system startup.

3.2.6 Policies and Goals

The design goals of the system regard to restrictions on the final application of the system. For example, one system may focus on quality-of-service; another system may focus on performance; and a third system may focus on low power consumption. Policies, however, focus on constraints on the operation of the system, usually bound to non-functional requirements. For example, one constraint can be the system to operate lower 80% of CPU usage, or the power consumption to be always below some threshold.

3.2.7 Procedures

Procedures are deterministic sequences of actions. Similarly to a real-time task, a procedure has timing requirements. However, procedures can also have explicit resources requirements, such as energy consumption and required network bandwidth. Decision logic uses procedures requirements to decide the best course of action to take, also considering the state of the system. We consider that two or more procedures can produce the same effects on the system while consuming different resources. For instance, one algorithm for task reallocation can take more time due to it looks for the optimal configuration, while another one may use some heuristic to find a proper place for the task to run, thus consuming less time but offering a slightly less efficient solution.

For the sake of the integrity of the system, multiple procedures cannot be executed at the same time if they impact on each other. This restriction enforces that the system will not overreact to abnormal events. An example of such a situation could be migrating a critical task to one CPU while gating the clock of that CPU at the same time, thus modifying the real-time parameters of that task. Although we suggest using synchronization mechanisms to avoid conflict between procedures, we are permissive on actuation, leaving the design of synchronization mechanisms to programmers to decide.

3.2.8 Actions

Actions correspond to atomic routines that cannot abort once started and rely mostly on built-in routines from the kernel (e.g., system calls) and support software (e.g. standard library). Examples of actions include spawning, killing, or allocating a task. In practice, every single function (or method) in the kernel, application, and other software elements can be an action. Actions differ from procedures in that procedures correspond to the timed execution of sequences of one or more actions, although actions can exist in the system without being part of any procedure.

3.2.9 Switches

Switches are hardware-specific mechanisms such as DVFS and power/clock gating. To be included in the self-adaptive model, these mechanisms must allow for external configuration. For instance, there could be a pin for controlling the operation mode of a CPU or even pins for activating one zone or another for DVFS. These pins would be mapped into memory space and accessed via software.

4. URSA: A MICRO (μ) RAPID-SIMULATION API

In this chapter, we present URSA, an application programming interface (API) for the modeling and simulation of computing platforms. In this work, we use URSA to model and simulate the ORCA MPSoC. We briefly discuss the motivations behind URSA in Section 4.1, where we also present some preliminary background on system simulation. Section 4.2 presents the simulation strategy adopted in URSA, which combines discrete event simulation with finite state machine models. We dedicate Section 4.3 to discuss the simulation API of URSA, where we point out the limitations of the chosen approach while suggesting alternatives to surpass these limitations. Finally, in Section 4.4, we present ORCASIM, a simulator made on top of the URSA API to simulate the ORCA MPSoC.

4.1 Background and Motivation

Simulation is an important activity in systems' development life-cycle, allowing development teams to detect defects early in the project. Similar to other verification and testing techniques, simulation can represent savings to the project's budgets since the costs of correcting a bug early in a project tends to be cheaper than that later in the project [64, 78]. In this context, simulation tools play an essential role in the project of embedded systems as they permit to validate functional aspects of the system early in the project.

Typical simulation tools are shipped in with a language, which can be either an extension of another general-purpose language (GPL) or a domain-specific language (DSL). In both cases, these languages provide constructions for describing the behavior of systems through models — abstractions that represent parts of systems. Another core component of simulation tools is the simulation engine, which interprets the associated language; these engines adopt different simulation approaches to extract information from the execution of underlying models. Examples of DSLs for system modeling include VHDL and Verilog (and SystemVerilog), implemented by standard on-the-shelf tools such as Mentor's Questa [50] and XILINX's ISE Design Suite [84], while extensions of GPLs include System-C [1], an extension of the C++ language; Open Virtual Platforms API (OVP) [37], an API over C language; and Gem5 [4], supporting Python and Ruby scripts.

Simulation tools usually aim either for (i) the emulation of the system behavior, whose goals include fast system prototyping and software validation, or (ii) highly accurate architecture simulation, which focuses on the validation of lower-level architecture aspects such as hardware synchronization protocols and energy evaluation. However, some tools may fall in-between these categories [65]. Highly accurate simulation tools usually rely on register-transfer level (RTL) languages, which is the preferred approach for modern on-the-

shelf tools. Advantages of RTL languages include their capability for synthesis, although modeling must be limited to use a subset of the language, as is the case of VHDL and SystemVerilog languages. Systems modeled in RTL have their behavior described mostly in terms of processes and signals, and modern languages' standards also consider object-oriented programming (OOP) in their specification. Although RTL languages provide a vast set of abstractions to describe systems, one drawback of using RTL is the time taken to simulate RTL models.

As an alternative to the canonical RTL modeling, system-level simulation provides a faster simulation environment, mostly at the cost of model accuracy. As well as in RTL, system-level simulation permits the modeling and simulation of computing systems through abstractions. However, system-level models may not be suitable for synthesis as languages may not detail the internal structures of circuitry. Instead, system-level simulators focus on the fast prototyping and simulation of systems to validate software components in the early stages of the project, e.g., peripheral drivers, while the hardware is still under development. Depending on the complexity of the system under simulation, system-level simulators can even outperform physical systems [77]. It is important to note that the performance of system-level simulators depends on the host machine's hardware, i.e., the architecture in which the simulator runs in.

URSA is an API for system-level modeling and simulation. As an API, URSA provides a set of classes, structures, enumerations, and other language-related assets that can be used to create system-level, cycle-accurate simulators. Contrarily to solutions that extend the syntax of general-purpose languages such as Spec-C [28], System-C [1], and Handel-C [51], URSA relies exclusively on existing C++ language constructions and standard library. Frameworks such as MyHDL [57] and JHDL [6] follow a similar approach, although they require Python and Java to run, respectively.

URSA presents a few advantages over simulators mentioned above. First, URSA has no dependencies on libraries or tools other than the standard C++ library, facilitating the integration of simulators created with URSA within other systems, e.g., web-based interfaces. Besides, C++ compilers can emit executable binaries whose execution tends to be faster than interpreted code (in the case of Python) or binary translation (in case of Java). Second, although a system-level simulation, URSA is capable of generating cycle-accurate results, since simplifications to system models permit URSA to ignore the internal state of hardware models. Lastly, the trade-off between simulation accuracy and speed can be worked out directly in models (see Section 4.2.2). We dedicate the rest of this chapter to discuss the simulation model behind URSA along with its API and implementation-specific features.

4.2 Simulation Model

In URSA, hardware models correspond to a set of finite state machines (FSM). Formally, a FSM [35], as Equation 4.1 shows, is a 6-tuple, where Q is the set of states, q_0 is the initial state, X is the set of inputs, Y is the set of outputs, δ is the transition function, and F is the set of final states. URSA comprises only the simulation of synchronous systems, that is, systems whose behavior is coordinated by clock events, i.e., cycles. Then, a cycle corresponds to the activation of the transition function of the FSM underlying to the system under simulation (SUS). When the SUS comprises multiple hardware modules, the transition functions of all underlying FSM are activated. Transition functions are activated one after another, respecting the time in which they would occur in a real system. Theoretically, some transition functions would be activated at the same time. However, a non-deterministic choice determines what functions will be activated first, as multiple functions cannot be activated at the same time in the adopted simulation model. Finally, the activation of a transition function, represented by an event, occurs based on the frequency of the associated hardware module, also supporting multiple clock domains and both edges of the clock.

$$FSM =< Q, q_0, X, Y, \delta, F >, \text{ where } q_0 \in Q, \quad \delta : X \times Y \rightarrow Q, \quad F \subset Q \tag{4.1}$$

An event, shown in Equation 4.2, is 3-tuple where t is the time in which the transition function of the FSM m must be activated, and T is the period of the hardware module associated to m. At each cycle, the value of T is added to t, denoting progress in simulation time. It is important to note that T can change during the simulation, allowing for the simulation of dynamic voltage-frequency scaling (DVFS)(see Section 4.2.3), for example. It is important to note that the simulation model adopted in this work resembles the one of discrete-event simulation (DES) [23, 42], although we do not guarantee both models to be equivalent.

$$Event =< t, m, T > \tag{4.2}$$

$$SIM =< \Phi, S, t_0, n, E, P >, S = \bigcup_{i=1}^{|\Phi|} \Phi_i.X \cup \Phi_i.Y \tag{4.3}$$

A simulation (SIM), as shown in Equation 4.3, is a 6-tuple, where Φ is the set of FSM, S is the set of shared signals, t_0 is the time in which the simulation begins, n is the amount of time to run the simulation, P is the priority-queue of events, and E is the set of initial events. At the time t_0, events associated with each of the transition functions ($\forall x \in \Phi, x.\delta$) of all FSM are created and pushed to P. From then, the simulation engine pops

an event from P, activates δ once, updates t by adding T, and push the event back to P. The simulation ends when there are no more events to occur before n, that is $\forall e \in P, t(e) > n$. Since elements in P are sorted by t, it is guaranteed that events that occur early will be popped from P first. Finally, FSMs are connected to each other by signals. Formally, a signal $s \in S$ is any input or output shared between two or more FSMs. By convention, two signals with the same name are considered the same signal, for any FSM.

Algorithm: System Simulation

```
1.  Input
2.      E is the set of initial events
3.      t0 is the time for the simulation to begin
4.      n is the time for the simulation to finish
5.  Locals
6.      P is a Priority Queue
7.      et is the current event
8.      t is the current simulation time
9.  Begin
10.     t ← t0
11.     For all e ∈ E:
12.         push e to P
13.     While t ≤ n, do:
14.         et ← top of P
15.         t ← et.time
16.         call et.m.δ
17.         et.time ← et.time + et.cycleTime
18.         pop from P
19.         push et to P
20. End
```

Algorithm 4.1 – The algorithm for system simulation used in URSA.

4.2.1 Hardware Models

Hardware models are representations of real hardware functioning, often described in RTL. In this work, hardware models correspond to simplified versions of synthesizable RTL models from several projects, each model corresponding to one hardware module — it is carried out by the simulation engine. In URSA, hardware models must be described in terms of finite state machines, using the classes provided by the API, in C++, with at least one FSM per module. From the simulation viewpoint, there is no difference between simulating one FSM or two for the same hardware module because, in practice, it is as if all

the modules were one, with several processes (FSMs) distributed all over a single model. The FSM inputs correspond to the signal to which these processes are sensitive. Models are also absent of clock signals as the simulation model guarantees the correct timing for each module. Inputs of models correspond to the input signals of hardware modules, as well as outputs corresponds to the output signals of those modules. The internal state of the hardware, represented in RTL by variables and signal, is mapped to states in the FSM. Figure 4.1 shows the structure of a hardware module.

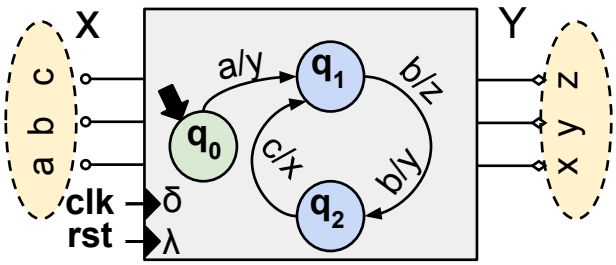

Figure 4.1 – An illustration of a hardware model in URSA. Hardware models comprise a set of inputs (sensitive signals), a set of outputs (non-sensitive signals), clock and reset signals (implicit, implemented by the simulation engine), and an internal state machine (explicitly programmed.

For this work, specifically, the following assumptions hold for modeling hardware modules as FSM.

- The state $q_0 \in Q$ is the initial state, to be called the *reset state*.
- The must be a transition defined from every state to the reset state, including one from the reset state to itself. This transition has the form λ/λ, with $\lambda \in X$ and $\lambda \in Y$.
- The set of final states F has cardinality zero, and the simulation can end regardless the state of any individual hardware model.
- Undefined inputs will be considered as transitions from the source state to itself with the form $t \in \delta :< Q_0, x > \rightarrow < Q_0, \lambda > \ \mid \ x \in X$ and $\lambda \in Y$.
- One state cannot have two or more transitions with the same input, i.e. non-deterministic choices cannot exist.

4.2.2 Performance vs. Accuracy Trade-off

As stated before, the performance (and accuracy) of models depends on their implementation as models are described using C++, and poor coding may jeopardize the performance of the whole simulation. The same applies to accuracy, as developers may model

the hardware using practically any feature available in C++. For instance, the HFRiscV model used within our platform can be either cycle or instruction-accurate. When working in cycle accuracy mode, the model has tracking of branches and adds bubbles to the pipeline when convenient, for example. When in instruction accuracy mode, the model has less code to execute as instructions' types are not observed anymore — the simulation would run faster. By running an image containing no application (HellfireOS kernel only), we could observe that disabling branch prediction reduced the execution time for the HFRiscV core model by approximately 2%, for example. Of course, by removing that feature we also remove precision from the model. However, we can tune the model to work "more closely" (or not) to the real design as much as we need, depending on the simulation purpose. Similar effects can be observed when disabling network congestion in router modules, memory space integrity in memory modules, or overflow/underflow checking on buffers. See Chapter 5 for more information on the configuration of these models.

4.2.3 DVFS Simulation and Dark Silicon

We can simulate DVFS by changing the period of a model during the simulation. Changes to the state of models are accompanied by timing information, which should be reported to the simulation engine right after each state transition. By manipulating timing information, we can, for example, decrease the system's performance by 50% by increasing the timing twice their default value. In the same way, we can reduce the timing information indefinitely. This strategy makes it possible to simulate dark silicon by reducing timing information to zero, simulating a "frozen hardware". Of course, by doing this, we enforce DVFS zones to be common multipliers. Besides, we can surpass this limitation by inserting fake, empty cycles to force the simulation to advance in time while not changing the state of models. Fake cycles can be added to models by using dummy states in underling state machines, serving as a strategy for implementing dark silicon in the platform. Tutorials for generating such models are provided withing the appendix sections.

4.3 URSA Simulation API

We actively use object-oriented design to provide entities to model and simulate systems. Since we design URSA having the simulation of ORCA MPSoC in mind, we present only the classes required to simulate ORCA, included in the ORCA-SIM design. However, one can extend these classes to simulate other systems. We developed two packages for this thesis: (i) the simulation engine package, which corresponds to the classes and entities for the simulation engine, and (ii) the model's package, corresponding to general-

purpose hardware models (e.g. memory cores). We extend the later to contemplate a broader set of IPs by adding specialized cores (e.g. processor and on-chip network router).

4.3.1 Simulation Engine Package

The simulation engine package comprises classes for dealing with the overall simulation model, including abstractions for a priority queue (PRIORITYQUEUE), events (EVENT), and hardware model (MODEL, UNTIMEDMODEL, and TIMEMODEL). The simulation engine (SIMULATOR) serves as a facade [27] to the whole package. A simplified class diagram representing the class hierarchy of the simulation engine package is shown by Figure 4.2.

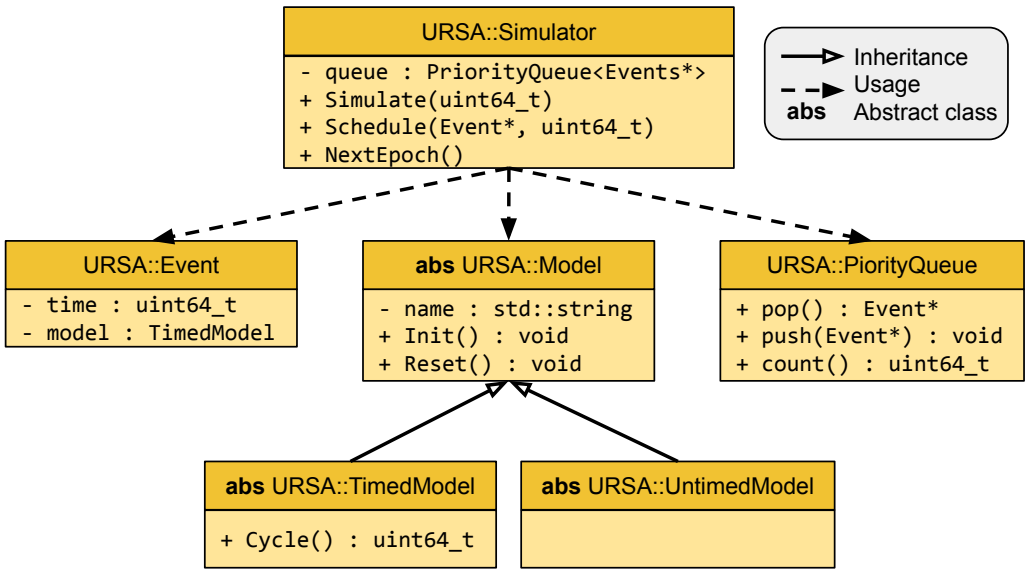

Figure 4.2 – A simplified class diagram representing the hierarchy and dependency among classes provided within the simulation engine package. Class fields are denoted by a "-" character, and methods are denoted by an "+" character. We intentionally omit auxiliary fields and methods.

- The **Event** class models a clock event, similarly to the POSEDGE/NEGEDGE construction of SystemVerilog or CLOCK'EVENT in VHDL. An underlying model is associated with each of these events. The execution of an event corresponds to the update of the internal state of the associated model. As a side effect of the event execution, the simulation time advances to the point in which that event would occur. For this reason, events must be executed in-order; hence, these events carry a time tag within them. For events carrying the same time tag, the simulation will execute these events following a non-deterministic order. After their execution, events are scheduled to be ocurr in the next hyper-cycle. An hyper-cycle correspond to the execution of the all cycles associated to the same point in time.

- The **PriorityQueue** class provided within ORCA wraps up the standard priority queue of C++ language (GNU's LIBSTDC++). This priority queue is used by the simulator class to rapidly sort events by the time that they must occur, preserving simulation's consistency. Once an event occurs, the simulator pops that event from the queue, executes the underlying model, and pushes that event back to the queue. Hence, the queue guarantees that the event with the least time tag would be popped first. However, the same event can be placed back in the queue in any place, as the new place depends on the time scheduled for its next execution.

- **Model, TimedModel and UntimedModel** are representations of hardware modules that correspond to parts of the system under simulation. Depending on the goals of the simulation, some hardware modules can be treated as combinational. For instance, in this work, we consider memory operations to be instantaneous, that is, writings and readings from the memory does not consume time. However, the CPU is timed, and will respect the timing constraints of the RTL design, thus accessing the memory only in certain points of the time. By removing the timing constraints from the memory core, we alleviate much of the simulation effort, which access time roughly compares to writing or reading from host's memory core. For those cases, we provide the UntimedModel class, which models modules that does not require to be scheduled in the simulation queue. For modules that depend on the simulation clock, we provide the TimedModel class, which has an associated Run method. This method is called once an event is popped from the simulation queue, and returns the number of time units that the underlying state machine spent to change states. This is a simple (although powerful) mechanism to emulate multiple clock domains. Besides, we use the same mechanism to treat the simulation of the processor core in this work, as it has instructions leaving the pipeline in different amounts of cycles. Any model must inherit strictly from either TimedModel or UntimedModel. Their base class, Model, provides general-purpose methods to extract general model information, e.g., their name.

- Finally, the **Simulator** class groups other classes in the package in a single simulation system. It contains an internal priority queue, variables to control the simulation time, operations to start and stop the simulation, and the control of signals' writing — which means that the simulation will treat signals and prevent them from being wrote outside the edge of the clock. Although we use only one instance of a simulator in this work, it is allowed to run multiple simulations simultaneously, including communicant systems. Additional features include the simulation of systems with multiple clock domains and energy estimation through hardware characterization. For this purpose, the increment of the tag must coincide with the frequency of the simulated hardware clock. It is important to note that the Simulator class does not implement the instantiation of hardware models, and thus it must be called from within an upper-level software layer.

4.3.2 Model Package

The model package provides generic hardware models to aid in the design of other modules. These modules correspond to memory cores (MEMORY), FIFO buffers (BUFFER), and busses (BUS). These models support reading and writing, although they have minor differences in the way that they carry data. Since we are not interested in the internal implementation of these models, we assume their operation to be executed within one cycle, not requiring them to be scheduled along with the remaining hardware parts of the system; thus, they inherit from the UNTIMEDMODEL class. An illustration of the hierarchy of the model package is shown in Figure 4.3.

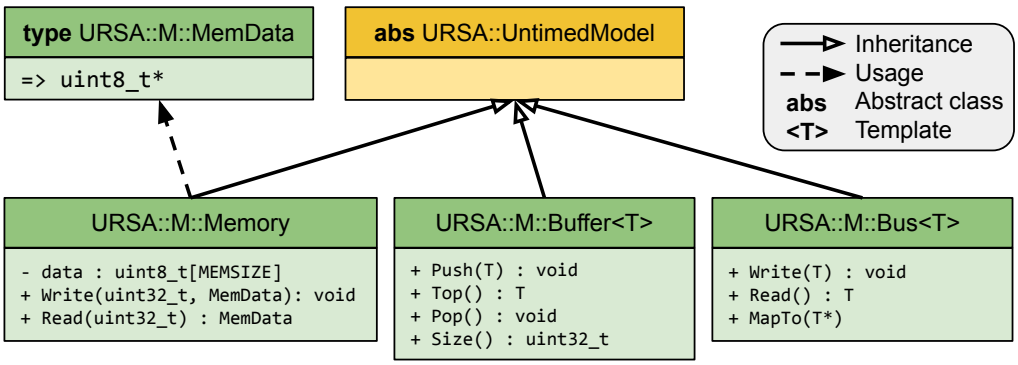

Figure 4.3 – A simplified class diagram representing the hierarchy and dependency among classes provided within the base model package. Important class fields are denoted by a "-" character, and important class methods are denoted by an "+" character. Other fields and classes are intentionally omitted.

- The **Memory** class models a generic RAM module, with operations for reading and writing data, wiping (zero fill), and save or load binary files to it. Memory modules implement a contiguous array of **MemData** elements along with variables for controlling memory mapping and illegal operations. MemData can be defined as any data type, and all memory operations will adjust to work with the new data type. In this work, memory data bus width is of four bytes due to it is the maximum amount of data that the HFRisc-V processor core can manipulate in the same cycle. The Memory class also has a method to map peripheral interfaces into memory regions so that they can be accessed via software.

- **Buffers** appear as parts of several hardware modules in ORCA. Although the modules have different buffer implementation (e.g., routers have their implementation of buffers), we assume the same buffer model for all these hardware, as we are more interested in the behavior of the hardware than in performance tweaks. Buffers are implemented as templates that can be specialized for a group of data types. The advantage of this approach is to have the same implementation being used all along the architecture

regardless the width of busses. In addition to that, buffers have facilities for detecting overflow, underflow, and special flags for full and empty checkings. We implement buffers as a wrapper on the STD::QUEUE class from the C++ standard library.

- In URSA, hardware models communicate to each other via busses, modeled by the **Bus** class, which has two operating modes. In VARIABLE mode, busses behavior becomes similar to the one of a variable in VHDL, where their values are available as soon as the driver process writes to it. In SIGNAL mode, busses admit a behavior similar to the one of a signal in VHDL, where their value is available only at the end of the current cycle. Depending on the purpose of the bus, one or another mode can be used, but modes cannot be changed during runtime, and only busses in signal mode can be mapped to memory spaces. An auxiliary method, MAPTO, is used to bind busses to regions in the memory space. These regions are generated by memory modules with the aid of GETMAP method.

4.4 ORCA-SIM, a simulator on top of URSA

We conducted simulation sessions using ORCA-SIM, a simulator made on top of URSA to simulate the ORCA MPSoC. We developed this tool by incorporating an (i) implementation of URSA whose features we mentioned previously in this chapter, an (ii) extended model package containing models for the hardware presented in Section 5, and a (iii) facade application to implement the simulation flow, instantiate models, and reporting.

We added four more classes to the package model. These classes represent the IPs of the platform: (i) DMANETIF, that models the network interface; (ii) HFRISCV, that models the HFRiscV processor core; (iii) ROUTER, that models the network router; (IV) and NETBRIDGE, that models models both the NBM and VEA. An overview of the class hierarchy of the extended model package considering only the newly added classes is shown in Figure 4.4. In the case of the HFRiscV model, we changed the existing simulator so that it could run as one of URSA models.

Four classes compose the facade application: TILE, PROCESSINGTILE, NETWORKINGTILE, and ORCASIM. Tile is the superclass that models a generic tile in the architecture. Two classes inherit from Tile: ProcessingTile, and NetworkingTile, representing a processing tile and an off-chip communication tile, respectively. The OrcaSim class is the application class that instantiates the tiles according to the configuration of the platform, connects routers, loads memory modules, and performs other startup routines. Figure 4.5 depicts the hierarchy of the facade application.

Since URSA provides the abstractions for discrete event simulation and hardware modeling, the remaining parts required for a complete simulator remain in the models them-

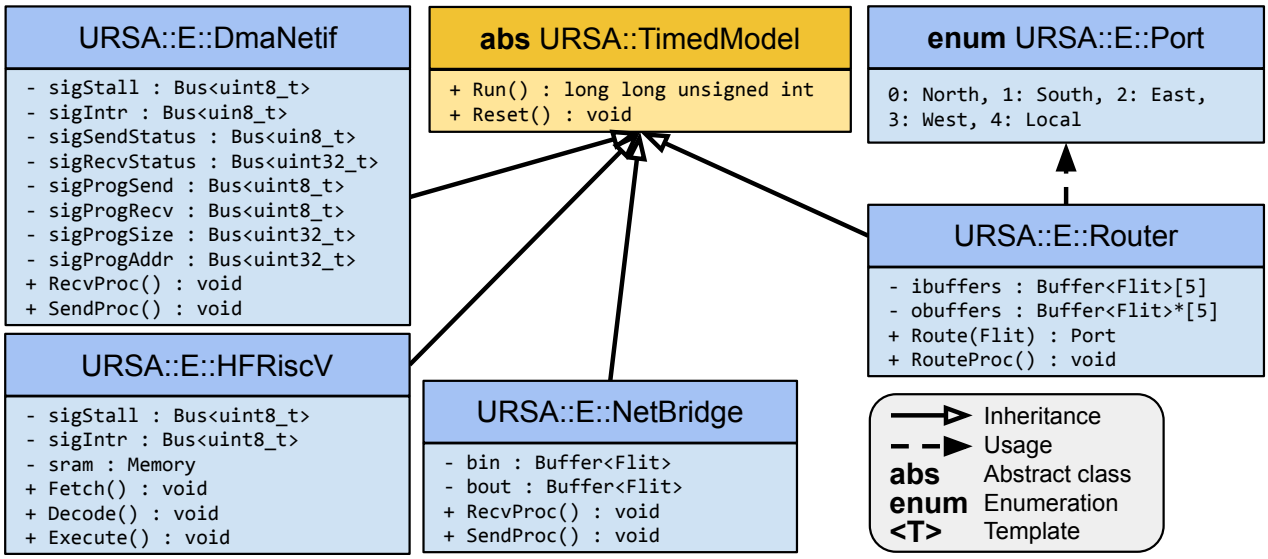

Figure 4.4 – A class diagram representing the hierarchy and dependency among classes added to the extended model package. Important class fields are denoted “-”, while methods are denoted by “+”. Some fields and methods are intentionally omitted.

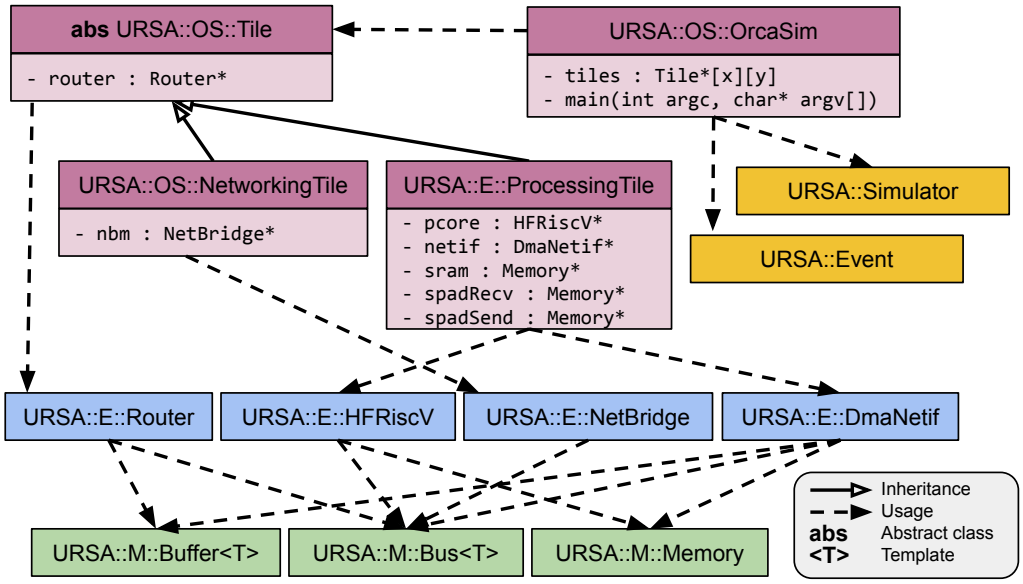

Figure 4.5 – A simplified class diagram representing the hierarchy and dependency among classes added in the facade application. Important class fields are denoted by a “-” character, and important class methods are denoted by an “+” character. Other fields and classes are intentionally omittedClass diagram for the facade of the simulator.

selves, plus some automation of the environment. Using URSA API, we developed models for each of the modules in ORCA: router, NI, processor core, memory cores, FIFO, network bridge, and virtual Ethernet controller. These models are reusable, that is, they can partake in other simulators build over URSA API. The interface of each of the models is as presented in Chapter 3, considering the following assumptions.

1. No hardware module has activity at the negative edge of the clock. Although supported by URSA, such a feature decreases the performance of the simulation. URSA currently supports two approaches for simulating in multiple edges. The first encompasses using two queues, one for simulating the positive edge and another one for the negative edge. In this case, queues would be swapped during the simulation. Another approach would be to schedule negative edge events together positive ones, using even time tags for one edge and odd time tags for another. In the first approach, switching queues would dramatically decrease the performance of the simulation. For the last one, the performance of the system would be the same, with the maximum number of cycles per epoch limited to half the allowed value.

2. Clock wires cannot be used within the logic of the module. The grain of abstraction used within URSA is one of a process, similar to the namesake construction in VHDL. A process sensitive to multiple clock signals must be reworked in two or more communicating processes. For instance, routers originally had multiple clocks for each of the adjacent buffers. We simplified the model by assuming a single clock, as in ORCA, the same clock expands to all hardware modules.

3. Signal driving must be defined at the design time and carried out manually by model programmers. Although URSA allows for consistency checking for multiple signal drivers at the runtime, such a feature decreases simulation performance. For this work, we disabled consistency checking as we have validated the hardware models before before performing the experiments presented in Chapter 7.

4. Signal values are stable until the driver overwrites them. Since signals' storages are emulated by variables, treatment for don't-care values would require the simulation engine to update signals when drivers have not assigned any value to it in the current cycle. Such a feature is supported by the platform, although we do not implement it since it would add a considerable processing overhead to the simulation, increasing simulation time.

5. HARDWARE COMPONENTS

In this chapter, we present the hardware components of ORCA MPSoC. The MPSoC comprises an arrangement of reusable tiles, which can be either off-chip communication tiles, presented in Section 5.3, or processing tiles, presented in Section 5.4. Tiles can connect through router modules, arranged in a mesh-topological network-on-chip, presented in Section 5.2. Our contributions include the design and implementation of all hardware modules presented in this chapter, excluding the HF-RiscV processor core. It is worth to note that although our router design follows the same behavior as Hermes' routers, we developed our router from scratch.

5.1 Top-Level Organization

ORCA MPSoC hardware architecture can be divided into two layers, where the topmost layer consists of a composition of intellectual property (IP) cores to form tiles, and the second one is tile interconnection. Two different tile compositions are herein presented: processing tiles, which correspond to usual PE in most MPSoC platforms, and off-chip tiles, which provide capabilities for other systems to communicate with the MPSoC. An instance of the MPSoC corresponds to a set of tiles organized in a mesh topology, connected by routers, forming a network-on-chip. In addition to their routers, a processing tile has memory, a processor core, and a network interface. For the main memory, a software image is loaded to it at the startup, containing hardware drivers, a real-time operating system, support libraries, communication middleware, and applications.

5.2 Networking Organization and Router Modules

ORCA MPSoC interconnection relies on Hermes [56]. We created a router model that resembles the design of Hermes routers. In Hermes, tiles are disposed in a bi-dimensional, rectangular-shaped, mesh-based topology, and connected by their on-chip routers. Routers can connect to up to four other routers outside the tile, attached to their NORTH, SOUTH, WEST, and EAST ports. A fifth port, the LOCAL port, is dedicated to hardware within the tile, providing an interface between the tile and the rest of the NoC. In this work, local ports of routers are either connected to a virtual Ethernet controller, in the case of an off-chip communication tile (see Section 5.3), or connected to network interfaces (NI), in case of a processing tile (Section 5.4). Besides, routers at the border of the NoC may have up to two

unconnected ports, which are grounded. An illustration of a network-on-chip in ORCA is shown in Figure 5.2.

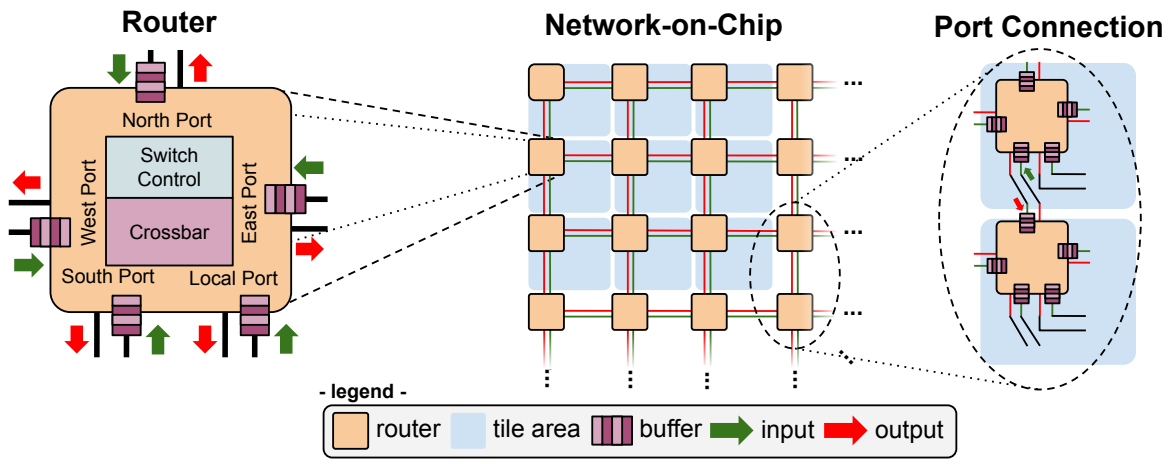

Figure 5.1 – Functional view of a router (left), an illustration of network-on-chip (middle), and the connection between two adjacent routers (right).

5.2.1 Routers

Routers communicate with each other by sending packets. A packet is a sequence (or burst) of flits, with a flit corresponding to a fixed-length string of bits, the smaller data chunk that can be transferred through the NoC. In ORCA, the length of a flit is configurable, although, for the sake of simplicity, we consider only a fixed width of 2 bytes (16 bits) in this work. Once a flit arrives in one port of a router, that flit gets stored in a buffer. As well as for flit width, the depth of buffers is configurable, and we assume a fixed depth of 16 flits in this work. Also, the number of routers in the NoC corresponds to the number of tiles, which allows for a theoretical value of 2^8 = 256 tiles (e.g., a 16x16 mesh). The maximum packet size corresponds to 2^{16} = 65536 flits, although we limit it to 64 flits.

The leading flit of a packet, the ADDRESS FLIT, stores the address of the destination router. When the value of the leading flit and the address of the receiver router are equals — routers are addressed at the design time following the physical addressing schema presented in Figure 5.2 —, the packet is delivered to the local port of that router. When the address differs from the address of the router, it gets routed to one of the other four ports. The router algorithm used in this work is XY, which is known to be deadlock-free [29]. In this algorithm, the packet is forwarded all the way in X-axis of the NoC until it reaches the same Y-coordinate of the destination router. Then, the packet is pushed towards the Y-axis until it reaches its final destination. The routing algorithm is executed once per packet, and consumes up to 4 cycles for the first flit, with the remaining flits following through the same port, one flit per cycle if the buffer at the receiver port is not full.

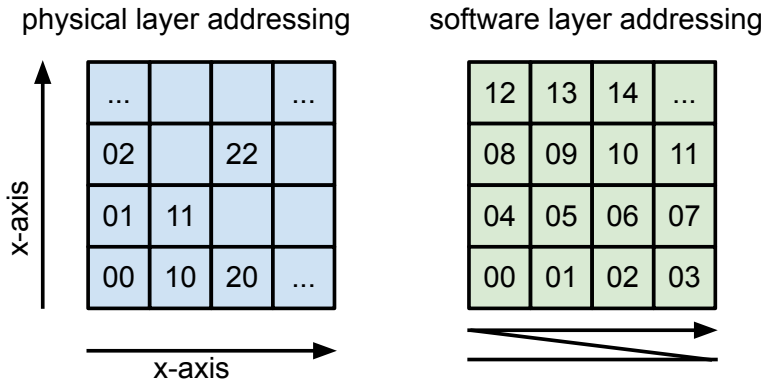

Figure 5.2 – Addressing system used within ORCA. Physical layer addressing (left) follows a pair (x,y) according to tile coordinate in the NoC. Software layer addressing (right) follows a sequential pattern.

The size of a packet is determined by the second flit of the packet, the SIZE FLIT. Since we use 2 bytes per flit, the theoretical maximum packet size is $2^{16} - 1 = 65535$ flits, although the number of flits of packets is limited in software to up to 64 flits, with the two first flits being necessarily the address and size flits. Since flits arrive at routers' ports in-order, and there is no deadlock in the network, packets are guaranteed to be eventually delivered, with bytes in-order. The wormhole strategy is used for package switching, where the packets' flits are sent one after another, without data interleaving. The router serves ports following a round-robin policy, one after another in a circular queue fashion, avoiding starvation. The switch control component performs both routing and packet switching, while the crossbar component establishes the connection between ports. The interface for a Hermes router module is shown in Figure 5.3.

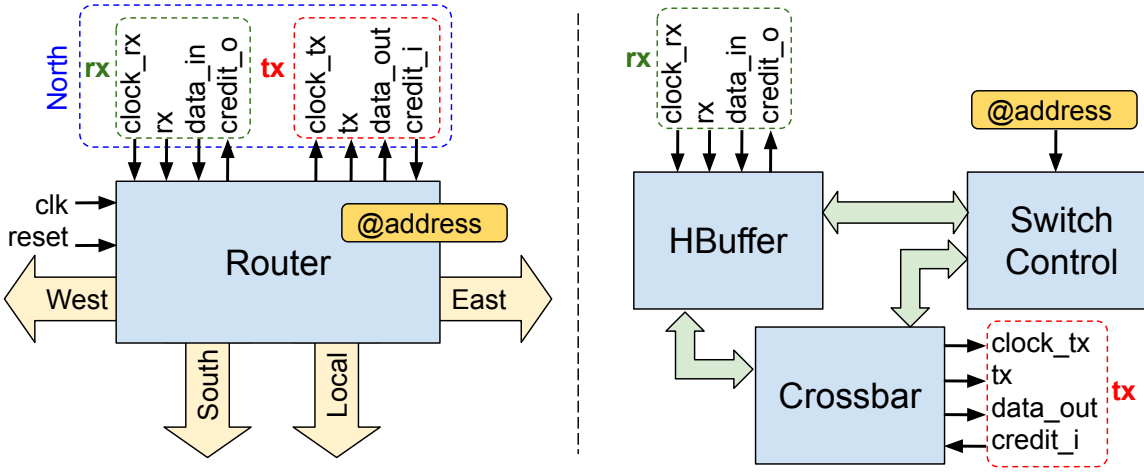

Figure 5.3 – A router (left) and its internal components (right).

The interface of buffers is the same for all ports. In summary, four signals control the receiving of new flits at each of the ports: CLOCK_RX, RX, DATA_IN, and CREDIT_O. Although Hermes supports multiple clock domains, we assume the same clock signal for all modules; thus, the signal clock_rx of all routers is bound to the global clock signal. The

signal RX (receive) raises when new data is available at DATA_IN. That data is copied into the memory of the buffer, where it resides until it becomes routed to another port. May the buffer becomes full, the CREDIT_O raises, preventing senders — whatever hardware is connected to the port — to inject new data to the buffer. It is important to note that these buffers were designed to work with Hermes' routers, and their interface is not the same as the other buffers in the system. For a comprehensive report on Hermes, please refer to Moraes et al. [56].

5.3 Off-chip Communication Tiles

An off-chip communication tile, shown in Figure 5.4 (left), is a reusable set of IPs that equip ORCA with capabilities for communicating with external UDP/IP networks through a virtual Ethernet adapter (VEA) and a network bridge module (NBM). As in processing tiles (see Section 5.4), the local port of the router is connected to hardware internal to the tile. For off-chip communication tiles, routers are always connected to an NBM.

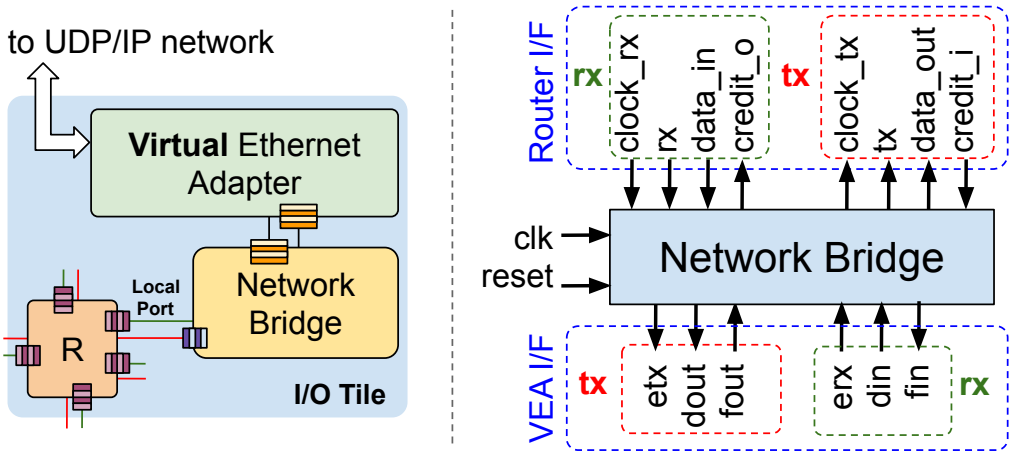

Figure 5.4 – Overview of an off-chip communication tile (left) and the interface of a network bridge module (right).

5.3.1 Network Bridge Module (NBM)

The NBM has two internal processes, one to translate packets incoming from the VEA to the NoC, and another one to translate packets in the opposite direction. A buffer similar to the one in routers provides access to the NoC, while another buffer of 32-bit width provides access to the VEA. Two state machines regulate the behavior of the module. The first, which translates NoC packets to UDP/IP, stacks flits that it receives from the NoC to form a UDP packet. Then, the process adds the proper UDP headers to the packet and

pushes it to the VEA. The second state machine, which translates UDP packets to NoC packets, removes UDP headers from packets and split the payload into flits. Then, these flits are pushed into the NoC one-by-one by the local router.

5.3.2 Virtual Ethernet Adapter (VEA)

In ORCASIM, VEA modules are emulated by a UDP socket in the host machine, thus available only when simulating the MPSoC using URSA. VEA implements two processes, one for receiving UDP packets, and one to send UDP packets, similarly to the NBM. A UDP server is instantiated for the former process, which pushes packets received from the UDP/IP network into a local buffer in chunks of 32 bits. This buffer is read by the NBM, which injects the data into the NoC. At the client-side, the VEA consumes a buffer connected to NBM to send packets to the UDP/IP network. The mindset behind having a module such as VEA is integrated the MPSoC with other systems, allowing for applications that would run part in the chip and part outside the chip. Also, the NBE interface can be reused when prototyping the system to an FPGA board, replacing the VEA by the communication method implemented by the board (e.g., I2C). An example of a distributed application that would favor from such a feature is given in Section 7.1.1.

5.4 Processing Tiles

Processing tiles provide computing power to ORCA MPSoC as they have the necessary hardware to run tasks, consisting of memory, processor core, and network interface (NI) modules. A NI orchestrates the functioning of a processing tile, serving as a gateway to the processor core. The memory access is handled by a custom multiplexer, whose activation depends on the value of the stall signal of the CPU, driven by the NI. When sending or receiving packets, the NI stalls the CPU, thus switching the memory access to the NI interface. Once done copying data, the NI releases the CPU while returning the memory access to the CPU interface. Figure 5.5 shows an overview of the organization of processing tiles.

5.4.1 Memory Core

For this work, we developed single-port memory cores, which have low area overhead when compared to dual-port cores. However, one drawback of having single-port cores in the design is the additional logic required by accessing protocols. In ORCA MPSoC, we use memory cores in two parts of the design. First, memory cores are attached

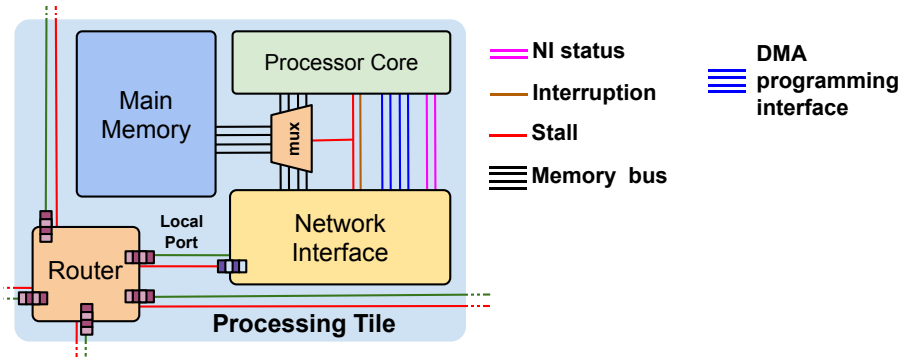

Figure 5.5 – Overview of a processing tile and its components.

to FIFO buffers, included into the NI modules. In these cases, there is no need for accessing protocols due to only FIFO buffers can read and write to these memories. The second appearance of memory cores corresponds to the main memory in processing tiles. In this case, both the processor core and the NI have access to the same memory module. The NI, which also has the role of DMA, coordinates the access protocol by configuring an attached multiplexer. An illustration of the interface of a memory core used within ORCA is shown in Figure 5.6.

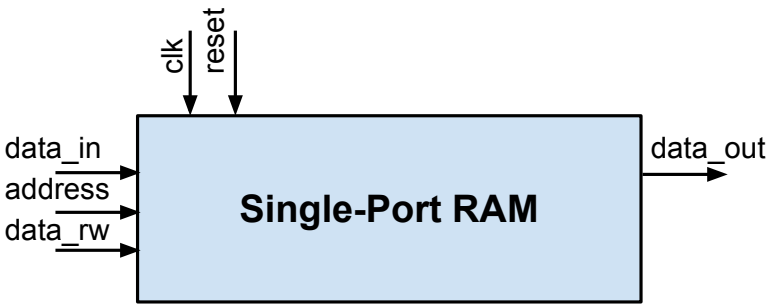

Figure 5.6 – Illustration of the single-port memory core used in ORCA.

The memory map for the main memory module is as shown in Figure 5.7. The organization of the memory space depends on the features defined at the design time for the platform, supporting (i) core, (ii) NoC, and (iii) monitoring extensions. The first extension corresponds to the memory map for a single core and is always enabled for any configuration. The system image containing software code and data is loaded to the memory at address 0x40000000, the origin address. Once started, the system will set the stack pointer to 0x40400000, the stack base address, increasing towards the origin address. The second extension, the NoC, presents the wires for controlling the network interface, enabling the core to send and receive packets through the network. We present the control signals for the network interface in Section 5.4.3. The last extension works in combination with the monitoring library, presented in Section 6.2.1.

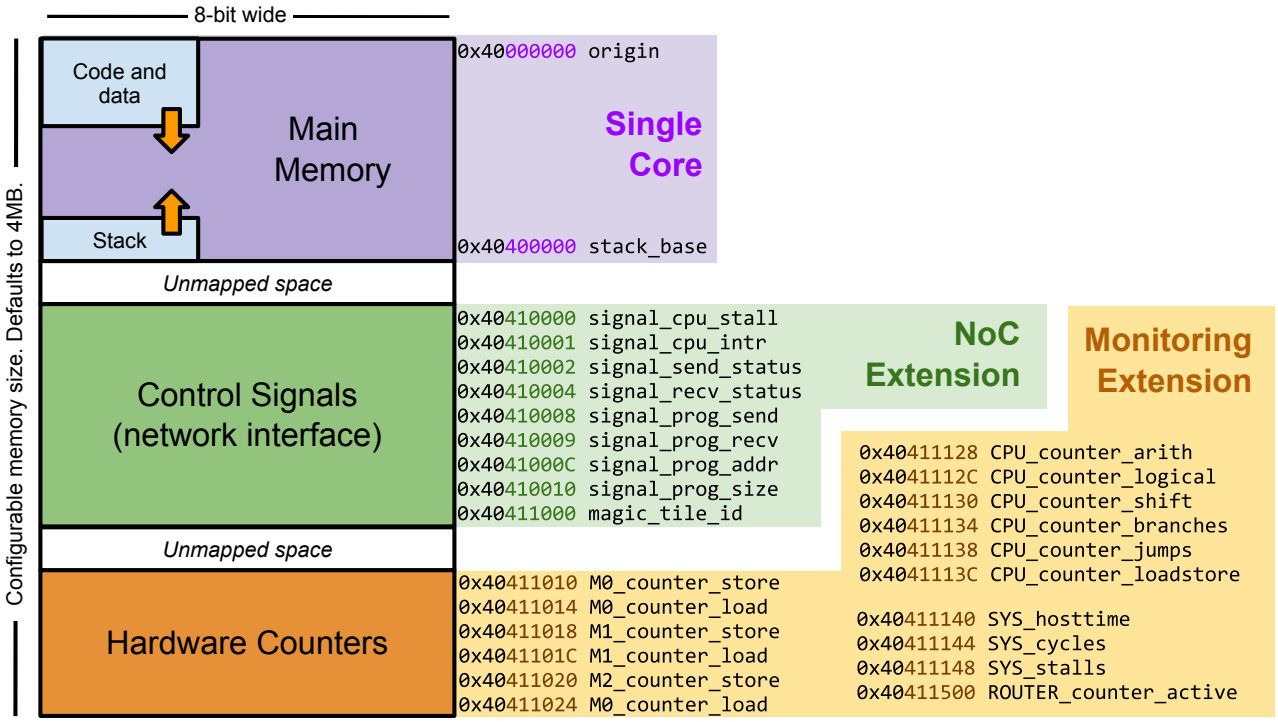

Figure 5.7 – Platform's memory map.

5.4.2 FIFO Buffers

FIFO (first-in, first-out) is an access model for memory buffers typically used to store temporary data, and are often called *queues*. We use FIFO buffers in situations where stored data must be processed in-order, that is, the first chunk of data to be written to the memory is the first to be read afterwards. In the case of ORCA MPSoC, buffers connect routers to network-interfaces, and routers to network-bridges. Although the implementation of FIFO buffers is nearly trivial, we briefly present the interface of our FIFO buffers for documenting purposes. It is important to note that these FIFO modules are not part of routers, as Hermes provides their queue design. Figure 5.8 shows the interface of the FIFO buffer module.

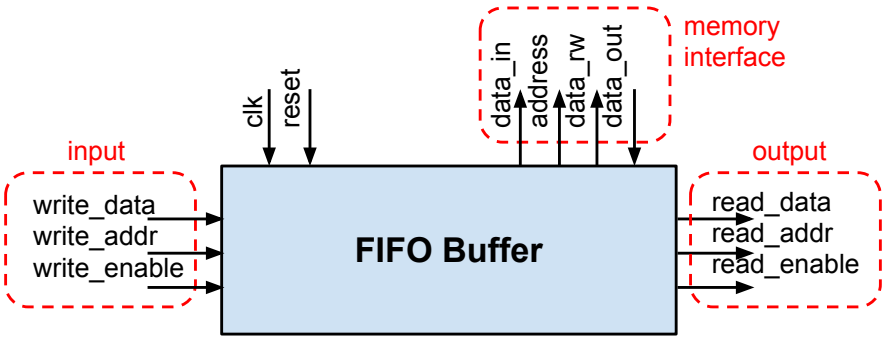

Figure 5.8 – Interface of a FIFO buffer.

5.4.3 Network Interface (NI)

The network interface (NI) module connects the router to other IPs in processing tiles, working as a gateway for the entire tile. To reduce the potential area of the chip, we opted to use one-port memory modules, which prevents the processor core from accessing the main memory at the same time as the NI. A peripheral driver orchestrates the operation of the NI, whose functioning is based on two processes: one for receiving packets, another one for sending packets. Different from the NI presented in Ruaro et al. [67], our NI does not require an arbiter process to work, as the processor core is unable to interact with both sending and receiving processes at the same time. Figure 5.9 shows the interface of the NI module and the state machines for the sending and receiving processes.

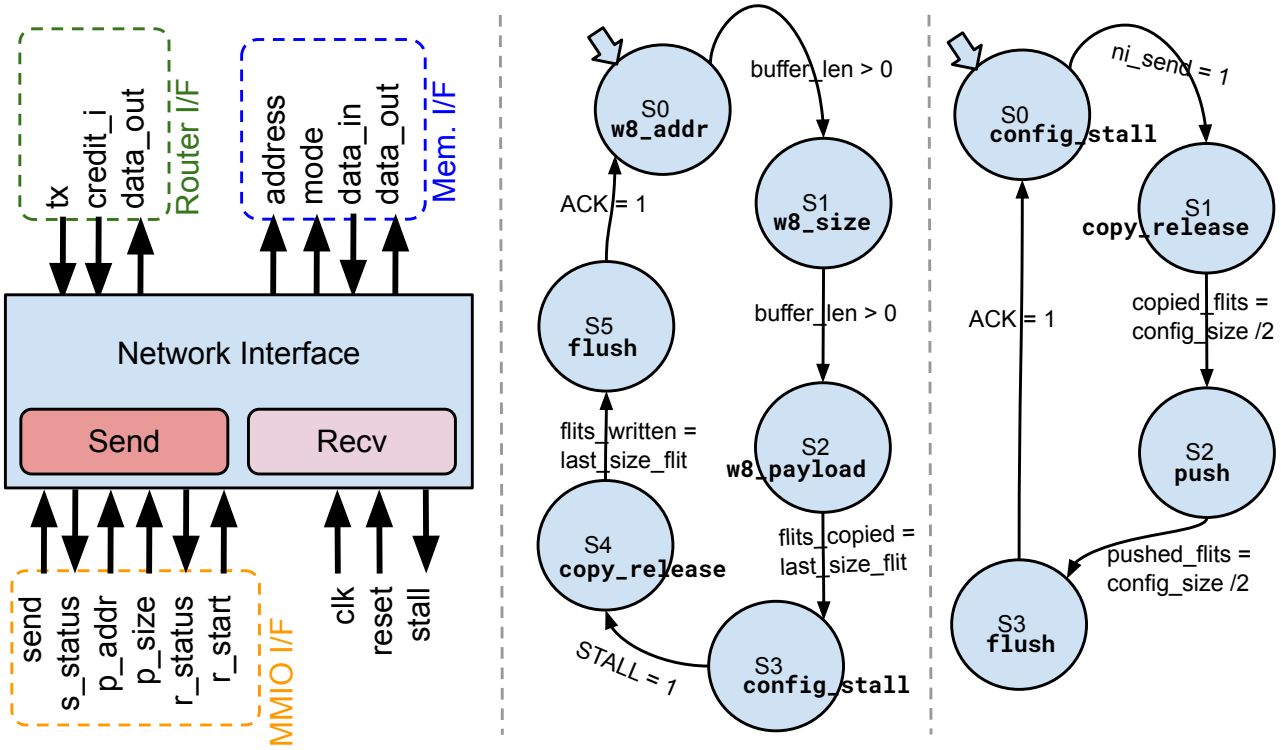

Figure 5.9 – NI interface (left), the state machine for the sending processes (middle), and the state machine for the receiving process (right).

When receiving a packet, the NI waits until all flits arrive for that packet (W8_ADDR, W8_SIZE, and W8_PAYLOAD). These flits e kept into the input buffer, which is an instance of the FIFO module presented in Section 5.4.2 and has the size of a network package. In this work, we use a packet size of 64 flits, although the size is configurable at the design time. Once all flits arrive at the buffer, the processor core is interrupted (CONFIG_STALL). Then, a peripheral driver configures the NI, which serves as a DMA module, to copy the packet into the main memory (COPY_RELEASE). The NI writes the number of received flits to the R_STATUS pin so that the processor core can notice the amount of memory necessary to

store the packet. When the last flit is copied, the NI releases the CPU, lowering the stall signal. Then, the NI wait for the CPU to acknowledge the operation (FLUSH), and then move to the initial state again (W8_ADDR).

When sending a packet, the processor core configures the NI to copy data from the main memory to the output buffer (CONFIG_STALL). Once configured, the NI stalls the CPU and copies the data, one flit per cycle, releasing the CPU when the last flit is copied (COPY_RELEASE). The NI waits for all flit to be injected into the network by the local router (PUSH). Once finished, it waits to lower the busy signal (FLUSH) and then proceeds to the initial state (CONFIG_STALL).

5.4.4 HFRiscV (processor core)

Our processor core model is an adaptation of an existing simulator for the HFRiscV processor core, `hfsim` [41]. That core implements either the RV32I (integer operations) or RV32IM ("M" extension for multiplication) instruction sets of the Risc-V user mode standard [81], with 32 user-level registers in the architecture (from $x04$ to $x31$), and four core instruction formats (R, I, S, and U-type). All registers are 32-bit wide, as is the length of all instructions. Instructions are submitted to a 3-stage pipeline with FETCH, DECODING, and EXECUTE stages. All instructions take 2 to 4 cycles to traverse the pipeline, where (i) branches not taken spend two cycles only, (ii) memory operations (reads and writes) take four cycles, and (iii) other operations take three cycles. A summary of the instructions format considering the instruction sets implemented in the HF-RiscV core are presented below.

R-Type Instructions comprises non-immediate arithmetic (ADD and SUB), logical (XOR, OR, AND), shifts (SLL, SLR, SLA), and compare instructions (SLT and SLTU). The "M" extension adds new instructions of the R-type format to support integer multiplication (MUL, MULH, MULHSU, MULHU, DIV, DIVU, REM, and REMU).

I-Type Instructions includes the immediate counterparts of R-type instructions (ADDI, XORI, ORI, ANDI, SLTI, SLTIU, SLLI, SLRI, SLAI, SLTI, and SLTIU), loads (LB, LH, LW, LBU, and LHU), synch (FENCE and FENCE.I), system (SYSCALL and SBREAK), and internal counters (RDCYCLE, RDCYCLEH, RDTIME, RDTIMEH, RDINSTRET, and RDINSTRETH) instructions.

S-Type Instructions are for stores (SB, SH, and SW). A variant, called B-type, includes branch instructions (BEQ, BNE, BLT, BGE, BLTU, and BGEU).

U-Type Instructions includes addressing instructions (LUI and AIUPC). A variant, J-type, correspond to the jump-and-link instructions (JAL and JALR).

The interface of the core, shown in Figure 5.10, exposes wiring for memory connection (ADDRESS, MODE, DATA_IN, and DATA_OUT), STALL (hold the cpu state when risen), and external I/O (EXTIO_I and EXTIO_O).

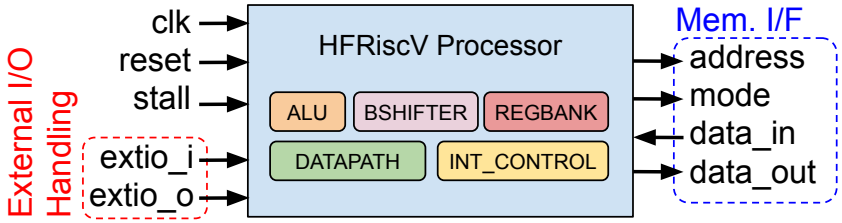

Figure 5.10 – Interface of HFRiscV processor core comprising memory interface and external I/O handling. Internal components include ALU, barrel shifter, register bank, datapath, and interruption controller.

5.4.5 Memory Multiplexer

In this work, the memory interface of the processor core is connected to a multiplexer, the MEMMUX. The activation signal of the multiplexer is the stall signal. When the stall signal rises, the memory access is transferred to the NI. When the stall lowers, the access is given back to the CPU. By using this mechanism, we avoid both the CPU and NI to simultaneously access the main memory. As the memory module has a single port, we connect the port directly to the multiplexer. Figure 5.11 shows the interface of the memory multiplexer.

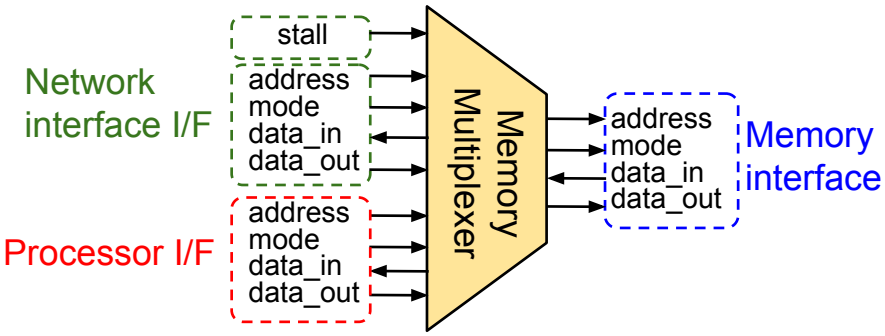

Figure 5.11 – Memory multiplexer and its interfaces with main memory, network interface, and processor core.

6. SOFTWARE COMPONENTS

In this chapter, we present the software components of ORCA, beginning with HellfireOS, the operating system running in each of the processing tiles of the platform, in Section 6.1. Two support libraries were developed in the context of this work. One library corresponds to the API for interacting with sensors, which we call ORCA Monitoring, presented in Section 6.2.1. The other one corresponds to an implementation of the publish-subscribe pattern presented in Section 6.2.2. Finally, a library for off-chip communication, the client library, is discussed in section 6.3. It is worth to mention that our contributions extend only to the software libraries, although we present minimal information on HellfireOS for documenting purposes.

6.1 HellfireOS

HellfireOS [40] is a preemptive, real-time operating system with support for dynamic, two-layer task scheduling. At the first layer, a real-time scheduler handles real-time tasks. The real-time scheduler can use either the earliest deadline first (EDF) or rate monotonic (RM) scheduling policies, configurable at the design time. In this work, we expressly adopt the EDF algorithm. In EDF, the scheduler takes the form of a priority queue, with tasks sorted by their deadline, from the earliest to the latest. The second layer of scheduling treats best-effort tasks, invoked only if the real-time scheduler has not consumed the whole CPU time. Hence, the execution of best-effort tasks cannot be guaranteed. Figure 6.1 shows the organization of HellfireOS.

HellfireOS software organization comprises five blocks: (i) hardware abstraction layer, (ii) kernel, (iii) device drivers, (iv) storage and networking systems, and (v) user tasks. The hardware abstraction layer (HAL) comprises the routines for machine initialization (boot up) and interruption management. At the kernel level, several data structures are provided: lists and queues, mutexes, semaphores, mathematic library, access to input/output (e.g., *printf*). Task management is also part of the kernel block. Device drivers correspond to the software for controlling the peripherals, including routines to deal with several hardware protocols, e.g., I2C, SPI, and the NoC. Storage and networking block is not used in this work, as we do not rely on the canonical OSI model [38] or data persistence.

Three key features make HellfireOS suitable for this work. The first regards the networking driver, which we could adapt to the hardware with just a few modifications, mostly related to the functioning of the network interface, and the driver had compatibility with Hermes' routers already. Second, the operating system offers support to Risc-V architecture. Third, HellfireOS supports real-time scheduling, which is a feature extensively mentioned in

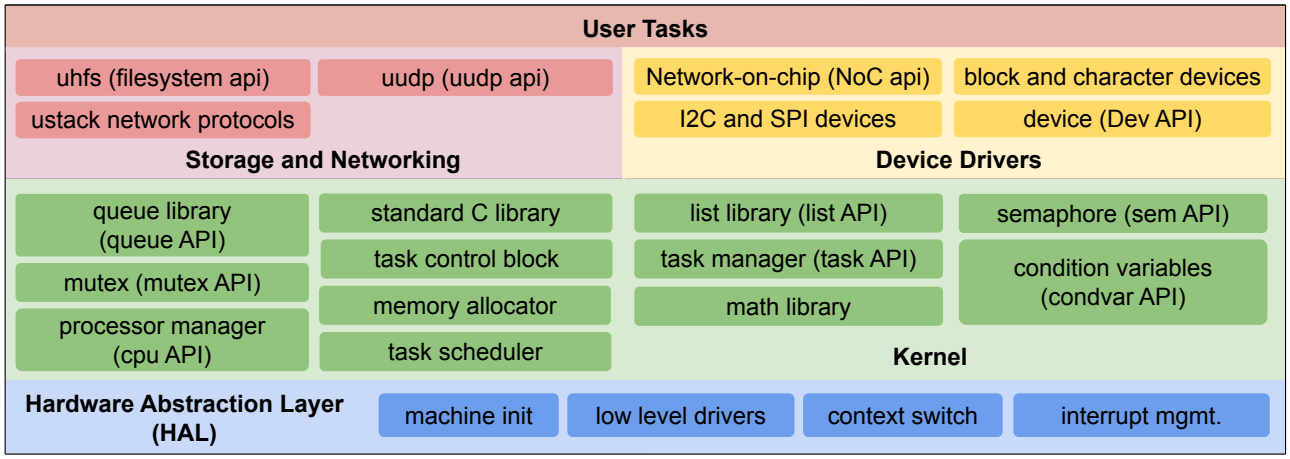

Figure 6.1 – Illustration of HellfireOS organization. The HAL provides low-level directives dealing with component-specific issues at the hardware level. The kernel itself provides support for the management of task and system-level API. Modules for data persistence (storage) and networking are provided as well. Lastly, device drivers provide protocol-level directives that may be accessed mostly by applications.

related works. Other benefits include a shipped-in library for fixed-point arithmetic, a very tiny standard library that reassembles the implementation of the C standard library, and routines for spawning and killing tasks, similar to those found in general-purpose operating systems [52].

6.2 Support Libraries

We developed two software libraries for HellfireOS. The former, ORCA Monitoring, provides access to hardware counters from the application-level, enabling the development of self-adaptive applications that can *sense*. The second library, ORCA Publish-Subscribe, allows processes to disseminate data through the system using an event-based strategy, the publish-subscribe model. All libraries were programmed in C language, and adapted to work with the HellfireOS kernel. We present each of these libraries below.

6.2.1 ORCA Monitoring

ORCA MPSoC is equipped with a couple of hardware counters. These counters can be either physical, when they do exist in the hardware design, or virtual when they are emulated by software. Regardless, all counters are exposed to the application level through the system API, which accesses reserved memory space to read from these counters via memory-mapped I/O, similarly to the approach used by Sarma et al. [74]. Physical sensors

correspond to specific registers that do not partake in components' functionality. Instead, the purpose of these registers is solely to store information on the performance of the system. Counters appear to the rest of the system as a memory region, accessible through software. Reads to that memory region returns the current value stored in the counter, and software can write to these counters to modify their value to zero, corresponding to a reset to the counter. Writing arbitrary values for debugging purposes is also permitted. Table 6.1 shows physical hardware counters available in ORCA.

Table 6.1 – Counters available in ORCA Monitoring.

Counter Alias	Description	Access	Availability
MEM_R(ψ, ϕ)	Returns the number of reading on memory ϕ of tile ψ since the last reset. Memory codes are the following: (0) main memory, (1) receiving memory, (2) sending memory.	System-wide	Hardware/ software
MEM_W(ψ, ϕ)	Returns the number of writes on memory ϕ of tile ψ since the last reset. Memory codes are the following: (0) main memory, (1) receiving memory, (2) sending memory.	System-wide	Hardware/ software
ROUTER_A(ψ)	Returns the number of cycles that the router of the tile ψ had being active since the last reset. Routers are active when at least one port is transmitting or receiveing data.	System-wide	Hardware/ software
CPU_INST(ψ, ϕ)	Returns the number of instructions that the CPU of the tile ψ executed since the last reset. Instructions are accounted per instruction class ϕ, which can be: arithmetic, logical, loads and stores, jumps, branches, and shifts.	System-wide	Hardware/ software
CPU_CYCLE(ψ)	Returns the number of cycles in that the CPU in tile ψ was not stalled since the last reset.	System-wide	Hardware/ software
CPU_STALL(ψ)	Returns the number of cycles in that CPU in tile ψ was stalled since the last reset.	System-wide	Hardware/ software
SYS_TIME(ψ)	Returns current timestamp of the system, converted to milliseconds. The calculi are based on the current frequency of the CPU and the cycle counter CPU_CYCLE(ψ).	Tile only	Software only
TSK_DLM(ψ, ϕ)	Returns the number of deadlines missed by task ϕ, running in CPU ψ, since the last reset.	Tile only	Software only
SYS_UTIL(ψ)	Returns the current percentage of utilization for CPU ψ. Cannot be reset.	Tile only	Software only
HST_TIME	Returns current timestamp of the host machine, converted to milliseconds. Depends on ORCASIM implementation and should be the same for all tiles.	Tile only	Simulation only

6.2.2 ORCA Publish-Subscribe

ORCA Publish-Subscribe is a software library for HellfireOS that enables the design of applications over the publish-subscribe pattern. We consider it a middleware as it interme-

diates the communication of the several tasks in the systems, sometimes requiring reduced effort *to design* these applications when compared to the message-passing model [18]. Regarding performance and resource usage, the approach has been used before in the context of MPSoCs, yet proved to add small memory and processing footprint to the system [31].

A few benefits come along with having a publish-subscribe system within ORCA, as the pattern provides space, time, and synchronization decoupling between communicant processes. With space decoupling, the address of communicant tasks becomes negligible at the application level, and engineers can design their applications despite the mapping of the tasks in the MPSoC, for example. For time decoupling, communication becomes unaffected by the temporary unavailability of one task or another since data can reside in the underlying network until it is delivered to the destination process. This feature is favored part by the NoC, which is capable of retaining data may a process fail to read it momentarily, and part by the kernel, which has a software-implemented buffer to stack packets until tasks could consume them. Finally, synchronization decoupling regards the elimination of communication blocking [32]. We present the operation implemented in ORCA publish-subscribe in Table 6.2. Three communicant parts participate in a typical publish-subscribe system: brokers, publishers, and subscribers. We discuss each of these parts below.

Brokers

Brokers are specialized tasks that orchestrate the communication of a publish-subscribe system. As so, we designed brokers so that they could be scheduled as if they were real-time tasks. Since the system uses an RT kernel, brokers can have a slice of CPU time dedicated to them. Another advantage of running brokers outside the kernel space is that any number of brokers can be spawned in the system — a useful feature if security is in question. Another advantage relates to fault-tolerance and resource management. When a broker fails, another broker can be spawned in any node of the system. For this work, we spawn brokers at the startup of the system, although we do not prevent tasks from spawning new brokers at any time. A similar approach has been used in other systems [19].

A broker task stays listening to a configurable port, to which messages of *advertise*, *unadvertise*, *subscribe*, and *unsubscribe* may arrive. Once a message arrives, the broker follows the behavior presented in Table 6.2. Broker tasks partake in all publish-subscribe operations except for publishing, although the broker keeps information on both publishers and subscribers of the system. The brokers maintain tables for storing information on publishers and subscribers. The publishers' table is updated by advertising and unadvertise operations, while the subscribers' table is updated on subscribe and unsubscribe operations.

Table 6.2 – Operations implemented in ORCA Publish-Subscribe.

Operation	Parameters	Behavior
pubsub_advertise	pubsub_node_info_t *pubinfo*, pubsub_node_info_t *brokerinfo*, topic_t *topic_name*	The publisher process sends a message to the broker (*brokerinfo*) informing that it (*pubinfo*) is going to publish to a topic (*topic_name*) in the near future. The broker stores the information in its internal tables and updates the publisher with the address of subscribers for that topic.
pubsub_unadvertise	pubsub_node_info_t *pubinfo*, pubsub_node_info_t *brokerinfo*, topic_t *topic_name*	The publisher process sends a message to the broker (*brokerinfo*) informing that it (*pubinfo*) ceased to publish to a topic (*topic_name*). The broker removes the corresponding entry from its internal publishers' table.
pubsub_publish	topic_t *topic*, int8_t* *content*, uint16_t *size*	The publisher process sends a message (*content*) of *size* bytes to each of the subscribers of a topic (*topic*). The list of subscribers is found in the publisher node, as the broker updates all publishers when a subscriber enters or leaves the system.
pubsub_subscribe	pubsub_node_info_t *subinfo*, pubsub_node_info_t *brokerinfo*, topic_t *topic*	The subscriber process (*subinfo*) sends a message to the broker (*brokerinfo*) requesting a subscription in a topic (*topic*). If the process is not subscribed to that topic already, the broker adds it to its subscribers table and looks up for publishers of that topic in the publishers table. Each publisher is updated with the information of the new subscriber, adding it to their publishers list.
pubsub_unsubscribe	pubsub_node_info_t *subinfo*, pubsub_node_info_t *brokerinfo*, topic_t *topic*	The subscriber process (*subinfo*) sends a message to the broker (*brokerinfo*) requesting its unsubscription from topic *topic*. If the process is subscribed to that topic, the broker removes it from its subscribers' table and looks up for publishers of that topic in the publishers' table. Each of the publishers is updated to remove the subscriber from their subscribers' list.

Publishers

As in Hameski's work [31], publishers store information about subscribers so that subscribers can receive messages directly from publishers. The main difference from a conventional centralized publish-subscribe system is that messages are not sent to brokers, reducing communication latency by shortening the number of hops in the network. However, the broker still has to keep the system consistent by storing proper information about publishers and subscribers. In the end, the cost of a publish operation, in terms of network hops, is not different than in conventional message-passing whatsoever.

Subscribers

Since the NoC's networking protocol has additional header fields to tag messages with channel information, we explore this feature in subscribers in a way that receiving a message from a publisher becomes the same as receiving any message via message-passing.

Hence, from an application perspective, messages transmitted using the publish-subscribe mechanism do not need to be treated separately from other messages, permitting applications to combine publish-subscribe with message-passing when convenient. The only overhead caused by the middleware is the subscription, which happens once for each subscribed topic and takes the same costs of sending an ordinary message via message-passing — it takes one message to perform any subscription.

6.3 Network Client Library

The network client library (NCL) provides an application-layer protocol, implemented over the HellfireOS communication API, to access the MPSoC nodes from outside the system. The library is useful for situation in which the MPSoC must integrate within an heterogeneous system, e.g. an FPGA board containing processors running a Linux system. In essence, the library is a driver for the off-communication module, as it is capable of translating the NoC protocol into another protocol. We used this library before to translate from the MPSoC protocol to UDP and vice-versa (see Section 7.1.1), yet it is possible to easily adapt the API to work with other protocols (e.g., I2C). We provide the API for the NCL in Table 6.3.

Table 6.3 – Operations implemented in the Network Client Library.

Operation	Parameters	Behavior
hf_send_open	std::string *addr*, uint32_t *port*	Opens a connection with the MPSoC, identified by a server address and port. The MPSoC to have a valid UDP/IP address within the network (requires one off-chip comm. tile). The opened connection can be used only to send data.
hf_send	uint16_t *target_cpu*, uint16_t *target_port*, int8_t* *buf*, uint16_t *size*, uint16_t *channel*	Sends a message to a node within the MPSoC. In HellfireOS, messages must be addressed to the a port, and can optionally be tagged with a channel number.
hf_send_close	std::string *addr*, uint32_t *port*	Closes a previously opened connection for a given address and port.
hf_recv_setup	std::string *addr*, uint32_t *port*	Opens a connection with the MPSoC, identified by a server address and port. The MPSoC must have a valid UDP/IP address within the network (requires one off-chip comm. tile). The opened connection can be used only to receive data.
hf_recv	uint16_t* *src_cpu*, uint16_t* *src_port*, int8_t* *buf*, uint16_t* *size*, uint16_t* *channel*	Receives a message from some node in the MPSoC. The receiving operation is blocking.
hf_recv_close	std::string *addr*, uint32_t *port*	Closes a previously opened connection for a given address and port.

7. EVALUATION

In this chapter, we present the evaluation of the proposed platform, both for the ORCA MPSoC, the hardware of the platform, and URSA, the simulation API used to simulate ORCA. In the first part of the chapter, we discuss the functional validation of the platform, presenting features mentioned in previous contributions (sections 7.1.1 and 7.1.2) along with an experiment for demonstrating task reallocation (Section 7.1.3). The second part of the chapter corresponds to the performance evaluation of ORCA-SIM, consisting of an experiment to measure the performance of the simulation engine for the single-core emulation of the Risc-V ISA (Section 7.2.1), and an experiment exploring the scalability of the simulation (Section 7.2.2).

7.1 Functional Validation

In the next three sections, we present the functional validation for the proposed platform, which consists of demonstrating some of the previously presented features, including off-chip communication, energy evaluation, and task reallocation. For the former two, we briefly summarize the contributions of two previously published papers [21, 79]. For the last one, we developed a minimal working example from scratch.

7.1.1 Integration with a Robotics Environment

In previous work [21], we presented the integration of the proposed platform with an environment for robotics simulation, relying on the off-chip communication tile introduced in Section 5.3. The contributions of the work included presenting the VEA module (see Section 5.3.2), wrappers for MPSoC protocols to work with the external system (based on the NCL, presented in Section 6.3), and an example application implementing a random-walk algorithm for a differential robot. We simulated the robot using Gazebo [60], a robotics simulator capable of simulating physics. The communication between the MPSoC and Gazebo was carried out by two applications developed over the ROS middleware [59]. An illustration of the discussed environment is shown in Figure 7.1. We learned from that work that ORCA-SIM could provide a cycle-accurate simulation environment while keeping pace with the simulation speed of Gazebo. We achieve preliminary results regarding the simulation performance, which in that case showed to be linear to the number of simulated tiles up to 49 tiles, as shown in Figure 7.2.

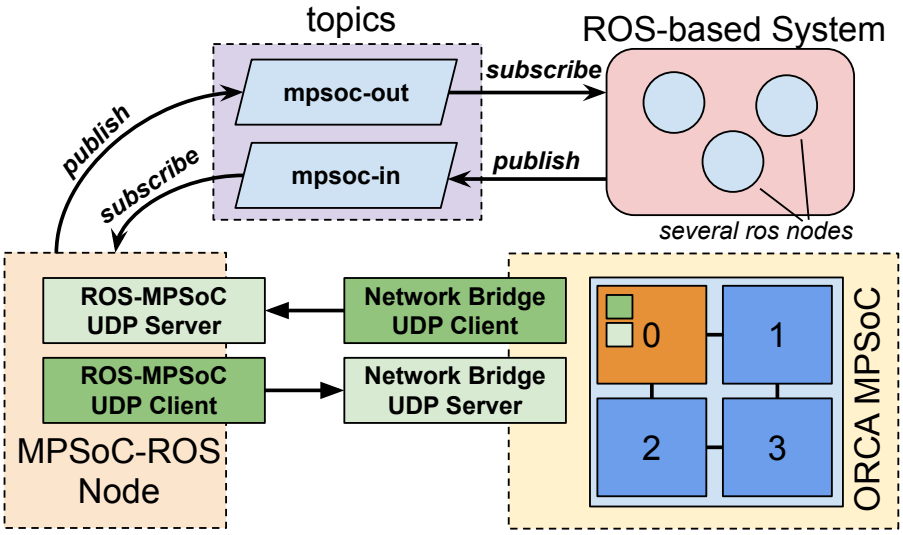

Figure 7.1 – Organization of the MPSoC integration into a ROS system. The MPSoC-ROS node exchanges data with the rest of the system by publishing to the mpsoc-out topic, as well as subscribing to the mpsoc-in topic. Arrows indicate the direction of the data-flow [21].

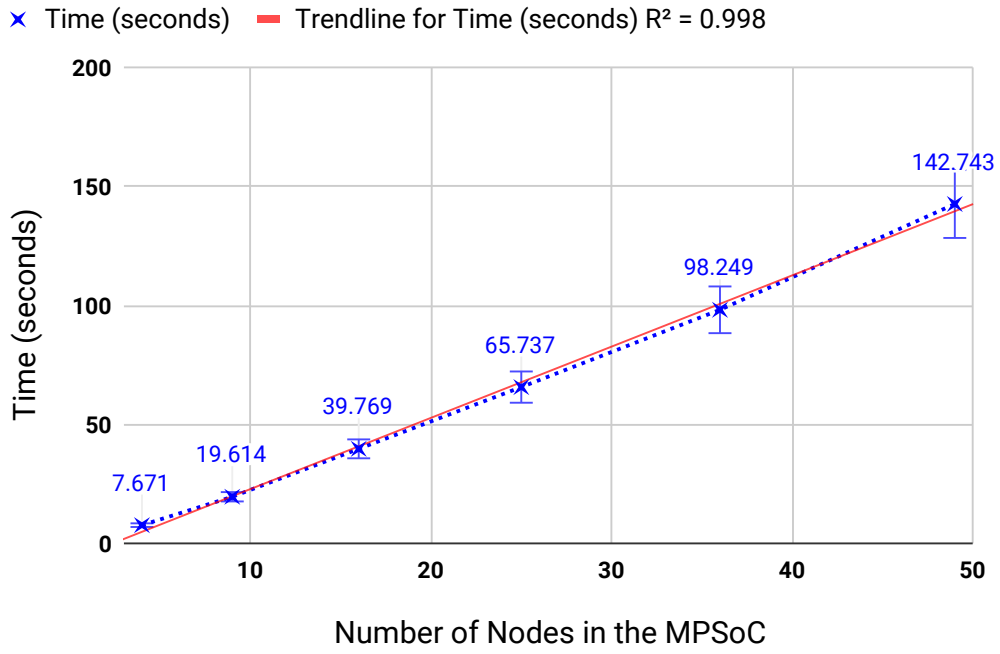

Figure 7.2 – Time taken to simulate 10M cycles for MPSoCs with 4, 9, 16, 25, 36, and 49 nodes. Results point that adding new nodes to the simulation makes the simulation time grows almost linearly to the number of nodes [21].

7.1.2 Energy Consumption Estimation

In recent work [79], we deliver several upgrades to the platform along with experiments for validating a couple of features. For the hardware architecture, we improved the processor model for the HF-RiscV core, adding new functionalities in URSA API to improve the access of memory-mapped peripherals. These changes had no impact on applications, although they dramatically increased simulation performance. Changes to the logic of buffers and router models were made as well. Finally, the network interface has been reworked, resulting in the model presented in Section 5.4.3. Minor changes have been performed on the simulation engine, contributing to the performance of the simulation.

The main contribution of that work is to demonstrate energy evaluation for a real-world robotic application. The platform provides all the requirements for energy evaluation. First, the hardware counters presented in Section 6.2.1 were used to estimate the power and energy consumption for several parts of the MPSoC, considering the approach proposed by Martins [48]. In that approach, hardware characterization must be done prior to the evaluation — see Martins [48, 49] work for a comprehensive discussion on the characterization process. The values for energy consumption obtained from characterization are stamped into the counters. The system periodically read from the sensors, summing the energy consumption for the several parts of the system, giving an estimation of the total consumption for the period. The operation is repeated over and over until the end of the execution of the system. It is worth to highlight that we did not perform characterization on the current hardware, and the contributions regarding energy estimation correspond only to the hardware counters and the calculi embedded to the platform. The user must insert the values for characterization, and we do not provide these values for our platform.

We applied the technique for a robotics application consisting of an extended Kalman filter (EKF) and a proportional integral derivative (PID) controller tasks. These tasks were

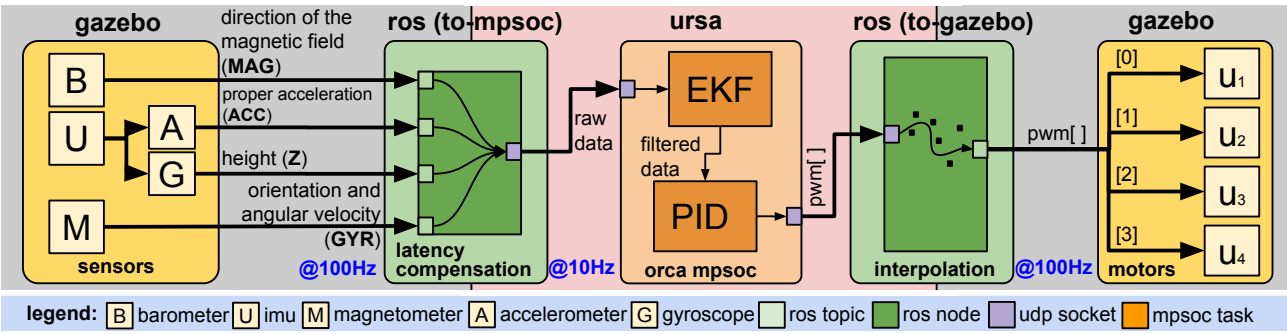

Figure 7.3 – Diagram representing the dataflow between the components of the proposed environment [79]. Yellow boxes represent gazebo, corresponding to simulation of sensors and motors. Green boxes represent the ROS system that encapsulate the MPSoC. The orange box represents ORCA-SIM, simulating both the EKF and PID tasks.

periodically fed by the sensors installed in an unmanned aerial vehicle (UAV), simulated in Gazebo [60], attached to a ROS [59] system, similarly to work [21] presented in Section 7.1.1. The EKF tasks were used to estimate the pose of the vehicle using the sensors' data, and the PID tasks were used to control the attitude and altitude of the vehicle.

As a result, we could feed the MPSoC with a data rate of 10Hz, with the platform simulating at 1.14MHz (real-time) for a 4x4 configuration (16 tiles). For instance, the platform used in Section 7.1.1 could run the same configuration for 200KHz only, nearly 500% slower. Figure 7.3 presents an illustration of the discussed system.

7.1.3 Task Reallocation with Software Sensing

In this section, we demonstrate a minimal working example for the task reallocation feature using real-time parameters of the target task. The platform is configured for a 2-by-3 system in which one of the tiles, namely tile zero, is an off-chip communication tile. The remaining tiles, numbered from one to five, are processing tiles running real-time tasks. The goal is to spawn new tasks in one of the processing tiles to force a specific task to miss deadlines. Once a configured number of deadlines is missed, the system will reallocate the task in another processing tile. We demonstrate such a feature using a producer-consumer application with the aid of the publisher-subscriber extension. For the sake of simplicity, the following assumptions hold.

1. The scheduling algorithm is Early Deadline First (EDF), which is *optimal*. Therefore, we assume the CPU to have utilization $\leq$ 100% when the task set is *schedulable*; that is, there is a possible execution order to avoid any task to miss its deadline.

2. The period is ten time units; that is, the task set is released every ten time units. When the sum of all capacities for the task set is $\geq$ 10, at least one task will miss a deadline.

3. All tasks have period of ten time units and deadline of one time unit (10% of CPU time).

4. All tasks have capacity and deadlines set to one unit of time; thus, the maximum number of tasks in a schedulable task set has to be equals to the period. Note that the restriction on the maximum number of simultaneous tasks is 32, although we do not explore it further.

5. Scheduling time is negligible and considered to be instantaneous (equals to zero).

At the system startup, no task is spawned yet. Once the kernel is ready, and all drivers and internal structures were initialized, a startup routine spawns the initial task set for each of the processing tiles. For this example, the following tasks compose the startup

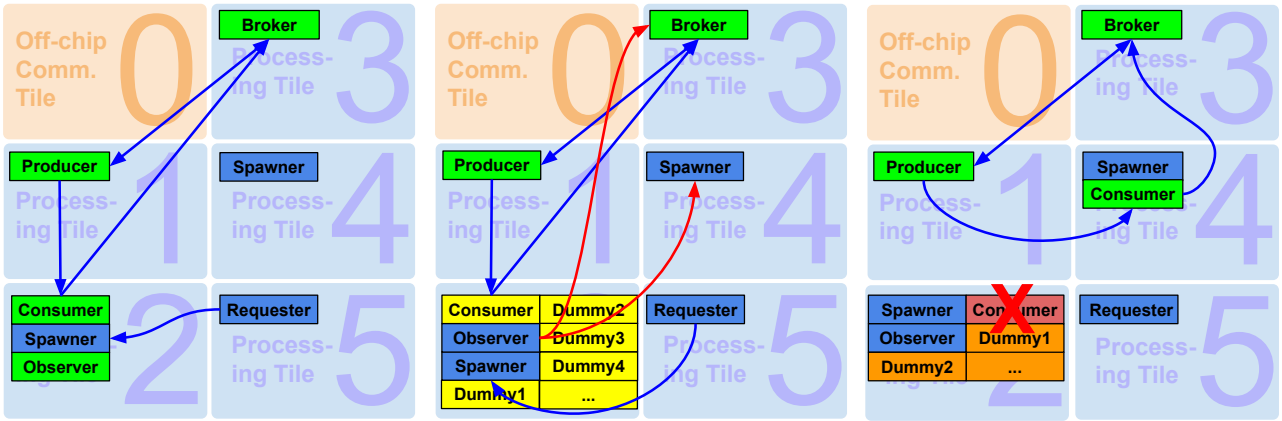

Figure 7.4 – Task set for each of the processing tiles, depicting initial allocation (a), deadline missing detection (b), and task reallocation (c). Arrows denote communication between tasks. Blue arrows are for continuous communication, and red arrows indicate that communication happens once. The color code for tasks is as follows: (green) healthy, (yellow) some deadlines missed, (orange) scheduling is unfeasible, (red) task removed from the task set.

configuration, as presented in Figure 7.4 (a). The organization of the system according to the taxonomy presented in Section 3.2 is shown in Figure 7.5.

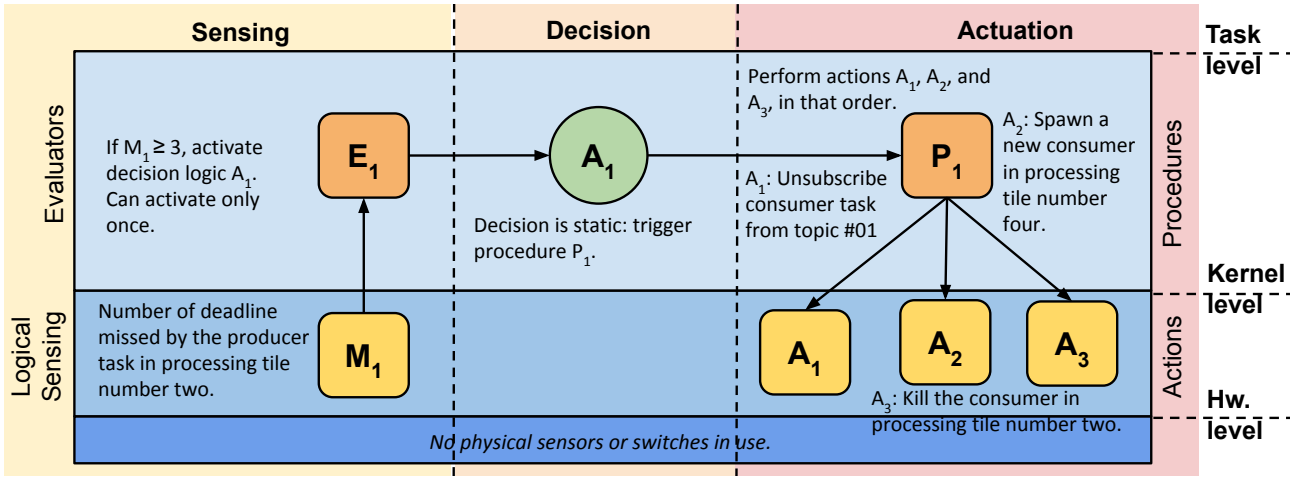

Figure 7.5 – Modeling of a self-adaptive technique that uses task reallocation.

1. A PRODUCER task is spawned to processing tile number one. These tasks produce packets containing a sequential number, and send these packets to the consumer task, spawned in tile two. The delay between packages is three milliseconds (3ms). Before sending the first packet, this task advertises to the system broker, informing that publication will be performed on topic #01.

2. SPAWNER tasks are deployed to processing tiles two and four. Spawner tasks can spawn tasks (including copies of itself) by invoking kernel calls in the node that they reside in. In this demonstration, we use one spawner task to create a couple of dummy

tasks, forcing a deadline in processing tile two; and another spawner task to reallocate one of the affected tasks.

3. A CONSUMER task is spawned to processing tile two. This task periodically receives packages from the consumer task, and print their content on the standard output (emulated UART). Once spawned, this task register to the system broker to subscribe to topic #01, the same topic to which the producer task advertises to.

4. A BROKER task is spawned to processing tile number three. The behavior of the task is as explained in Section 6.2.2.

5. An OBSERVER task resides in the same processing tile as the consumer. The observer task monitors the number of deadlines of the consumer task, and notifies the spawner in processing tile four in case the number of deadlines goes greater than a certain threshold. We use three deadlines as threshold for this demonstration.

6. A REQUESTER task is placed in processing tile number five. The requester task periodically asks the spawner in processing tile two to spawn new dummy tasks.

7. DUMMY tasks, implemented as empty loops, are spawned by the spawner task in node 4. Spawns occur every 100 milliseconds, with the first spawning occurring one second after the startup (time zero).

At the startup, the kernel spawns all tasks for the initial task set. The task set is schedulable and consumes at most 30% of CPU (worst case is processing tile number two). Both the producer and consumer tasks have been registered with the broker for publishing and subscribing to topic #01. At the time $100ms$, the producer task start to generate packets to the consumer tasks. New packets are generated once per $5ms$. At the time $1000ms$ (configured at the design time), the requester at tile number five starts requesting the spawner in tile number two to spawn new dummy tasks. In HellfireOS, the kernel can have up to 32 tasks running simultaneously. At most, 28 dummy tasks could be spawned to tile number two, as HellfireOS reserves one task slot for the idle task, a best-effort task that does not belong to the real-time task set, having no impact on the demonstration. Since the period of all tasks is ten units of time, the system can theoretically schedule the initial task set plus new six dummy tasks without missing any deadline. Once the seventh dummy task enters the system, the scheduler cannot guarantee that the deadline for the tasks will be met. Once the consumer task misses its third deadline (configured at the design time), the observer tasks notify both the broker (to unsubscribe the task from topic T0) and the spawner in tile four (to spawn a new consumer task). This process is depicted in Figure 7.4 (b).

Once the broker receives the notification from the observer, it updates the produces (which is the only publisher for topic #01) so that the topic will not generate new messages until a new subscriber enters the system. In meanwhile, the spawner in tile four is spawning a

new consumer task. The consumer registers at the broker as a subscriber of topic #01 once it starts. Then, the broker notifies the producer, which starts to push packets to the network once more. Since the consumer task has registered with the broker, the only destination of the new packets is the newly spawned consumer, as shown in Figure 7.4 (c).

With the demonstration, we learned that task reallocation depends on several features of the MPSoC. First, particular tasks for spawning and killing other tasks are necessary, as implementing them as part of the kernel would add additional overhead to the system. Also, these tasks can be added and removed from the system when necessary. Regarding the reallocation time, it will depend on the performance of the network. Since we do not transfer the context of tasks between tiles (otherwise it would be called *migration* instead of *reallocation*), only control messages traverse the network.

Future works on task reallocation include migrating the context of tasks between processing tiles. Other MPSoCs, e.g., HeMPs, use an external application repository so application code can be migrated as well. In ORCA, all processing tiles share the same kernel image, containing the same application code, so migrating the application code is not necessary. However, such a feature is desired when working with multiple ISA, e.g., Risc-V 32I and 32IM, where the former does not require a multiplier unit. In this case, the generated code would not be the same the multiple architectures.

7.2 Performance Evaluation

7.2.1 Single-Core Performance

Comparing simulators is a laborious task, as it includes the configuration of many tools. Also, it is a complex activity due to the available simulators include many parameters, and a single compilation flag may jeopardize the accuracy of the comparison. For these reasons, we do not compare ORCA (either the simulation engine or ORCA-SIM itself) against other simulators head-to-head; instead, we provide a minimal experiment to situate ORCA in the broad group of simulation tools, mainly discussing these groups in terms of goals, simulation performance, and accuracy.

We ran a benchmark for a few simulation tools so that we could get an insight on time that ORCA-SIM takes for simulating a single-core Risc-V architecture when compared to other tools. The benchmark consisted of two applications: one for a single empty loop implementation, and another one containing a bubble sort implementation. We chose these applications due to they do not depend on other software libraries, as we could not guarantee that the simulation tools would support these libraries. The same applies for input and output, as different architectures may map these peripheral in different memory regions. For each

pair application-simulator, we ran these applications five times, discarding the shortest and largest times for each application, taking the mean between the remaining values as the execution time for each pair. We performed the experiments on a Intel© Core™ i5-6500 at 3.20GHz CPU, with Ubuntu 16.04. We show the results of this benchmark in Figure 7.6.

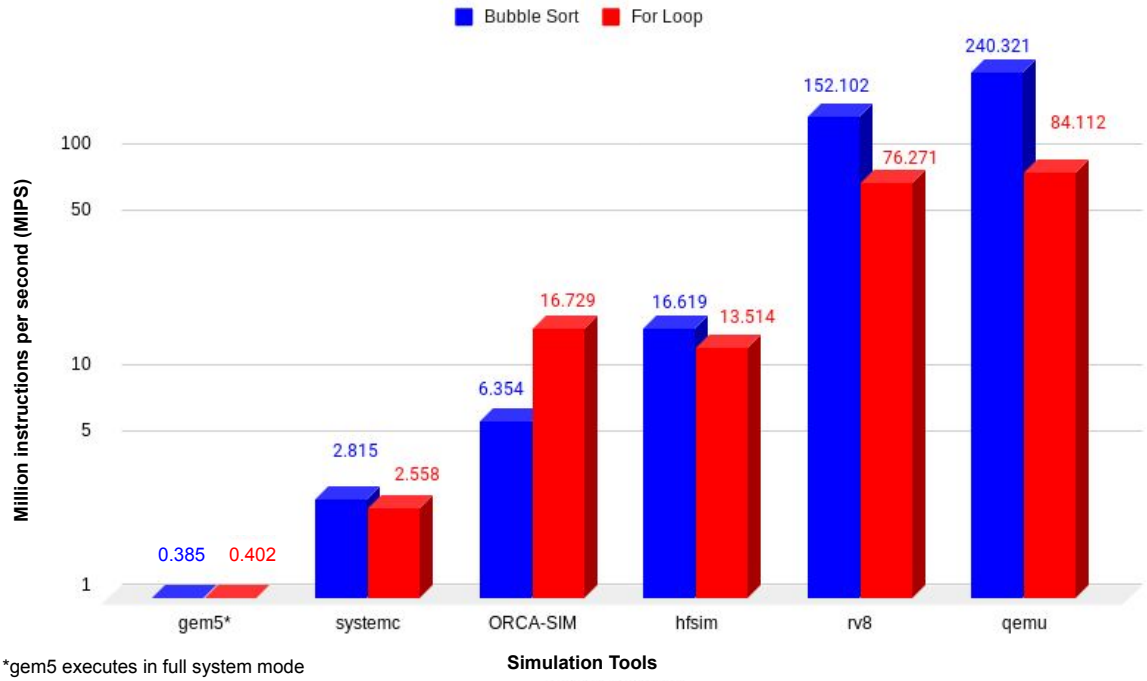

Figure 7.6 – Results for Bubble Sort and For Loop applications running in several simulators.

The results point out that ORCA-SIM performance lies between SystemC and hfsim. As expected, some performance degradation occurs due to the features added to the HFRisc-V processor model. Although we disabled some of these features for the experiment[1], ORCA-SIM could not achieve the same performance of hfsim for the Bubble Sort application. For the For Loop application, ORCA-SIM could perform better due to the optimizations to the memory access implemented in the simulator. Both ORCA-SIM performs better than SystemC as it ignores the internal state of the hardware. For comparison purposes, we considered gem5 [65], rv8 [13], and qemu [12] simulators. For gem5, running in full system mode — which considers the internal state of the hardware —, presented the worst performance. The other two tools use binary translation to achieve near-native speed. As future work, we intend to explore binary translation to achieve near-native simulation speed. However, it is important to note that binary translation is a technique for translating non-native instruction to native instruction. Hence, it cannot provide cycle-accurate information on the simulation. Another future work includes using a more elaborated benchmark to compare architectures, e.g., CoreMark™ [70].

[1] For the configuration of the simulators and applications used in the experiment, see https://github.com/andersondomingues/ursa-benchmark.

7.2.2 Scalability and Multi-core Performance

The performance of the simulation depends on the specs of the host machine, as well as the applications being simulated as the processor core have a different execution for each instruction. We run this experiment in a Dell Precision Tower 3420 with four Intel(R) Xeon(R) CPU E3-1220 v5 @ 3.00GHz, 32GiB of RAM, running Debian 10 (Buster) with Linux kernel version 4.19.0-6 machine. The simulator was compiled with GCC 8.3 (targeting x86_64), with optimizations (-O3, -MARCH, -MTUNE). We run ORCA-SIM five times for each of the tested configurations, discarding the worst and best value, considering the mean value of the remaining readings as the execution time. All configuration includes one off-chip communication tile, plus one or more processing tiles, connected to the NoC. The same application is loaded to all tiles, consisting of the bare-bone Bubble Sort application used in the experiment presented in Section 7.2.1. Figure 7.7 shows the results for the experiment.

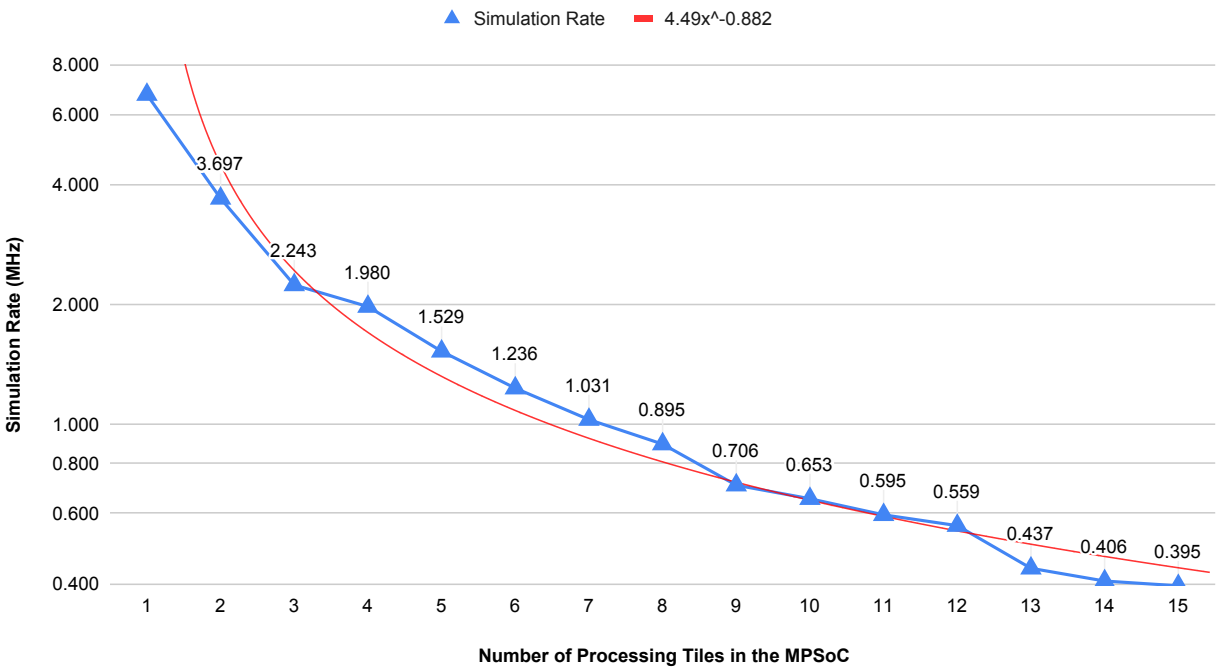

Figure 7.7 – Results for the experiment on the number of cores in the platforms. The trend line for the power series approximation is displayed in red color.

The results point out that the simulation performance is degraded as we add more processing tiles to the MPSoC. For a 2x2 mesh configuration (one off-chip, plus three processing tiles), the simulation speed reaches nearly 2MHz (baseline is host time). Such a result enables ORCA-SIM to be used as an emulator when considering that the simulation rate approximates the clock frequency of the emulated hardware [21, 79]. When working with more than ten nodes, the simulation rate drops below 5KHz, which seems to be imprac-

ticable for system emulation, even for embedded systems, which tend to operate in lower clock frequencies when compared to general-purpose systems.

Future works on URSA includes optimizing the simulation queue and rework the API to allow more modeling assets. For the simulation queue, we expect to replace the priory queue from the standard template library (STL) for one handcrafted specially for URSA, due to we observed during the experiments that most of the simulation time is consumed by managing the queue. By doing this, we expect to decrease the simulation time and allowing our API to be portable to other languages, as the priority queue would be part of it. For the API itself, we intend to create abstract classes representing generic modules such as processor cores, allowing the platform to operate with other ISA families. In the future, we also intend to explore binary translation so that we can configure ORCA-SIM either for both cycle-accurate simulation and emulation (similarly to qemu).

8. CONCLUSIONS AND FUTURE WORK

This work presented ORCA, a platform to support the development of self-adaptive MPSoCs. The platform comprises an MPSoC architecture, simulator, software libraries, and a framework for self-adaptation. The MPSoC combines Hermes NoC with HFRisc-V processor cores to form a programmable NORMA system. The processing nodes of the system run instances of the HellfireOS, a preemptive and real-time kernel. We developed two software libraries to support the development of self-adaptive systems; the first provides support for monitoring counters and sensors in the system for both software and hardware components, and the second library implements the publish-subscribe pattern for on-chip communication.

We validated the platform for an example of a self-adaptive technique. In that technique, monitors watch the number of deadline misses for a synthetic application, and activates task reallocation, moving the application to a neighbor processor core. The technique uses both software libraries. The publish-subscribe library is used for application communication while using the monitoring library is used for capturing the number of deadlines misses for the target application. Finally, the results presented in this thesis were collected using ORCA-SIM, a tool that simulates the ORCA MPSoC. We developed ORCA-SIM on top of URSA, an API for simulating computing systems.

We could successfully model and simulate several MPSoC configurations using URSA's API and implementation. In the future, we intend to add more features to the simulator (e.g., distributed simulation). Although the focus of URSA is to support the simulation of the proposed platform, other platforms can be simulated as well, as long as their models conform to URSA's API. URSA project and platforms' hardware models are available through GNU GPL v2 licensing. All associated files and documentation can be found at URSA's GitHub repository (https://github.com/andersondomingues/ursa).

8.1 Author's Words and Research Outlook

Self-adaptive systems have found their way to the world of MPSoCs, with several studies being published in the last twenty years. The trend is to have more and more similar studies out there. In this context, having a development platform to test and validate self-adaptive techniques is of great value. In this thesis, we present a platform that was made totally on open-source technology, and relies on the fresh recently-proposed Risc-V ISA. Both features make ORCA a powerful tool for the development of new self-adaptive techniques.

However, there is much work to be done before ORCA gets all the features necessary to comply with the vast range of self-adaptive techniques developed in the last two decades. We plan to add new features to the platform so that we can reach a broader number of techniques. It is important to highlight that although we developed ORCA having in mind a couple of other self-adaptive platforms, the research to come will shape the next versions of the platform. Below we enlist a couple of directions for further improvement of the platform.

- Debugging software is a time consuming-activity. In this work, we could not set up the proper tooling for dealing with debugging, and no existing solution could be seamlessly adapted to work for this purpose. Thus, we plan to **extend ORCASIM to support the GDB Remote Serial Protocol (RSP)** [26] for remote debugging, permitting one to inspect the state of each one of the processors during the runtime.

- Ongoing research on autonomous vehicles uses ORCA to control a quadrotor. The software running in the MPSoC consists of two tasks: an extended Kallman filter (EKF) and a proportional–integral–derivative (PID) controller. Both tasks are very time-sensitive, making the ORCA MPSoC a suitable architecture to run them. Although we could validate the software using ORCASIM, **prototyping ORCA to an FPGA board** is on schedule. The FPGA will be attached to real quadrotor vehicle.

- We will add more sensing and actuation capabilities to the platform. The platform currently supports DVFS, but hardware modules have no DVFS zones defined. **We plan to define DVFS zones for each of the hardware modules**. Another interesting feature is task migration. ORCA currently supports task reallocation, which roughly compares to kill a task in one processor and starting it in another one, losing all application data. **We intend to add task migration**, that is, the system will transfer applications' data (stack, heap, and registers) between cores.

REFERENCES

[1] Accellera Systems Initiative. “SystemC”. Source: https://www.accellera.org/downloads/standards/systemc, November 2019.

[2] Arcaini, P.; Riccobene, E.; Scandurra, P. “Modeling and Analyzing MAPE-K Feedback Loops for Self-Adaptation”. In: IEEE/ACM International Symposium on Software Engineering for Adaptive and Self-Managing Systems, 2015, pp. 13–23.

[3] Becker, J.; Brändle, K.; Brinkschulte, U.; Henkel, J.; Karl, W.; Köster, T.; Wenz, M.; Wörn, H. “Digital On-Demand Computing Organism for Real- Time Systems”. In: International Conference on Architecture of Computing Systems, 2006, pp. 17.

[4] Binkert, N.; Beckmann, B.; Black, G.; Reinhardt, S. K.; Saidi, A.; Basu, A.; Hestness, J.; Hower, D. R.; Krishna, T.; Sardashti, S. “The Gem5 Simulator”, *SIGARCH Computer Architecture News*, vol. 39–2, May 2011, pp. 1–7.

[5] Bouajila, A.; Bernauer, A.; Herkersdorf, A.; Rosenstiel, W.; Bringmann, O.; Stechele, W. “Error Detection Techniques Applicable in an Architecture Framework and Design Methodology for Autonomic SoCs”. In: Biologically Inspired Cooperative Computing, 2006, pp. 107–113.

[6] Brigham Young University. “JHDL: FPGA CAD TOOLS – Brian Young University”. Source: http://www.jhdl.org/, February 2020.

[7] Brinkschulte, U.; Becker, J.; Ungerer, T. “CARUSO - An Approach towards a Network of Low Power Autonomic Systems on Chips for Embedded Real-Time Applications”. In: International Parallel and Distributed Processing Symposium, 2004, pp. 124–129.

[8] Brun, Y.; Di Marzo Serugendo, G.; Gacek, C.; Giese, H.; Kienle, H.; Litoiu, M.; Müller, H.; Pezzè, M.; Shaw, M. “Engineering Self-Adaptive Systems through Feedback Loops”. Springer Berlin Heidelberg, 2009, chap. 3, pp. 48–70.

[9] Caimi, L. L.; Fochi, V.; Wachter, E.; Moraes, F. G. “Runtime Creation of Continuous Secure Zones in Many-Core Systems for Secure Applications”. In: IEEE Latin American Symposium on Circuits Systems, 2018, pp. 1–4.

[10] Carara, E. A.; de Oliveira, R. P.; Calazans, N. L. V.; Moraes, F. G. “HeMPS — A Framework for NoC-based MPSoC generation”. In: IEEE International Symposium on Circuits and Systems, 2009, pp. 1345–1348.

[11] Castilhos, G.; Mandelli, M.; Madalozzo, G.; Moraes, F. “Distributed Resource Management in NoC-based MPSoCs with Dynamic Cluster Sizes”. In: IEEE Computer Society Annual Symposium on Very Large Scale Integration, 2013, pp. 153–158.

[12] Clark, M. J. “QEMU with RISC-V (RV64G, RV32G) Emulation Support”. Source: https://github.com/michaeljclark/riscv-qemu, February 2020.

[13] Clark, M. J. “rv8 | RISC-V simulator for x86-64”. Source: https://rv8.io, February 2020.

[14] Condon, S. “NVIDIA unveils Orin, its Next-Gen SoC for Autonomous Vehicles and Robots”. Source: https://www.zdnet.com/article/nvidia-unveils-orin-its-next-gen-soc-for-autonomous-vehicles-and-robots, February 2020.

[15] DeBenedictis, E. P. “It’s Time to Redefine Moore’s Law Again”, *Computer*, vol. 50–2, Feb 2017, pp. 72–75.

[16] Dey, S.; Singh, A. K.; Wang, X.; McDonald-Maier, K. D. “DeadPool: Performance Deadline Based Frequency Pooling and Thermal Management Agent in DVFS Enabled MPSoCs”. In: IEEE International Conference on Cyber Security and Cloud Computing / IEEE International Conference on Edge Computing and Scalable Cloud, 2019, pp. 190–195.

[17] Diguet, J.-P. “Self-Adaptive Network On Chips”. In: Symposium on Integrated Circuits and Systems Design, 2014, pp. 24:1–24:6.

[18] Dollimore, J.; Kindberg, T.; Coulouris, G. “Distributed Systems: Concepts and Design”. Addison-Wesley, 2005, 5th ed., 644p.

[19] Domingues, A. R. P.; Hamerski, J. C.; Amory, A. “Broker Fault Recovery for a Multiprocessor System-an-Chip Middleware”. In: Symposium on Integrated Circuits and Systems Design, 2018, pp. 1–6.

[20] Domingues, A. R. P.; Hamerski, J. C.; de M. Amory, A. “A Fault Recovery Protocol for Brokers in Centralized Publish-Subscribe Systems Targeting Multiprocessor Systems-on-Chips”, *Analog Integrated Circuits and Signal Processing*, vol. 3–20, Mar 2020, pp. 1–29.

[21] Domingues, A. R. P.; Jurak, D. A.; Filho, S. J.; Amory, A. M. “Integrating an MPSoC to a Robotics Environment”. In: IEEE Latin American Robotics Symposium, 2019, pp. 4.

[22] Ebi, T.; Kramer, D.; Schuck, C.; von Renteln, A.; Becker, J.; Brinkschulte, U.; Henkel, J.; Karl, W. “DodOrg—A Self-adaptive Organic Many-core Architecture”. Springer Basel, 2011, chap. 4, pp. 353–368.

[23] Fishman, G. S. “Discrete-Event Simulation – Modeling, Programming and Analysis”. Springer Science+Business Media New York, 2001, 1st ed., 554p.

[24] Fochi, V.; Caimi, L. L.; da Silva, M. H.; Moraes, F. G. “Fault-Tolerance at the Management Level in Many-Core Systems”. In: Symposium on Integrated Circuits and Systems Design, 2018, pp. 1–6.

[25] Fowler, M. “What I Talk About When I Talk About Platforms”. Source: https://martinfowler.com/articles/talk-about-platforms.html, February 2020.

[26] Free Software Foundation. “Debugging with GDB - Protocol”. Source: https://ftp.gnu.org/old-gnu/Manuals/gdb/html_node/gdb_129.html, January 2020.

[27] Gamma, E.; Helm, R.; Johnson, R.; Vlissides, J. “Design Patterns: Elements of Reusable Object-Oriented Software”. Addison-Wesley, 1994, 1st ed., 590p.

[28] Gerstlauer, A. “The Spec-C System”. Source: http://www.cecs.uci.edu/~specc/, February 2020.

[29] Glass, C. J.; Ni, L. M. “The Turn Model for Adaptive Routing”. In: Annual International Symposium on Computer Architecture, 1992, pp. 278–287.

[30] Guang, L.; Plosila, J.; Isoaho, J.; Tenhunen, H. “HAMSoC : A Monitoring-Centric Design Approach for Adaptive Parallel Computing”. CRC Press, 2011, chap. 6, pp. 135–164.

[31] Hamerski, J. “Support to Run-Time Adaptation by a Publish-Subscribe Based Middleware for MPSoC Architectures”, Ph.D. Thesis, Pontifícia Universidade Católica do Rio Grande do Sul, Porto Alegre, Brazil, 2019, 80p.

[32] Hamerski, J. C.; Abich, G.; Reis, R.; Ost, L.; Amory, A. “Publish-Subscribe Programming for a NoC-based Multiprocessor System-on-Chip”. In: IEEE International Symposium on Circuits and Systems, 2017, pp. 1–4.

[33] Hamerski, J. C.; Domingues, A. R. P.; Moraes, F. G.; Amory, A. “Evaluating Serialization for a Publish-Subscribe Based Middleware for MPSoCs”. In: IEEE International Conference on Electronics, Circuits and Systems, 2018, pp. 773–776.

[34] Herkersdorf, A.; Rosenstiel, W. “Towards a Framework and a Design Methodology for Autonomic Integrated Systems”. In: Gesellschaft für Informatik Workshop on Organic Computing, 2004, pp. 610–615.

[35] Hopcroft, J. E.; Motwani, R.; Ullman, J. D. “Introduction to Automata Theory, Languages, and Computation”. Addison-Wesley Longman Publishing Co. Inc., 2006, 3rd ed., 550p.

[36] IBM. “An architectural blueprint for autonomic computing”. Source: https://www-03.ibm.com/autonomic/pdfs/AC%20Blueprint%20White%20Paper%20V7.pdf, February 2020.

[37] Imperas Software. “Welcome Page | Open Virtual Platforms”. Source: http://www.ovpworld.org, November 2019.

[38] International Organization for Standardization (ISO). “International Standard ISO/IEC 7498-1”. Source: https://standards.iso.org/ittf/PubliclyAvailableStandards/s020269_ISO_IEC_7498-1_1994(E).zip, February 2020.

[39] Isuwa, S.; Dey, S.; Singh, A. K.; McDonald-Maier, K. “Teem: Online thermal- and energy-efficiency management on cpu-gpu mpsocs”. In: Design, Automation Test in Europe Conference Exhibition, 2019, pp. 438–443.

[40] Johann, S. F. “sjohann81/hellfreos: HellfireOS Realtime Operating System”. Source: https://github.com/sjohann81/hellfireos, December 2018.

[41] Johann, S. F. “sjohann81/hf-risc: HF-RISC SoC”. Source: https://github.com/sjohann81/hf-risc, December 2018.

[42] Kofman, E.; Muzu, A.; Zeigler, B. “Theory of Modeling and Simulation: Discrete Event and Iterative System Computational Foundations”. Elsevier Academic Press, 2019, 3rd ed., 694p.

[43] Kramer, D.; Buchty, R.; Karl, W. “Organic Computing — A Paradigm Shift for Complex Systems”. Springer Basel, 2011, 1st ed., 627p.

[44] Kreuzinger, J.; Schulz, A.; Pfeffer, M.; Ungerer, T.; Brinkschulte, U.; Krakowski, C. “Real-Time Scheduling on Multithreaded Processors”. In: International Conference on Real-Time Computing Systems and Applications, 2000, pp. 155–159.

[45] Krupitzer, C.; Roth, F. M.; VanSyckel, S.; Schiele, G.; Becker, C. “A Survey on Engineering Approaches for Self-Adaptive Systems”, *Pervasive and Mobile Computing*, vol. 17, Feb 2015, pp. 184 – 206.

[46] Kuhn, T. S. “The Structure of Scientific Revolutions”. University of Chicago Press, 1970, 1st ed., 210p.

[47] Lipsa, G.; Herkersdorf, A.; Rosenstiel, W.; Bringmann, O.; Stechele, W. “Towards a Framework and a Design Methodology for Autonomic SoC”. In: International Conference on Autonomic Computing, 2005, pp. 391–392.

[48] Martins, A. L. D. M. “Multi-Objective Resource Management for Many-Core Systems”, Ph.D. Thesis, Pontifícia Universidade Católica do Rio Grande do Sul, Porto Alegre, Brazil, 2018, 147p.

[49] Martins, A. L. d. M.; da Silva, A. H. L.; Rahmani, A. M.; Dutt, N.; Moraes, F. G. “Hierarchical Adaptive Multi-Objective Resource Management for Many-Core Systems”, *Journal of Systems Architecture*, vol. 97, Aug 2019, pp. 416 – 427.

[50] Mentor Graphics. “Questa advanced simulator - mentor graphics”. Source: https://www.mentor.com/products/fv/questa/, February 2020.

[51] Mentor Graphics. “Handel-C Synthesis Methodology – Mentor Graphics”. Source: https://www.mentor.com/products/fpga/handel-c/, December 2019.

[52] Microsoft. “Commands by Server Role”. Source: https://docs.microsoft.com/en-us/windows-server/administration/windows-commands/commands-by-server-role, February 2020.

[53] Miele, A.; Kanduri, A.; Moazzemi, K.; Juhász, D.; Rahmani, A. R.; Dutt, N.; Liljeberg, P.; Jantsch, A. “On-Chip Dynamic Resource Management”, *Foundations and Trends® in Electronic Design Automation*, vol. 13–1-2, Jul 2019, pp. 1–144.

[54] Mishra, P.; Nidhi., A.; Kishore, J. K. “Custom Network on Chip architecture for Map Generation in Autonomous Navigating Robots”. In: Annual IEEE India Conference, 2012, pp. 086–091.

[55] Moore, G. E. “Cramming More Components Onto Integrated Circuits”, *IEEE Solid-State Circuits Society Newsletter*, vol. 11–3, Sep 2006, pp. 33–35.

[56] Moraes, F.; Calazans, N.; Mello, A.; Möller, L.; Ost, L. “HERMES: An Infrastructure for Low Area Overhead Packet-Switching Networks on Chip”, *Integration*, vol. 38–1, Oct 2004, pp. 69 – 93.

[57] MyHDL Community. “MyHDL”. Source: http://www.myhdl.org/, February 2020.

[58] NVIDIA. “The Benefits of Multi-core CPUs in Mobile Devices”. Source: https://www.nvidia.com/content/PDF/tegra_white_papers/Benefits-of-Multi-core-CPUs-in-Mobile-Devices_Ver1.2.pdf, February 2020.

[59] Open Robotics. “ROS.org | Powering the world’s robots”. Source: https://www.ros.org, February 2020.

[60] Open Source Robotics Foundation (OSRF). “Gazebo”. Source: http://gazebosim.org, February 2020.

[61] Pasricha, S. “Tutorial T2F: Silicon Nanophotonics for Future Manycore Chips: Opportunities and Challenges”. In: International Conference on Very Large Scale Integration Design and International Conference on Embedded Systems, 2018, pp. xlv–xlvi.

[62] Pena, M. D. V.; Rodriguez-Andina, J. J.; Manic, M. “The Internet of Things: The Role of Reconfigurable Platforms”, *IEEE Industrial Electronics Magazine*, vol. 11–3, Sep 2017, pp. 6–19.

[63] Porrmann, M.; Purnaprajna, M.; Puttmann, C. “Self-optimization of MPSoCs Targeting Resource Efficiency and Fault Tolerance”. In: NASA/ESA Conference on Adaptive Hardware and Systems, 2009, pp. 467–473.

[64] Rodrigues, E. d. M. “PLeTs : A Product Line of Model-Based Testing Tools”, Ph.D. Thesis, Pontifícia Universidade Católica do Rio Grande do Sul, Porto Alegre, Brazil, 2013, 130p.

[65] Roelke, A.; Stan, M. R. “RISCV5: Implementing the RISC-V ISA in gem5”. In: Workshop on Computer Architecture Research with RISC-V, 2017, pp. 1–7.

[66] Ruaro, M.; Caimi, L.; Fochi, V.; Moraes, F. “Memphis: A Framework for Heterogeneous Many-Core SoCs Generation and Validation”, *Design Automation for Embedded Systems*, vol. 23, Mar 2019, pp. 103–122.

[67] Ruaro, M.; Lazzarotto, F. B.; Marcon, C. A.; Moraes, F. G. “DMNI: A Specialized Network Interface for NoC-based MPSoCs”. In: IEEE International Symposium on Circuits and Systems, 2016, pp. 1202–1205.

[68] Rupanetti, D.; Salamy, H. “Energy Efficient Scheduling with Task Migration on MPSoC Architectures”. In: IEEE International Conference on Electro Information Technology, 2019, pp. 156–161.

[69] Rupp, K. “42 Years of Microprocessor Trend Data”. Source: https://www.karlrupp.net/2018/02/42-years-of-microprocessor-trend-data, November 2019.

[70] Rupp, K. “CPU Benchmark - MCU Benchmark - CoreMark - EEMBC Embedded Microprocessor Benchmark Consortium”. Source: https://www.eembc.org/coremark/, November 2019.

[71] Sametriya, D. P.; Vasavada, N. M. “HC-CPSoC: Hybrid Cluster NoC Topology for CPSoC”. In: International Conference on Wireless Communications, Signal Processing and Networking, 2016, pp. 240–243.

[72] Sarma, S.; Dutt, N.; Gupta, P.; Nicolau, A.; Venkatasubramanian, N. “On-chip Self-awareness Using Cyberphysical-systems-on-chip (CPSoC)”. In: International Conference on Hardware/Software Codesign and System Synthesis, 2014, pp. 1–3.

[73] Sarma, S.; Dutt, N.; Gupta, P.; Venkatasubramanian, N.; Nicolau, A. “CyberPhysical-System-On-Chip (CPSoC): A Self-Aware MPSoC Paradigm with Cross-Layer Virtual Sensing and Actuation”. In: Design, Automation & Test in Europe Conference & Exhibition, 2015, pp. 625–628.

[74] Sarma, S.; Dutt, N.; Venkatasubramanian, N.; Nicolau, A.; Gupta, P. “Cyberphysical-System-On-Chip (CPSoC): A Self-Aware Design Paradigm with Cross-Layer Virtual Sensors and Actuators”, Technical Report, University of California, Irvine, 2013, 26p.

[75] Schuck, C.; Lamparth, S.; Becker, J. “artNoC - A Novel Multi-Functional Router Architecture for Organic Computing”. In: International Conference on Field Programmable Logic and Applications, 2007, pp. 371–376.

[76] Siddiqui, F.; Hagan, M.; Sezer, S. “Pro-Active Policing and Policy Enforcement Architecture for Securing MPSoCs”. In: IEEE International System-on-Chip Conference, 2018, pp. 140–145.

[77] Smart, K. “The Life Cycle of a Virtual Platform”. Springer US, 2010, chap. 2, pp. 7–24.

[78] Utting, M.; Legeard, B. “Practical Model-Based Testing: A Tools Approach”. Morgan Kaufmann, 2006, 4th ed., 456p.

[79] Vancin, P. H.; Domingues, A. R. P.; Paravisi, M.; Johann, S. F. “Towards an Integrated Software Development Environment for Robotic Applications in MPSoCs with Support for Energy Estimations”. In: IEEE International Symposium on Circuits and Systems, 2019, pp. 6, *to appear in*.

[80] Wachter, E. W.; Fochi, V.; Barreto, F.; Amory, A. M.; Moraes, F. G. “A Hierarchical and Distributed Fault Tolerant Proposal for NoC-Based MPSoCs”, *IEEE Transactions on Emerging Topics in Computing*, vol. 6–4, Oct 2018, pp. 524–537.

[81] Waterman, A.; Asanović, K. “The RISC-V Instruction Set Manual, Volume I: User-Level ISA, Document Version 2.2”. Source: https://content.riscv.org/wp-content/uploads/2017/05/riscv-spec-v2.2.pdf, February 2020.

[82] Wireshark Foundation. “Wireshark – Go Deep.” Source: https://www.wireshark.org, July 2020.

[83] Wolf, W.; Jerraya, A. A.; Martin, G. “Multiprocessor System-on-Chip (MPSoC) Technology”, *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 27–10, Oct 2008, pp. 1701–1713.

[84] Xilinx Inc. “ISE Design Suite”. Source: https://www.xilinx.com/products/design-tools/ise-design-suite.html, February 2020.

[85] Zeppenfeld, J.; Bouajila, A.; Herkersdorf, A.; Stechele, W. “Towards Scalability and Reliability of Autonomic Systems on Chip”. In: IEEE International Symposium on Object/Component/Service-Oriented Real-Time Distributed Computing Workshops, 2010, pp. 73–80.

[86] Zeppenfeld, J.; Bouajila, A.; Stechele, W.; Herkersdorf, A. “Learning Classifier Tables for Autonomic Systems on Chip”. In: Jahrestagung der Gesellschaft für Informatik, 2008, pp. 771–778.

[87] Zeppenfeld, J.; Herkersdorf, A. “Applying Autonomic Principles for Workload Management in Multi-core Systems on Chip”. In: ACM International Conference on Autonomic Computing, 2011, pp. 3–10.

[88] Zhou, J.; Sun, J.; Zhou, X.; Wei, T.; Chen, M.; Hu, S.; Hu, X. S. “Resource Management for Improving Soft-Error and Lifetime Reliability of Real-Time MPSoCs”, *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 38–12, Dec 2019, pp. 2215–2228.

APPENDIX A – TUTORIAL FOR USING THE ORCA PLATFORM

This appendix is an introductory tutorial to introduce the basics of the operation of the ORCA platform. By the end of this tutorial, the reader should be able to download the source code for the required components from the proper repositories, compile and set up the components, and run a full system simulation using the ORCA platform.

The platform comprises four source bases; (i) one for the ORCA-SIM simulation tool, (ii) one for the software assets, (iii) one hosting a synthesizable RTL project of the MPSoC, and (iv) one for integration tools. In this tutorial, we cover the first two source bases.

A.1 Requirements

For this tutorial, we need two different compilers; one for targeting the host machine's architecture and another one targeting the MPSoC architecture, which implements the RV-32i instruction set [81]. For the host machine, we advise installing GNU Compiler Collection (GCC), preferably the latest one available. If compiling code for the MPSoC, a cross-compiler is also necessary. One can either acquire a pre-compiled package from hardware vendors (e.g., Si-Five) or build your cross-compiler from the source. We present two alternatives for acquiring a compatible compiler below. In addition to the compilers, a git client and Make are also required tools, both available from most package managers in Unix-like systems.

We developed the platform using Debian 9 and tested for Debian 8, 9, and 10, Ubuntu 16.04 and 18.04 systems. Although we did not perform any tests on other platforms, the platform should work fine in any system running GCC with C++17 support.

- **Si-Five's Toolchain**
  - https://www.sifive.com/boards
- **Johann's Build Script**
  - https://github.com/sjohann81/hellfireos/tree/master/usr/tools/riscv-toolchain

A.2 Acquiring the source code

The next step is to gather the source code for the platform components from the respective source base. One must download the source code for both the ORCA-SIM simu-

lator and the software assets — using git to do so is optional, although advised. We provide the URL for both source bases below.

- **ORCA-SIM**
  - https://github.com/andersondomingues/orca-sim
- **ORCA-SOFTWARE-TOOLS**
  - https://github.com/andersondomingues/orca-software-tools

A.3 Compiling the simulator and software image

After downloading the source code from both source bases, we compile the respective code using the previously installed compilers. First, from a system's terminal, we navigate to the ORCA-SIM project folder, which should be called `orca-sim`. To compile the project, enter `make` to the terminal and wait for the project to compile. As a result, we can observe that an executable binary named after the platforms' name is deployed to the `orca-sim/bin` folder. One can change the name of the executable, paths, and other configurations by modifying the `orca-sim/Configuration.mk` file. Instructions regarding the configuration parameters are provided in the same file. Finally, the generated program, the binary, can be invoked, passing a software image file as parameters. Before we invoke the simulator, we must generate the software image.

To generate the software image, we navigate to the `orca-software-tools` folder and type `make` to the terminal. Similarly to the ORCA-SIM build process, the results of the compilation are deployed to the `orca-software-tools/bin` folder. Note that there are many files in the folder, as these are partial results of the many steps of the compilation process. The software image is called `image.bin`. Please note that the compilation of the software image requires a cross-compiler targeting the RV32i instruction set. The provided makefile expect compilation tools to have the prefix `riscvXX-unknown-elf` (e.g., `riscvXX-unknown-elf-gcc`, `riscvXX-unknown-ld`), where XX stands for either 32 or 64. Please note that the `riscv64-unknown-gcc` can also emit 32-bit instructions. Make sure these tools are added to the environmental path (usually through the `$PATH` system variable) before beginning the compilation process.

The software image contains the executable code and data for applications, libraries, and an operating system. By compiling the software image as it is, applications and libraries for the examples provided in this thesis will be included in the compilation, all they run on top of the HellfireOS kernel. It is worth to note that all the software is optional, as there is no dependency between ORCA-SIM and the software image. For instance, the compiled software image can be replaced by another image containing standalone bare-

metal applications. Another possible software image may include the same applications and libraries like the one that we provide but using another operating system.

A.4 Running the simulation

After having compiled both the simulator (`orca-sim/bin/orca-dma.exe`) and the software image (`orca-software-tools/bin/image.bin`), we invoke the simulator passing the software image as parameter. Assuming that the source code of both components are deployed to the same folder in the file system, we navigate to `orca-sim/bin` and enter `./orca-dma.exe ../orca-software-tools/bin/image.bin` to the terminal.

The simulator will promptly launch, and some information will be put on the terminal, including simulation speed (considering host machine execution time), the architecture configuration (number of processor cores, memory size), and current simulation time. The simulation can be interrupted any time by typing `CTRL+C` to the terminal. One can change the `Configuration.mk` to simulate for a fixed number of cycles if necessary. At the end of the simulation, the state of the simulation hardware will be reported. During the simulation, the output (UART) of each processor core will be written to log files, deployed to the `orca-sim/bin/logs` folder. These files can be inspected using tools such as `multitail`. For that purpose, one can execute the script `orca-sim/bin/output-uart.sh` if `multitail` is installed. The simulation results should be as shown in Figure A.1.

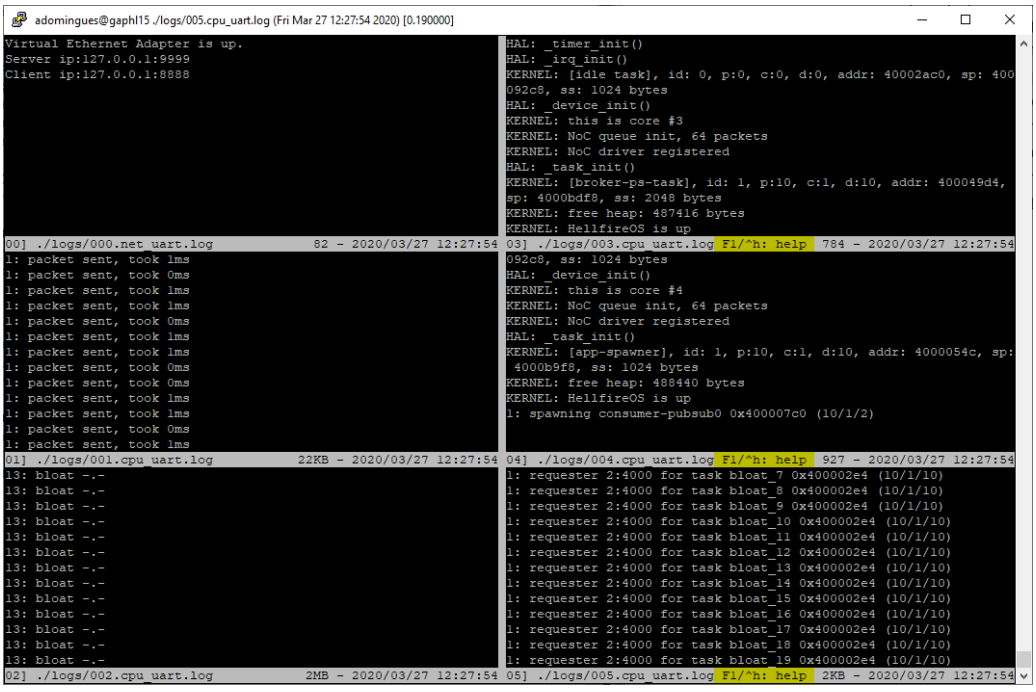

Figure A.1 – An example of simulation visualization using the multitail tool. Log files are displayed in a grid layout resembling the MPSoC configuration.

APPENDIX B – TUTORIAL FOR CREATING AN UNTIMED MULTIPLIER

In this tutorial, we create an untimed multiplier and attach it to the ORCA platform's processor cores. A multiplier is a small piece of hardware that can perform multiplications. For simplicity, we consider the multiplier to be untimed; that is, any clock domains do not influence the modules' operation. Hence, multiplication operations occur instantly, and the result at the output will always match the product of given inputs. Of course, restrictions apply to the size of the operands — the result will be correct if it fits in the 32-bit wide output register. The interface of our multiplier is as shown in Figure B.1. We will write the multiplier model from scratch, integrate it into the processor core model, and write a small software application to test our module.

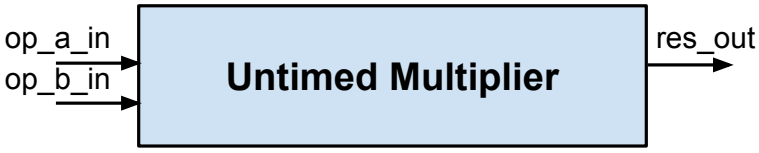

Figure B.1 – Interface of an untimed multiplier module. Operands A and B are presented by `op_a_in` and `op_b_in` inputs. The result of the multiplication of both operands, the product, is always available at the `res_out` output.

B.1 Coding the Hardware Module

In URSA, hardware modules are represented by classes that can inherit from UntimedModel or TimedModel classes. In the case of our multiplier, we extend the UntimedModel class. First, we create the header and implementation sources for our multiplier, defining five operations, where the first two correspond to the constructor and destructor of the class, and the other three correspond to methods for accessing the input and output of our module. An excerpt from the header file for our multiplier is shown in Figure B.2.

```
01 #include <UntimedModel.h>
02
03 class UMultiplier : public UntimedModel{
04
05 private:
06    uint32_t _opa;
07    uint32_t _opb;
08
09
10 public:
11    UMultiplier::UMultiplier();
12    UMultiplier::~UMultiplier();
13
14    void UMultiplier::SetOpA(uint32_t);
15    void UMultiplier::SetOpB(uint32_t);
16
17    uint32_t UMultiplier::GetResult();
18 };
```

Figure B.2 – Excerpt of the header file for the multiplier model.

Our multiplier must have at least two registers of 32 bits to store both operands. Please note that we use `uint32_t` as the type due to our module is capable of performing

only integer multiplication. For floating-point multiplication, `float` must be used instead. The implementation source code for our module is shown in Figure B.3.

```
01 #include <UMultiplier.h>
02
03 UMultiplier::UMultiplier(std::string n)
04    : UntimedModel(n){
05    //nothing to do
06 }
07
08 UMultiplier::~UMultiplier(){
09    //nothing to do
10 }
11 void UMultiplier::SetOpA(uint32_t a){
12    _opa = a;
13 }
14
15 void UMultiplier::SetOpB(uint32_t b){
16    _opb = b;
17 }
18
19 uint32_t UMultiplier::GetResult(){
20    return _opa * _opb;
21 }
```

Figure B.3 – Implementation file for the multiplier model.

In this tutorial, we use the ORCA-SIM project as the basis for our implementation, so we do not need to set up the environment from scratch. We deploy header and implementation files of our multiplier to the `models` folder, located in the root folder of ORCA-SIM, editing `models/Makefile` to include the new hardware in the compilation. Recompile the project, entering `make` to the terminal in the root folder of ORCA-SIM.

B.2 Changing between platforms

Before connecting the multiplier to the processor cores, let us change the target platform. By default, ORCA-SIM is configured to behave as a many-architecture, comprising a mesh of several processor cores. However, a single-core platform is also included in ORCA-SIM. We must set the compilation to use this platform instead of the default many-core by changing the `Configuration.mk` file in the root directory of ORCA-SIM. Change the variable `PLATFORM` to `single-core`.

B.3 Connecting the multiplier to the processor core

There are a couple of ways of integrating our module in the platform‘s hardware. For this tutorial, we include the multiplier into the processor core as a peripheral whose inputs and outputs are mapped to the memory space. To do so, locate the source of the processor core module and modify its header `models/include/THellfireProcessor.h` and source `models/src/THellfireProcessor.cpp` accordingly. For the header, add include for the multiplier header and a field to hold a pointer for a multiplier instance. For the source, instantiate the multiplier in the processor core constructor method. The lines to be modified are shown in Figure B.4.

(models/include/THellfireProcessor.h)

```
10 #include <UMultiplier.h>
87 ...
88 private:
89
90 ...
91    UMultiplier* _mu;
92 ...
```

(models/src/THellfireProcessor.cpp)

```
 13 #include <UMultiplier.h>
 14 ...
687
688 THellfireProcessor::THellfireProcessor ...
689
690    _mu = new UMultiplier();
691 ...
```

Figure B.4 – A pointer to a multiplier (left) and a new instance of multiplier (right).

B.4 Interacting with the multiplier via memory-mapped I/O

A new instance of the multiplier has been added to the architecture. However, the processor core cannot interact with the multiplier yet. We must map the inputs and outputs of the multiplier to the memory space before the peripheral can be accessed via software. First, we must define an address for each of the operators and a third address to hold the multiplication result. We define the addresses show in Figure B.5, adding these definitions to the top of the `UntimedMultiplier.h` file. For more information on the address availability, see `orca-sim/platforms/single-core/include/ProcessingTile.h`.

```
01 #define MULT_OPA 0x4000F000
02 #define MULT_OPB 0x4000F004
03 #define MULT_RES 0x4000F008
```

Figure B.5 – Inputs and outputs of the multiplier mapped into the memory space.

The next step is to modify the processor core code access the peripheral when reading or writing to these specific addresses, modifying the `THellfireProcessor::mem_read` and `THellfireProcessor::mem_write` methods. An excerpt of the modified methods is shown in Figure B.6 and Figure B.7.

```
 96 int32_t THellfireProcessor::mem_read ...
 97 ...
151 switch(address){
152    case IRQ_VECTOR: return s->vector;
153 ...
160    case UART_READ:  return getchar();
161    case UART_DIVISOR: return 0;
162    case MULT_RES : return _mu->GetResult();
163 }
```

Figure B.6 – Modified mem_read method.

```
187 int32_t THellfireProcessor::mem_write ...
188 ...
247 switch(address){
248    case IRQ_STATUS:{
249 ...
267    case DEBUG_ADDR: output_debug ...
268    case UART_WRITE: output_debug ...
269    case UART_DIVISOR: return;
270    case MULT_OPA: _mu->SetOpA(value); return;
271    case MULT_OPB: _mu->SetOpB(value); return;
272 ...
```

Figure B.7 – Modified mem_write method.

B.5 Accessing the multiplier through software

The last step is to write a software piece to access the multiplier and perform some multiplications. As we use the memory-mapped I/O strategy to access our peripheral, there is no need to extend the ISA to include new instructions or modify the processor core logic. Instead, the processor core can access peripheral's inputs and output by reading and writing to the memory space. We defined three memory addresses for the multiplier: `MULT_OPA`, `MULT_OPB`, and `MULT_RES`. Every time the processor core reads from `MULT_RES`, the multiplication between `MULT_OPA` and `MULT_OPB` will be available in that address. Please note that, in practice, nothing prevents the CPU from reading from `MULT_OPA` and `MULT_OPB`, neither from writing to `MULT_RES`. For that reason, we write a small driver software to expose the proper API to the application and kernel software levels.

A software driver that works for our multiplier is show in Figure B.8. In the driver, we prevent the software from executing illegal writings and readings to the peripheral. Besides, we provide the routine `int ext_imul(int a, int b)`, ready to be used by application developers. It worth to note that the driver must compiled using the cross-compiler targeting the RV32i ISA.

(mul_driver.h)

```
01 #define MULT_OPA 0x4000F000
02 #define MULT_OPB 0x4000F004
03 #define MULT_RES 0x4000F008
04
05 volatile int* __mopa_ptr
06    = (volatile int*)(MULT_OPA);
07 volatile int* __mopb_ptr
08    = (volatile int*)(MULT_OPB);
09 volatile int* __mres_ptr
10    = (volatile int*)(MULT_RES);
11
12 int ext_imul(int a, int b);
```

(mul_driver.c)

```
01 #include <mul_driver.h>
02
03 int ext_imul(int a, int b){
04   *__mopa_ptr = a;
05   *__mopb_ptr = b;
06   return *__mres;
07 }
```

Figure B.8 – A simple peripheral driver for the untimed multiplier module.

APPENDIX C – TUTORIAL FOR CREATING TIMED MODELS

Timed models correspond to modules that are sensitive to one or more clock domains. In URSA, these models have their clock cycles mapped to events, whose scheduling depends on the frequency of the simulated hardware. In this tutorial, we create a timed integer divisor capable of performing integer division, with the result being available at the output after a non-fixed number of cycles. As in the last tutorial, we write a software driver to interact with the divisor. The interface of our divisor is shown in Figure C.1.

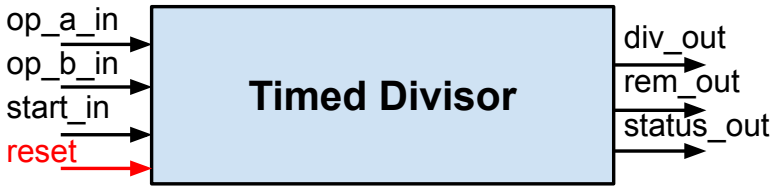

Figure C.1 – Inputs and outputs for the divisor module. Inputs highlighted in red orange are optional, and inputs in red are abstracted by the simulation engine.

The operands of division operations will feed two inputs, `op_a_in` and `op_b_in`, where the results of the division between `op_a_in` and `op_b_in` will be available at `div_out`, and the remainder of the operation will be available at `rem_out`. Since the timed divisor takes more than one cycle to produce the results, we add the `status_out` output to the model. The divisor have three states: `idle` (0), `processing` (1), and `done` (2). To start the divisor, the processor core must write `1` to the `start_in` input, and write `0` later to acknowledge the operation. Finally, our module have a `reset` input, which is connected to the global reset of the system.

C.1 Modeling timed models as state machines

To simplify the implementation of our module, let us assume the algorithm for integer division by sucessive subtraction[1]. In other words, our module will subtract the `op_b_in` from `op_a_in` until no more subtractions can be performed anymore. After the last subtraction, the number of subtractions will be pushed to `div_out`, and the remainder of the division will be pushed to `rem_out`. Our module is capable of performing one subtraction per cycle, taking always (`op_a_in` $\div$ `op_b_in` + k) cycles to perform the whole division algorithm, where k is the number of cycles spend with the handshaking protocol. We translate the behaviour of the divisor into the transition system shown in Figure C.2.

[1] For a RTL implementation of the sucessive division algorithm, see https://ieeexplore.ieee.org/stamp/stamp.jsp?tp=&arnumber=5158757.

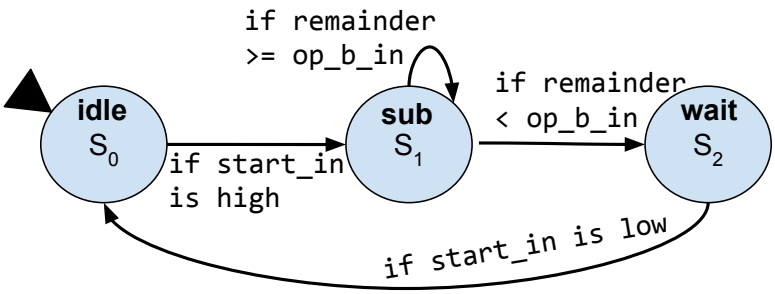

Figure C.2 – Transition system representing the divisor.

Although a transition system provided useful information about the behavior of the system, it does not captures the relation between the inputs and outputs of the system. This relation can be captured, however, by more powerful models such as finite state machines. Below we provide the formal modeling of the divisor. It is worth to note that δ is a partial function whose undefined elements are left to the implementation to decide.

$$DIV = (Q, Q_0, X, Y, \delta, F) \tag{C.1}$$

$$Q = \{idle, sub, wait\} \tag{C.2}$$

$$Q_0 = idle \tag{C.3}$$

$$X = \{op\_a\_in \in 2^{32}, op\_b\_in \in 2^{32}, start\_in \in 2^1\} \tag{C.4}$$

$$Y = \{div\_out \in 2^{32}, rem\_out \in 2^{32}, status\_out \in 2^2\} \tag{C.5}$$

$$\delta : (X \times Q) \rightarrow (Y \times Q) = \{(x \in X, idle) \mapsto ((0, 0, 1), sub), ...\} \tag{C.6}$$

C.2 Translating the model into code

To be accepted as a valid model for simulation, any model must be a class and must inherit from the TimedModel class `orca-sim/simulation/include/TimedModel.h`. Any timed model must explicitly implement the method `long long int Run()`, which must contain the implementation of the module, usually based on the underlying transition system. Since URSA uses a discrete-event simulation system, it permits modules to skip idle cycles to gain simulation performance. One of the advantages of this strategy is that the models can be designed to match the behaviour of RTL models. One drawback is that the simulation accuracy is conditioned to the model design.

Let us create two files for our module, one for the header, and one for the implementation. The header file, `orca-sim/models/include/TDivisor.h`, show in Figure C.3 must

```
01 #include <TimedModel.h>
02
03 class TDivisor : public TimedModel{
04
05 private:
06    USignal* _sig_opa;
07    USignal* _sig_opb;
08    USignal* _sig_start;
09    USignal* _sig_div;
10    USignal* _sig_mod;
11    USignal* _sig_status;
```

```
13 public:
11    TDivisor::TDivisor(std::string);
12    TDivisor::~TDivisor();
13    long long int TDivisor::Run();
14    USignal* TDivisor::GetSigOpa();
15    USignal* TDivisor::GetSigOpb();
16    USignal* TDivisor::GetSigStart();
17    void TDivisor::SetSigDiv(USignal*);
18    void TDivisor::SetSigMod(USignal*);
19    void TDivisor::SetSigStatus(USignal*);
20 }
```

Figure C.3 – Header file for the divisor module.

contain the definition of our class, which we call `TDivisor`. In addition to the required Run method, we define a couple of instances of the class USignal in our module. The USignal class `orca-sim/models/include/USignal.h` is used for the communication of two or more timed models. Other communications methods are available in URSA: `UBuffer.h`, for FIFO channels, and `UMemory.h`, for communication via shared memory.

For the implementation file, `orca-sim/models/src/TDivisor.cpp`, we implement the getters and setters of the class, implement the constructor and destructor, and the `Run` method. We show the implementation file in Figure C.4 without the `run` method.

```
01 #include <TDivisor.h>
02
03 TDivisor::TDivisor(std::string n)
04    : TimedModel(n){
05    _sig_opa = new USignal(0,0,0);
06    _sig_opb = new USignal(0,0,0);
07    _sig_start = new USignal(0,0,0);
08 }
09
10 TDivisor::~TDivisor(){
11    delete _sig_opa;
12    delete _sig_opb;
13    delete _sig_start;
14 }
15
16 long long int TDivisor::Run(){
17    ...
18 }
```

```
19
20 USignal* TDivisor::GetOpA(){
21    return _sig_opa ; }
22
23 USignal* TDivisor::GetOpB(){
24    return _sig_opb ; }
25
26 USignal* TDivisor::GetStart(){
27    return _sig_start ; }
28
29 void TDivisor::SetDiv(USignal* s){
30    _sig_div = s; }
31
32 void TDivisor::SetRem(USignal* s){
33    _sig_rem = s; }
34
35 void TDivisor::SetStatus(USignal* s){
36    _sig_status = s;}
```

Figure C.4 – Implementation file for the divisor module.

There a couple of alternatives for implementing the `Run` method. First, we can design a state machine to fake idle cycles while delivering the correct result for the division at the last cycle. This approach will require scheduling the division in each cycle and approximates its RTL design. The second approach is to truly skip idle cycles by scheduling the division once per operation. For both approaches, the result of the division operation will

be delivered at the last cycle of operation. Figure C.5 illustrates the main difference between both approaches.

models/src/TDivisor.cpp (approach A)

```
87 ...
88 long long int Run(){
89
90   switch(_state){
91      case IDLE: /*...*/ break;
92      case SUB: /*...*/
93        idle_counter--; break;
93      case WAIT: /*...*/ break;
94   }
95   return 1;
96 ...
```

models/src/TDivisor.cpp (approach B)

```
87 ...
88 long long int Run(){
89
90   switch(_state){
91      case IDLE: /*...*/ break;
92      case SUB: /*...*/
93        return op_a % op_b;
93      case WAIT: /*...*/ break;
94   }
95   return 1;
96 ...
```

Figure C.5 – The Run method emulating idle cycles (left) and skipping idle cycles (right).

In approach A, we tell the simulation engine to schedule the divisor to execute in every cycle, similarly to RTL models' execution. The method Run must return the number of cycles to wait until the next scheduling. In this case, one represents the next cycle. Since the divisor executed at every cycle, the state transition from state SUB to state WAIT occurs when the remainder of the division is less than `op_b_in` . Although this approach is slower than approach B, it may be useful in case when the RTL design must be compared to the functional models.

In approach B, we tell the simulation engine to schedule the once for each state. When in state SUB, the functions Run will return the number of subtractions that occur before the module determine the final result. By doing this, we skip these subtractions and push the final result to the output in exactly one cycle. However, to keep track of model timing, we need to wait for that many cycles before executing the next state transition. By implementing this approach, we use of underlying host's hardware to perform the division operation and skip the remaining cycles, which is faster than executing every single cycle, as done in approach A.

C.3 Creating a testbench

Testbenches written in software would work pretty much the same as testbenches written in RTL. The goal is to validate the design of a module by pushing values to the input and comparing the module's outputs with the expected values. Let us create a module named DivisorTB. This module will be our testbench for the divisor module. We present a stub of that testbench in Figure C.6.

```
01 #include <TimedModel.h>
02 class DivisorTB: public TimedModel{
03
04 private:
05   USignal* _sig_opa; //input
06   USignal* _sig_opb;
07   USignal* _sig_start;
08   USignal* _sig_div; //output
09   USignal* _sig_rem;
10   USignal* _sig_status;
11 public:
12   DivisorTB::DivisorTB(std::string);
13   DivisorTB::~DivisorTB();
14
15   long long int DivisorTB::Run();
16
17   /* setters here */
18   /* getters here */
19
20 };
```

Figure C.6 – Testbench header file.

For simplicity, let our testbench perform a single division operation and compare the achieved result with what is expected. For instance, we should push the values {opA = 10, opB = 3, start = 1} to the input signals during the first cycle. Then, after 3 cycles we should get the value {div = 3, mod = 1, status = 1} from the divisor. We know that the number of cycles for the divisor to perform the division is 3 division it used successive subtractions to find the answer. We provide the expected values for the signals in Table C.1.

Table C.1 – Implementation of requirements per platform.

Cycle	**Signals**					
	op_a_in	op_b_in	start_in	div_out	rem_out	status_out
1	10	3	1	0	0	0
2	10	3	1	0	0	0
3	10	3	1	0	0	0
4	10	3	1	0	0	0
5	10	3	1	3	1	1
6	10	3	0	3	1	1
7	10	3	0	3	1	0

As an exercise, we can design our testbench using state machines. Different from most RTL languages, URSA cannot add delays to signals or schedule signals to change at a specific point in time. Instead, we must define a state machine that could control the hardware module's signals, in this case, the testbench. Our testbench must implement a state machine with at least three states: one state that injects the inputs into the divisor, one state that waits for the divisor to complete the operation, and another state that compares the outputs to the expected values.

C.4 Creating a simulation tool

The last step in the process is to create a simulation tool. This tool will instantiate our design, connect the hardware modules, and run the simulation. Let us create a file named `platforms/MySim/MySim.cpp`, where `MySim` is the name of the tool to be created. In this file, we include the headers for the module that we designed before, the divisor and the testbench. In addition to these files, we must include headers for the Event and Simulator classes. The next step is to create instances of the hardware modules.

After creating the new instances of modules, we must connect the modules by binding their signals. The binding process in RTL files is done almost automatically, as all signals must be present in the modules' interface. In URSA, signals do not require to be in the interface to be bound to external modules. This feature permits more complex signal binding at the cost of code readability.

Lastly, we schedule modules to run in the first simulation cycle. From then on, the simulation engine can process the logic within these modules without any intervention. We show the complete `MySim.cpp` file in Figure C.7.

```
01 #include <Simulator.h>
02 #include <Event.h>
03 #include <Divisor.h>
04 #include <DivisorTB.h>
05
06 int Main(){
07   Divisor* div
08         = new Divisor("div");
09   DivisorTB* tb
10         = new DivisorTB("tb");
11
12   Simulator* sim = new Simulator();
13
14   sim->Schedule(Event(1, div));
15   sim->Schedule(Event(1, tb));
16
17   div->SetOpA(tb->GetOpA());
18   div->SetOpB(tb->GetOpB());
19   div->SetStart(tb->GetStart());
20   tb->SetDiv(div->GetDiv());
21   tb->SetRem(div->GetRem());
22   tb->SetStatus(div->GetRem());
23
24   sim->Run(10);
25 }
```

Figure C.7 – The simulation file, MySim.cpp.

After running the program, nothing will be shown to the terminal. Since we are using C++ to program our simulator, we can output to the terminal the values of any signal at any time. To control the simulation's time, we can call executions of one cycle in a for loop, using `sim->Run(1)`. By doing this, we have access to the modules' values at every cycle; then, we can output them to the screen.

Pontifícia Universidade Católica do Rio Grande do Sul
Pró-Reitoria de Graduação
Av. Ipiranga, 6681 - Prédio 1 - 3º. andar
Porto Alegre - RS - Brasil
Fone: (51) 3320-3500 - Fax: (51) 3339-1564
E-mail: prograd@pucrs.br
Site: www.pucrs.br