Universidad de Alcalá

PhD. Program in Electronics: Advanced Electronic Systems. Intelligent Systems

Co-simulation techniques based on virtual platforms for SoC design and verification in power electronics applications

PhD. Thesis Presented by

Edel Díaz Llerena

2022

Universidad de Alcalá

PhD. Program in Electronics: Advanced Electronic Systems. Intelligent Systems

Co-simulation techniques based on virtual platforms for SoC design and verification in power electronics applications

PhD. Thesis Presented by

Edel Díaz Llerena

Advisors

Raúl Mateos Gil

Emilio José Bueno Peña

Alcalá de Henares, January 11th, 2022

Acknowledgements

Madre, padre, este trabajo, así como todo lo que he conseguido y conseguiré en mi vida es vuestro. Gracias por enseñarme a vivir y a saber cómo conseguir cualquier cosa que me proponga en la vida.

Miriam, gracias por levantarme cada vez que me he caído. Gracias por cuidarme y guiarme. Gracias compartir cada segundo de tu vida conmigo. Te amo.

Hermano, mi eterna referencia. Gracias por enseñarme a aprender y a pensar.

Raúl y Emilio, gracias por creer en mí. Gracias por mostrarme lo que soy capaz de hacer y enseñarme el camino para hacerlo.

Abstract

In the last decades, investment in the energy sector has increased considerably. Nowadays, several companies are developing equipment such as power converters or electrical machines with state-of-the-art control systems. The current trend is to use System-on-chips and Field Programmable Gate Arrays devices to implement the whole control system. These devices enable the use of more complex and efficient control algorithms, improving the efficiency of the equipment and enabling the integration of renewable systems into the power grid. However, the complexity of control systems has also increased considerably and the difficulty of their verification.

Hardware-in-the-loop (HIL) systems offer a solution for non-destructive verification of energy equipment, avoiding accidents and expensive laboratory tests. HIL systems simulate in real-time the behaviour of the power plant and its interface to perform the tests with the control board in a safe environment.

This thesis focuses on improving the verification process of control systems in power electronics applications. The overall contribution is to provide an alternative to using HILs for control board hardware/software verification. The alternative is based on the Software-in-the-loop (SIL) technique and attempts to overcome or address the limitations found in SIL up to date.

To enhance the qualities of SIL, a software tool called COSIL has been developed to co-simulate the final implementation and integration of the control system, be it software (CPU), hardware (FPGA) or a mixture of software and hardware, as well as its interaction with the power plant. This platform can work at multiple levels of abstraction and includes support for mixed-language co-simulation such as C or VHDL.

Throughout the thesis, emphasis is placed on improving one of the limitations of SIL, its low simulation speed. Different solutions are proposed, such as the use of software emulators, different abstraction levels of software and hardware, or local clocks in the FPGA modules. In particular, an external synchronisation mechanism is provided for the QEMU software emulator enabling its multi-core emulation mode. This contribution enables the use of QEMU in virtual co-simulation platforms such as COSIL.

The entire COSIL platform, including the use of QEMU, has been analysed under different types of applications and an actual industrial project. Its use has been critical

to develop and verify the software and hardware of the control system of a 400 kVA converter.

Contact: Edel Díaz Llerena <edel.diaz@uah.es>.

Keywords: HIL, SIL, co-simulation, QEMU, verification.

Contents

List of Figures

List of Tables

List of source code listings

Chapter 1

Introduction

"The main goal of this dissertation is on improving the verification process of control systems in power electronics applications."

In recent decades, the interest in energy management, control, generation and energy efficiency has increased significantly. The commitment of countries and institutions to generate and control clean energy[1] are changing the energy system. Also, the interest of companies to offer more energy-efficient equipment to a highly competitive market has led to a growth in the number of energy-related projects.

In 2008, the European Union (EU) planned the EU's energy pathway and its impact on the climate by 2020 [Eur09]. This route aimed to reduce greenhouse gas emissions by 20% and energy consumption by 20%, increasing equipment efficiency and minimising energy generation and transport losses. A third goal was to increase energy production from renewable sources by 20%. The same year Tesla Motors, founded in 2003, was the fastest-growing car manufacturer in the market [Tes19] by launching its first full-electric car with a range of up to 400 km.

In order to achieve the 2020 goals, from 2012 to 2020, the EU has funded a multitude of public and private projects. Some examples are the Horizon 2020 project [Eur13], focusing on research and development with an initial budget of EUR 1.087 billion, and the NER300 project [Eur12], focusing on renewable energy technologies and carbon capture and storage, with an initial budget of EUR 2 billion. In the case of Horizon 2020, although this project involves broader areas such as economic, social and territorial cohesion, competitiveness for growth and employment and sustainable growth, which together account for 86% of the total budget, a significant percentage of each of these areas have been kept for EU energy development. For example, since 2014, funds from the EU combined with national investment have amounted to EUR 5.3 billion annually [Eur19].

[1] Only products and equipment strictly comply with legally established production standards can be labelled as clean. Regulation (European Union (EU)) 2018/848 of the European Parliament and the Council of 30 May 2018 on organic production and labelling of organic products and repealing Council Regulation (EC) No 834/2007, art. 3.2

Another example of the interest in energy improvement is the goals set by the EU for 2030 [Eur19]. These goals aim to overhaul the automotive and energy market to reduce greenhouse gas emissions by up to 40% and reduce CO2 emissions from passenger cars by up to 37%. In addition, funding for climate conservation will be increased by 5% between 2021-2027, and the aim is to ensure that at least one-third of all energy consumed in the EU comes from renewable energy sources. The goals for 2020 and 2030 proposed by the EU are summarized in the Table below 1.1 [Eur18a, Eur18b].

Table 1.1: EU energy goals for 2020 and 2030.

	Greenhouse gas emissions	Renewable Energy	Energy efficiency	Inter-connection	Climate in EU-funded programmes	CO2 from:
2020	-20%	20%	20%	10%	2014-2020 20%	
2030	≤ -40%	≤ 40%	≤ 32.5%	15%	2021-2027 25%	CARS -37.5% Vams -31% Lorries -30%

As a consequence, many companies are developing equipment with advanced control systems. These control systems make it possible to introduce more complex and efficient control algorithms, improving the efficiency of converters or machines, and enabling the integration of renewable systems into the power system. However, the complexity of control systems has also increased, and therefore, their verification process.

The main goal of this dissertation is focussed on improving the verification process of control systems in power electronics applications.

A control system is made up of one or more *control boards*. The control board processes the sensor measurements with its processing unit and produces changes in the reference of the power plant to meet the desired behaviour or setpoints [Soz17].

The increasing performance and resources of control boards have made it possible to implement complex control systems to manage power plants. In recent years, devices such as Microprocessors (μP), low-cost Digital Signal Processors (DSP), Field Programmable Gate Array (FPGA), System-on-Chip (SoC), or a mixture of the above have been used [BDD19]. The most widely used option for implementing the control platform is coupling a processing unit, like a μP, a DSP or a SoC, with an FPGA. While the processing system simplifies the implementation process of the control algorithms, the FPGA complies with the interfaces, latency, throughput and reconfigurability requirements.

The emergence of Multi-Processor System-On-Chip (MPSoC) with homogeneous and heterogeneous processor architectures has increased the difficulties of Hardware (HW) and software Software (SW) modules and their verification [RAVPM15]. Companies such as Xilinx, Intel, or ARM provide devices that integrate MPSoCs. These devices can combine

the flexibility of the software, such as Operating System (OS) or bare-metal applications, with high-performance hardware designs, such as hardware accelerators and sophisticated Intellectual Property module - Hardare accelerator (IP) modules. However, since it is necessary to simulate the whole behaviour of the system to verify the platform, i.e., software and hardware, and their intercommunication, the verification of these devices is intricated.

The integration process is based on merging the software and hardware modules, and the final version of the source code is developed. The integration process is usually custom-made and is precisely the stage where many source code bugs appear. Verification of the integration, i.e., the full source code, is essential in this type of mixed software-hardware application.

The most extended technique for testing software is debugging. The main drawback of verifying control algorithms in power electronics is that they are implemented in real-time systems. Consequently, it is not possible to stop the full control logic to apply typical debugging techniques. This debugging could stop the execution of the controller, leaving the power plant out of control.

Hardware-in-the-loop (HIL) systems have been presented as a solution for non-destructive verification of power plants, avoiding high cost tests and accidents. HIL systems emulate the behaviour of the power plant and its interface in real-time to perform the tests with the control board in a safe environment. However, the main drawbacks of HILs are their high cost, and the test can only be done at a late development stage. That is when the control platform is available, and it has been programmed. Also, the control board is usually not a standard solution, and its interface is usually not compatible with HIL equipment. This requires the development of an extra platform to communicate the control board with the HIL system. Since the tests are performed on a real-time platform, the debugging capabilities are reduced compared to software verification tools.

Another function of HILs or similar techniques is rapid prototyping. Here, instead of emulating the power plant, the control board is emulated. Thus, it is possible to auto-generate software and hardware modules and test them directly on a real plant. Although this option is very attractive, it does not generate the final source code of the control board, and it only tests specific hardware/software modules.

Different techniques based on modelling and simulation of the power plant and control algorithms have been used as an alternative to HIL. One of these techniques is called Software-in-the-loop (SIL). SIL simulation integrates compiled source code into a simulation environment. Thus, SIL supports closed-loop verification of software or hardware with the power plant. Their use allows verifying designs without having real boards, which dramatically reduces costs and verification time. Nevertheless, the main limitations are the low speed of this co-simulation. Therefore, it can run in real-time. Another limitation is that the verified source code is usually simplified, which is not the same as the final

implementation.

SIL solutions are based on making virtual platforms. Virtual platforms are presented as a perfect solution for evaluating whole systems and analysing hardware/software proposals at the early development stage. These virtual platforms use hardware/software co-simulation techniques profusely described in the literature [Fuj01, KKH19, WFMH20]. In those, the software and hardware designs can be combined at different levels of abstraction and with different timing accuracy.

Typically, three options have been used to speed up co-simulation. Since simulation speed depends on the number of triggered events, an option is to use a higher abstraction level, which requires fewer triggered events. This thesis develops a virtual platform that allows using SystemC language to describe hardware, as in [MPNB12], and Transaction-Level Modeling (TLM) to model communications [IEE11].

Another alternative is to parallelise the resources of the virtual platform, focusing on the hardware or software simulators. In recent years, Moore's Law has stagnated due to the physical limitations of the current technology [DDC+19]. To continue offering higher performance, the current trend consists of increasing the number of Central Processing Unit (CPU)s. Thus, significant efforts to parallelise the software verification tool and adapt them to new challenges have been described in the literature.

The last option is based on emulation or virtualisation. The emulation and virtualisation allow using the host hardware resources to run the whole guest system. Hence, it can get quick emulations or achieve native performance. Therefore, most tools apply techniques to emulate software behaviour rather than simulate it, decreasing debugging features and increasing execution speed.

The synchronisation carried out in a co-simulation virtual platform is essential to obtain correct results. Even though works have advanced in the parallelisation of the software emulator, they have not addressed the synchronisation of the software emulator with other applications (i.e., the hardware simulator).

This thesis presents a verification tool based on a co-simulation virtual platform. The platform has been called Co-simulation Software-In-the-Loop (COSIL) and is based on the Software-in-the-loop methodology. This co-simulation tool mixes the power plant simulations with the final software (CPUs/SoC) and hardware (FPGA) of the control board. Thus, it allows having a global vision of the system in the same environment. This contribution reduces the development times because the code verified by co-simulation can be installed directly on the board without any modification. Therefore, COSIL focuses on verifying the full integration of the hardware/software source code. The power plant model is now simulated with the whole control board implementation in the same workstation, allowing a closed-loop verification. Then, it is presented as a HIL alternative.

In order to increase COSIL performance, a multi-core software emulator has been used. However, currently the synchronisation between multi-core emulators and external

simulators has not been addressed in the literature. This thesis analyses the current software emulators in the market and presents an external synchronisation mechanism for Quick Emulator (QEMU), the most compatible open-source software emulator. The synchronisation mechanism modifies the original QEMU proposal of the Multi-Thread translator (parallelised mode). The modifications enable parallelized-QEMU to emulate multi-core embedded processors in hardware/software co-simulation virtual platforms. The proposed mechanism does not introduce a substantial overhead in the QEMU speed, and this novel solution allows running hardware/software co-simulation quickly.

The COSIL platform includes the QEMU emulator (open-source) and uses the contributions of this thesis to synchronise QEMU, as a software emulator, with the hardware and power plant simulators. COSIL is based on SystemC/C++ (open-source) to describe the co-simulated virtual platform at multiple levels of abstraction. SystemC furthermore controls the advance of the co-simulation with its event-driven simulation kernel. As COSIL is composed of open-source elements, COSIL presents itself as a more cost-effective verification tool than HIL systems, among other advantages.

An industrial designs has been considered to evaluate the performance of COSIL. That project implements the control system for a 400 kVA back-to-back converter based on Zynq-7000 SoC+FPGA (dual-core Cortex A9+Artix-7). The converter acts as a grid emulator to test photovoltaic converters. Therefore, it will have different setups, such as voltage/current source or programmable load. All these functionalities have been verified using the COSIL tool before being implemented on the real converter. COSIL allows verifying any improvement to be included in the converter once it is manufactured and installed. The COSIL performance analysis and tests are added using different detail levels for both software and hardware.

Although the platform has been tested for this application, COSIL is a universal tool that can be used to test any converter and even for other applications that are not in the area of power electronics.

1.1. Contributions

The overall contribution of this thesis is to provide a HIL alternative for the verification of control board designs in power electronics applications. The alternative is based on the Software-in-the-loop technique and attempts to overcome or address the limitations found in Software-in-the-loop for control system verification.

This overall contributions is made-up with a set of individuals contributions. They will be describe in this thesis:

- A software tool, called COSIL, has been developed. COSIL is a co-simulation virtual platform focus on verifying the code integration of software/hardware designs imple-

mented in the control boards. This tool allows verifying the final software/hardware code with the power plant simulation models. The tool allows mixing multiple levels of abstraction, adapting the complexity of the co-simulation to the user's needs.

- An external synchronisation mechanism to use QEMU in the co-simulation virtual platform as a multi-core software emulator. That allows speed up the execution of hardware/software co-simulation in COSIL.
- Multiple recommendations are presented to optimise the speed of co-simulation.

1.2. Structure of the dissertation

The following chapters illustrate the issues and contributions presented in this introduction:

- Chapter 2 - State of the art - The second chapter shows the state-of-the-art of thesis. The chapter explains co-simulation techniques, design and verification of control systems in power electronics, QEMU internals and details about SystemC.
- Chapter 3 - QEMU External Synchronization Mechanism - The third chapter explains how to enable the QEMU external synchronisation with other simulators in the parallel emulation mode.
- Chapter 4 - COSIL: Co-simulation Software-In-the-Loop - That chapter presents the COSIL platform and shows a performance study in an industrial project.
- Chapter 5 - Test and results - The fifth chapter shows the results obtained to demonstrate the feasibility of the contributions.
- Chapter 6 - Conclusions - The sixth chapter exposes the thesis conclusion, discussions and future works.

1.3. Publications and related

International journals

- E. Díaz, R. Mateos, E. Bueno and R. Nieto, "Enabling Parallelized-QEMU for Hardware/Software Co-Simulation Virtual Platforms," Electronics 2021, 10, 759. https://doi.org/10.3390/electronics10060759
- E. Díaz, R. Mateos and E. Bueno, "COSIL: a new co-simulation Software-in-the-Loop tool for verification in Power Electronics," IEEE Transactions on Industrial Informatics, 2021 [in process of revision].

International congress

- E. Díaz, R. Mateos and E. Bueno, "Virtual Platform of FPGA based SoC for Power Electronics Applications," 2019 IEEE 28th International Symposium on Industrial Electronics (ISIE), 2019, pp. 1371-1376, doi: 10.1109/ISIE.2019.8781247.
- E. Díaz, R. Mateos and E. Bueno, "Virtual Platform of FPGA based MPSoC for Power Electronics Applications: OS simulation," IECON 2019 - 45th Annual Conference of the IEEE Industrial Electronics Society, 2019, pp. 3118-3123, doi: 10.1109/IECON.2019.8927331.
- E. Díaz, R. Mateos, J. Pavón and D. Calvo, "Versatile SoC architecture for integration of HW accelerators in power electronics applications," 2021 22nd IEEE International Conference on Industrial Technology (ICIT), 2021, pp. 817-822, doi: 10.1109/ICIT46573.2021.9453471.

Collaborations

- R. Nieto, R. Mateos, Á. Hernández and E. Díaz, "Dual-Core Architecture for PLC Channel Estimator and ASCET Equalizer in a FBMC Transmultiplexer," 2019 24th IEEE International Conference on Emerging Technologies and Factory Automation (ETFA), 2019, pp. 1342-1345, doi: 10.1109/ETFA.2019.8869476.
- R. Nieto, E. Díaz, R. Mateos and Á. Hernández, "Evaluation of Software Inter-Processor Synchronization Methods for the Zynq-UltraScale+ Architecture," 2020 XXXV Conference on Design of Circuits and Integrated Systems (DCIS), 2020, pp. 1-6, doi: 10.1109/DCIS51330.2020.9268616.

Research and others projects

- University of Alcala, Norvento Distributed Energy, S.L. and IMDEA Centre for Energy, "Microgrid-on-chip, battery inverter with integrated converter and microgrid control." RETOS 2017 project, RTC-2017-6262-3.
- University of Alcala and TOTYTRANS, S.L., "Development of a control system for a 1.5MVA Back-to-back-converter." Ref UAH: 106/2018.
- University of Alcala and CERE Certification Entity , "Development of advanced functionalities in a test bench based on a 400KVA back to back converter." Ref UAH: 194/2020.
- University of Alcala and CERE Certification Entity, "Design of a 400kVA power converter operating as grid-forming." Ref UAH: 5/2020.

- University of Alcala and REPSOL S.A, "Provision of an E POWER QUALITY converter capable of compensating for power failures of up to 15 minutes." Ref UAH: 77/2021.

Funding

This work has been supported in part by the Spanish Ministry of Science, Innovation, and Universities through the RETOS 2017 project, RTC-2017-6262-3 and the INERCIA0 2018 project RTI2018-098865-B-C33.

Chapter 2

State of the art

The keywords: co-simulation, control systems in power electronics, QEMU emulator and SystemC language.

This chapter presents state-of-the-art related to the subject of the thesis. It is intended to give the reader a strong base and references for a good understanding. In particular, co-simulation and co-verification techniques for digital devices such as system-on-chip will be discussed. Then, it will show current methodologies and trends in implementing digital devices for controlling electrical systems. Finally, details about the emulator and the language used to achieve the goals of the thesis will be detailed.

2.1. Co-simulation based on virtual platforms

In the last decades, simulation has been the most used step to verify any design. This is because it allows reducing design cycles drastically. The first academic source to show the qualities of simulation is The Art of Simulation (1964) [VT64]. This paper explained how to built and use an industrial plant simulator with multiple machines. Nowadays, there are simulations in almost all engineering areas: digital systems, analogue systems, thermodynamics, aerodynamics, energy, economics, etc. According to the last Aspencore Embedded Market survey [Eva19], for years, multiple engineers have reported that the simulation stage has been the most critical stage for the computer systems community, followed by emulation.

A *simulation* represents the behaviour of a real system over time. Therefore, it allows us to know how the systems work, even before the real system has been built. To simulate a real system, it is necessary to make a model of the real system that represents its functionality. Different degrees of detail can be achieved in a simulation, and the observability of the model takes priority over the speed of the simulation. The most basic simulation (see Figure 2.1) consists of a *simulation model* and a *solver*. The solver performs the model calculations by obtaining the outputs as a function of the inputs for each iteration.

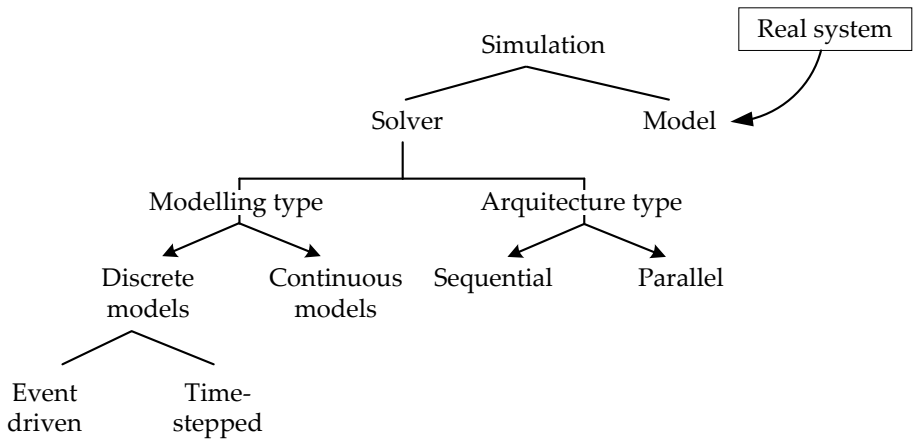

Figure 2.1: Basic composition of a simulation environment [Figure adapted from Figure 2 in [PVL+17] and Figure 2.1 in [Fuj01]].

According to the *modelling* of the real system and the *architecture*, different types of solvers can be used to manage the models (see Figure 2.1) [PVL+17]. It can be classified into two categories in function on modelling used: discrete models and continuous models. Furthermore, the architecture can be sequential or parallel.

Continuous models are used for systems whose states change continuously over time. They are usually described using differential or algebraic equations. Otherwise, the states of a *discrete model* change only at specific instants. The process of converting a continuous model to a discrete model is called discretisation. In a discretisation, the discretisation frequency is used to define the minimum period to wait to update the model's state. A discretisation always implies discretisation errors. This is the same concept that is used to convert digital signals to analogue and vice versa.

In the case of solvers for *discrete models*, there are two *time-flow mechanisms*: event-driven and time-stepped. The time-flow mechanism defines how the states of each model should change when time advances. On the one hand, the *event-driven* mechanism, also called *variable-step* solver, updates the models only when "something interesting" happens. That is when the state of the model changes. The instant at which the change of model state occurs is called an event. The event is an abstraction used to model an instantaneous change in the modelled system. For instance, a value change in a clock signal is an event. Each event has a *timestamp* and a list of possible actions to be executed when that event is simulated.

On the other hand, the *time-stepped* mechanism, also called *fixed-step* solver, periodically updates the states of the models, whether something interesting happens or not. This could reduce simulation performance, especially in systems where the frequency with which models change is low. Time-stepped solvers are often used in situations where, using an event-driven solver, the concentration of events is so high that it does not allow the simulation to proceed at an adequate speed. In addition, they allow maintaining a constant temporal accuracy in all models.

Even-driven solvers are the basis of sequential *Discrete Event Simulator (DES)*. A DES incorporates a *state machine* that manages all events in the simulation. An example of a DES state machine is shown in section 2.4. Usually, the state machine manages three data structures.

- The variables that describe the state of each model. A change in state variables means a change in the state of the existing system resulting from processing an event.
- A list of events containing the events to be processed. This list can change dynamically throughout the simulation, depending on the results of each model. If an event is generated due to the last event, the DES incorporates it in the event list.
- A global clock. This clock is used as a time axis for all models in the simulation.

From an architectural point of view (see Figure 2.1), a *sequential solver* employs an inherently sequential algorithm to manage the events of all models. In sequential solvers, synchronisation between all modules is relatively simple. However, its main drawback is that simulations of large real systems are limited by their sequentially. The solution could be to parallelise the sequential algorithm by creating a *parallel solver*. The main advantage is that the execution time of the simulation is reduced by parallelising the resources. However, the complexity of synchronisation in a parallel solver is considerable [Dom16]. Works such as [DLS17] has parallelised sequential DES simulators such as SystemC. Although they have been shown to increase simulation performance, the complexity of the synchronisation algorithms is proving to be an impediment to it usability. The option of combined simulators with sequential and parallel solvers is desirable as it would allow mixing the advantages of a simple synchronisation with the performance of a parallel solver.

As with simulation, *co-simulation* is not a new concept. In the last three decades, authors such as Becker [BST92], Fujimoto [Fuj01] or Fitzgerald [FLV14] have deeply studied the background of co-simulation. Other authors such as [NBTN17] or [PVL+17] have highlighted the usefulness of co-simulation in verifying the cyber-physical. The concept of co-simulation is related to running multi-domain simulators simultaneously to verify the whole system. In addition to the simulators, a master algorithm is needed. The *master algorithm* configures, initialises and synchronises the simulators. It also manages the information and event traffic between the simulators. Figure 2.2 shows an example layout for a co-simulation using two simulators.

The distribution of simulators in a co-simulation depends on the type of application. Currently, there are two approaches, mono-process and multi-process setup (see Figure 2.3).

A *mono-process co-simulation*, also called monolithic co-simulation, includes all simulators in the same process from the point of view of the operating system. The main

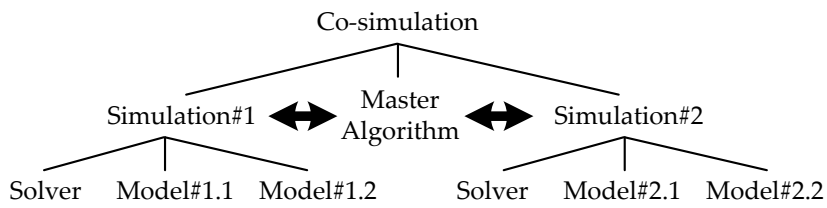

Figure 2.2: The composition of a co-simulation based on two simulations.

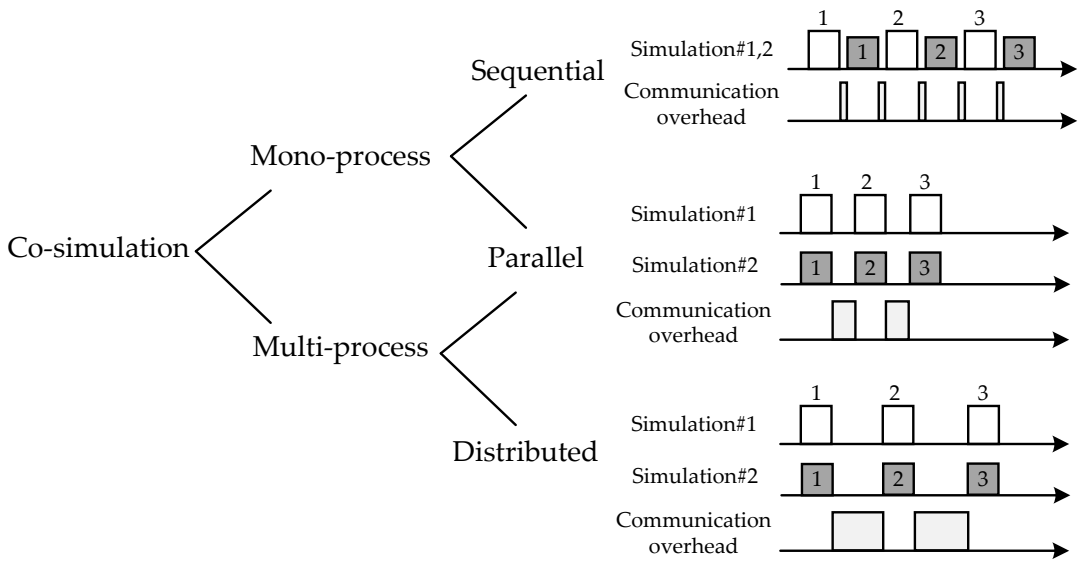

Figure 2.3: Types of co-simulación.

drawback of this distribution is that the overall performance of the co-simulation is low. Usually, it does not take advantage of the available host resources where the co-simulation is running. Actually, using the host resources depends on how well the operating system manages them. Nevertheless, this setup is the easiest to build and has the most debugging features. This is because all modules and variables of the co-simulation can be accessed from a single process. Another advantage is that it has minor penalties for communication and synchronisation between simulators. Although it uses a single process, the simulators and the designs of each simulator can be parallelised through threads. Therefore, we can have a *single-process parallel co-simulation*.

In the case of *multi-process co-simulation*, each simulator uses its own process. Therefore the workload of the co-simulation is parallelised. If all processes run on the same host machine, it is called *parallel* co-simulation. If the processes run on different machines, such as servers, it is called *distributed* co-simulation. Besides the location, the communication channel is the other main difference between parallel and distributed. In the parallel case, it often uses communication mechanisms of the operating system itself, such as Inter-Process Communication (IPC) like sockets or shared memory. Because communication via sockets is slow [DBK+16], some works have used shared memory to synchronise both simulators. In the distributed case, machines typically communicate via Transmission Control Protocol (TCP). This option allows specialising the simulation

hardware to each simulator. The latest trend is to use the cloud-based simulations approach [NTB+18, RKK+19, HMUH19]. That is, using the high computational capacity of the cloud to perform co-simulations. The main challenge here is to maintain a coherent interface to connect simulators from multiple domains and vendors.

A primary aspect of any co-simulation is the execution speed. In real applications, it depends on many factors. Some of them are the workload of the models, the performance of the host or server used, or the penalties for communication between models.

When verifying complex digital systems such as microprocessors through simulation, the main drawback of a simulator is the speed of execution. The workload required to simulate complex devices is quite high. In order to increase the speed of simulation, complex models such as microprocessor models are often simplified. This simplification means that the simulated source code does not include all the details of the final code. For these reasons, other alternatives have been demanded to verify the functionality of microprocessors. Currently, one of the most attractive is to emulate the microprocessor.

An *emulator* is a software or hardware system that replicates the behaviour of the real system. Emulation allows the behaviour of a system to be reproduced at the same speed as the real system. This may involve simplifications of certain parts of the real system. Also, it is usual to run the emulation using the host's resources equivalent to the real system. In this case, speed is prioritised over the observability of the design. Examples of emulators are OpalRT (power systems)[OPA20], ZeBu Server by Synopsys (digital systems)[Syn21] or microprocessor emulators such as QEMU (computer systems)[QEM].

The main difference between an emulator and a simulator resides in the speed with which it reproduces the behaviour of the real system [McG02]. While the simulator focuses on test and develop behaviour, an emulator focuses on achieving real-time execution. The simulation speed depends on the level of detail analysed, while the emulation speed is as close to reality as possible.

A digital system can be composed of multiple devices such as processors, peripherals, memories, digital and analogue circuits, interfaces, etc. In order to be able to verify the whole board at the same time and evaluate possible design alternatives, a good option is to build a virtual platform. A *virtual platform* is software that represents the behaviour of a real platform; it means the board.

The virtual platform can be composed of multiple specialised simulators or emulators. For example, a virtual platform can be built to simulate a state-of-the-art memory model using a Hardware Description Language (HDL) simulator for the memory and a software emulator to represent the functionality of the processor accessing the memory. In such a case, a *co-simulation virtual platform* is used. Another example is shown in Figure 2.4 where an acquisition system is verified on a motherboard. In that example, the microprocessor uses the PCI Express interface to deliver the acquired data to a graphics card. In addition, HDL designs are used in digital devices such as FPGAs to handle

the signals acquired in the analogue stage. As it can see, multiple domains are merged in the same application. Although the co-simulation concept was initially intended to run multiple simulators, its synchronisation techniques can also be used with emulators. Thus it is possible to have a co-simulation virtual platform composed of simulators and emulators.

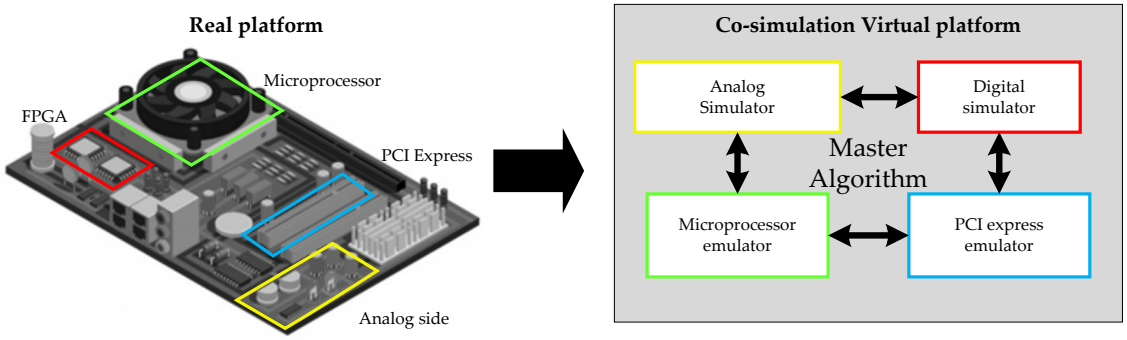

Figure 2.4: Example of a virtual platform for the co-simulation of a motherboard.

As it was mentioned, co-simulation can mix simulators and emulators from multiple domains. Therefore, nowadays, it is easy to find *hybrid co-simulations*. The hybrid co-simulations are complex as they mix different types of architectures, solvers and synchronisation mechanisms in the same tool. Some of the main challenges have been presented by multiple authors [BGL+15, PVL+17]: semantic adaptation, error validation, different step size, events synchronisation or standardized interfaces, and others.

2.1.1. Abstraction levels in simulation

One of the most commonly used techniques to reduce time-consuming is to simplify simulation models. Simplifying simulation models reduces the number of events and the workload of each simulator. Simulation speed decreases with the number of triggered events. Thus, co-simulation techniques have to deal with the *abstraction levels* of each model.

On the one hand, if abstraction of the simulation model increases, a lower level of detail and precision will be obtained, and therefore fewer events. On the other hand, fewer events mean more simulation speed. Figure 2.5 summarises the mains abstraction levels used in a hardware/software implementation flow.

Several works like [CYT11, CLP14, KYH16] have shown that the most appropriate level to verify both software and hardware is the Transaction level, which is detailed enough to check architectures temporarily but does not have all the details of Register-Transfer Level (RTL). In this thesis we focus on the Transaction level to increase the speed of the hardware and software co-simulation. However, the key to allowing flexibility in a co-simulation tool is to allow the use of multiple levels of abstraction. That flexibility

makes it easy to use design methodologies and to focus on simulation resources.

When multiple modules are described in different levels of abstraction, it is necessary to include an adapter called the *transactor*. A transactor translates messages exchanged between modules with different abstraction levels. This concept has been used for years by multiple authors in the literature [BFP06, HLGD18]. Its main use focuses on translating from Transaction-level Modeling (TLM) to Register-transfer level (RTL) and vice versa. An example of a transactor adapting TLM designs to RTL can be seen in Figure 2.6.

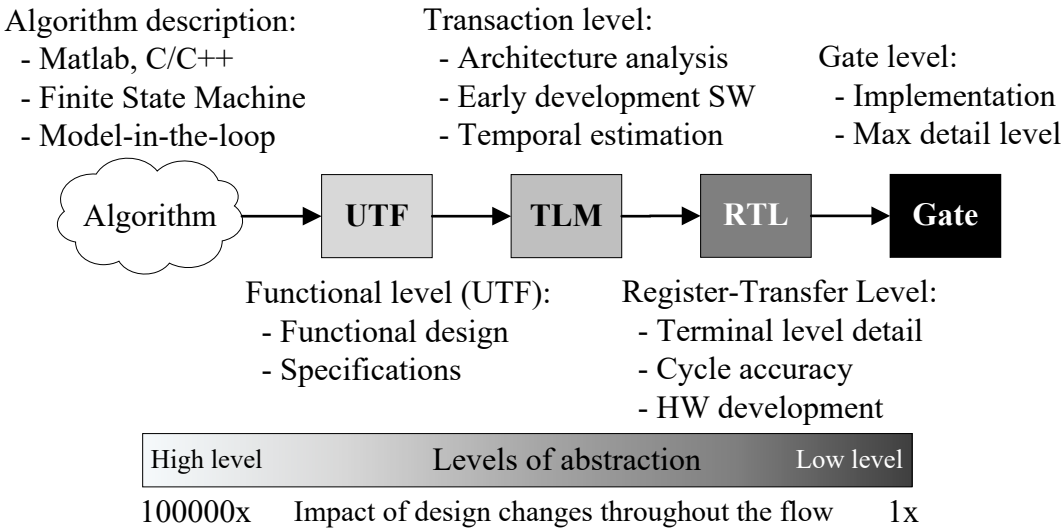

Figure 2.5: Levels of abstractions.

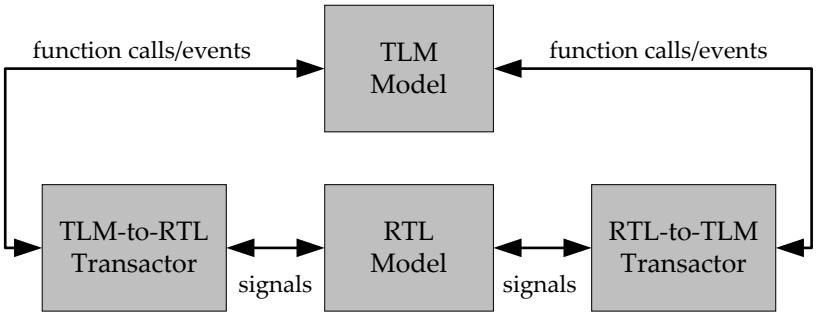

Figure 2.6: TLM-to-RTL transactor overview.

2.1.2. Co-simulation techniques

Co-simulation techniques are the art of orchestrating the execution of multiple simulators. A co-simulation deals with the speed/precision trade-off. Therefore, it is up to the user to choose the most appropriate option for each application. Orchestrate implies *timing and order* aspects will be taken into account in the execution of the co-simulation. Therefore, before presenting the different co-simulation techniques, it is necessary to define which *timing types* exist in a simulation [Fuj01]:

- Physical time: time of the real system.
- Simulation time: abstraction used to model the time being simulated. In multi-simulators setup, there is a *global simulation time*, i.e. the reference time for all simulators, and multiple *local simulation times* to each simulator:
  - Global simulation time: this is the time seen by all simulators. This time update and advances each time all simulators are synchronised.
  - Local simulation time: each simulator can advance at a different speed so each simulator can have its own local time.
- Wallclock time: time taken by the simulator to run the simulation.

It is possible to take 10 hours (wallclock time) to run a simulation of the interactions carried out during 1 ms (physical time) between 100 neurons. However, if we are only interested in the first interactions, we can stop the simulation when 1 ms (simulation time) has been simulated from the beginning.

The next step is to define when the simulators synchronise time to achieve consistent results. The co-simulation techniques used to come from previous work in the field of parallelised-DES [Fuj01]. Remember that the master algorithm is in charge of synchronising the simulators. The master algorithm manages the progress of each simulator taking into account the local simulation time of each simulator and global simulation time. In some cases, it will have situations where the local simulation time of each simulator is different. However, the timing notion between simulators should be the same when there is an interaction between them, which avoids any causality problems.

Traditionally, the master algorithm has used two synchronisation schemes: synchronous and asynchronous. In the *synchronous* one, the global time drives the simulation time, and simulators have the same local time, which is a copy of the global time. This limits the progress of each simulator to a regular time interval. Each simulator must handle all events before the end of its available time interval. This ensures that each simulator will not process an event before its local time. This scheme has high timing accuracy and is appropriate for applications where simulators interaction is high. However, its main drawback is the high message traffic between simulators when there are hardly any interactions. Note that each stopping to send a message introduces a performance penalty in the co-simulation. To reduce this penalty, some works such as [Mat06, DBK+16] have proposed to group all simulators in a single-process architecture instead of using a multi-process approach. The single-process architecture allows the use of shared memory to send information easily from one simulator to another. Therefore their communication penalty is minimised.

The *asynchronous* alternative is based on each simulator has its own local time, and its time advance is variable. In this case, each simulator advances freely as long as the

causality of the system is maintained. This avoids all unnecessary messages between simulators. The asynchronous synchronisation scheme must perform forward scheduling for each simulator. Its performance depends on the success of this scheduling. There are two possible schedules: conservative or optimistic (see Figure 2.7).

Conservative asynchronous scheduling releases the progress of each simulator as long as it is guaranteed that one simulator does not generate events that another simulator must process. This method aims to identify how much time one simulator can advance without impacting another simulator. This time is called *look-ahead time*. Its implementation is usually based on periodic synchronisation points whose period coincides with the look-ahead time. In this case, simulators run and stop at the synchronisation points to exchange messages. This schedule is appropriate for simulators that interact periodically over a long period. This is the case for control applications in power electronics, where the control algorithms are often divided into software and hardware modules. These applications are running periodically with a relatively low rate of interaction between software and hardware.

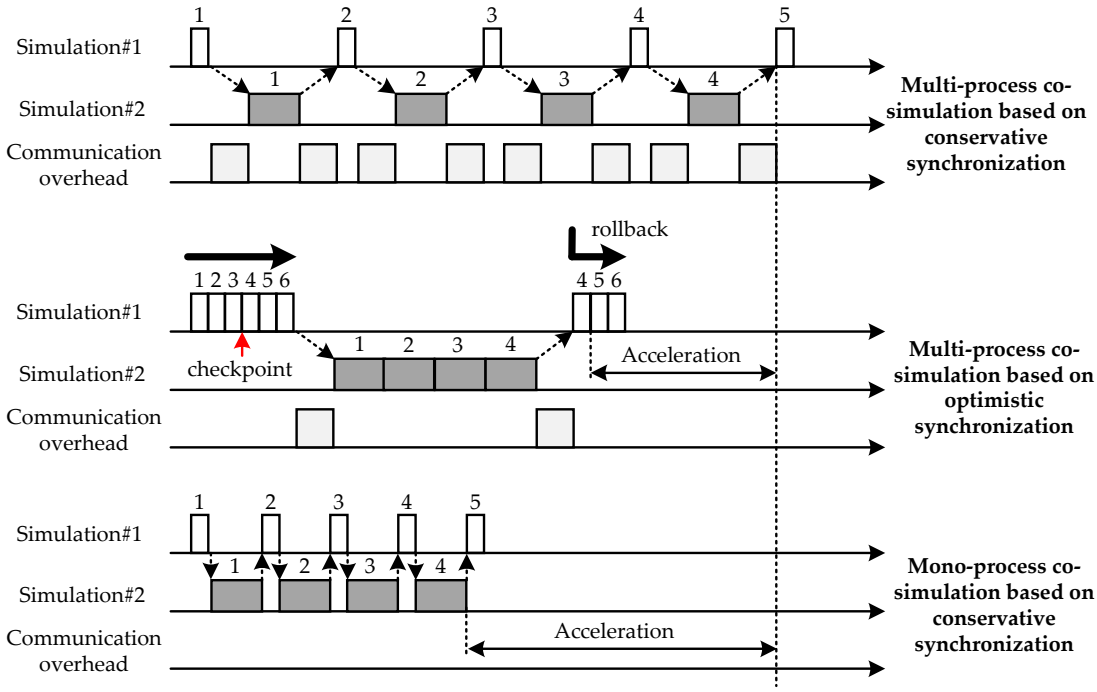

Figure 2.7: Example of conservative and optimistic asynchronous schedulings. [Mat06].

Optimistic asynchronous scheduling is based on letting each simulator run as far as possible, saving simulators' state in checkpoints. If an interaction requires some simulator to return to a past state, the checkpoints can recover the simulator state. Checkpoints ensure that there are no causality problems. The main drawback of this method is the excessive memory consumption of constantly saving each simulator's state. By contrast, it is the fastest option in situations where there are not many interactions.

The reader may have noticed that in Figure 2.7 multiple simplifications have been

made to clarify the explanation. Note that the three synchronisation mechanisms shown in the Figure are executed sequentially. However, as discussed in the previous section, it is possible to have single-process and multi-process co-simulations. Therefore, it is possible that the progress of each simulator is not sequential but also parallel. A second simplification is that the duration of each interval in which the simulator is running is fixed and limited. This is guaranteed for simulators using a fixed-step solver. However, it may be the case that some simulators use a variable-step solver. In this case, it has a hybrid co-simulation. Typically, the strategy to synchronise this type of co-simulation is conservative. The most restrictive or slowest simulator is usually used as the master of co-simulation [PVL+17].

Figure 2.7 also shows an example of single-process conservative synchronisation. The communication penalties are reduced compared with the multi-process distribution. Therefore a speed-up of the co-simulation performance is expected. However, the speed-up depends on each evaluated model, the number of interactions between each simulator, and the communication latency between each simulator.

2.1.3. Hardware/software co-verification

The term of *co-verification* has been defined many years ago and comes from the need to verify embedded devices whose applications interact between software and hardware [And05]. Embedded devices often mix processing blocks with hardware digital modules such as SoC+FPGAs. The verification process of a SoC+FPGA can be done in multiple ways. In most cases, verification usually includes a simulation stage that saves time and reduces hardware/software bugs. However, it is possible to skip the simulation stage and use systems that verify the software and hardware by emulation. Usually, such systems perform a distributed co-simulation based on specialised emulators. Examples of such equipment emulating software and hardware are Cadence Palladium, Mentor Veloce or Synopsys Zebu [Meh18]. Their main drawback is their high cost.

This section will focus on SoC+FPGA co-verification carried out on the same host workstation. In this setup, co-simulation tools are often used to verify software, hardware and their interactions simultaneously. Co-verification tools use and synchronise multiple simulators/emulators (e.g. software emulation with hardware simulation). If all tools simulate, it is referred to as co-simulation. When at least one tool uses emulation, it is often called co-emulation. To co-simulate or co-emulate an embedded device, it is necessary to built a virtual platform that represents the functionality of the embedded device or board and includes the software and hardware simulators or emulators.

The most used strategy to build a hardware/software co-simulation virtual platform is based on a *sequential Discrete-Event Simulator* (DES) or simulation kernel. The DES distributes the tasks between the software module (CPU-Processing System) and the hardware design (FPGA fabric-Programmable Logic) [Fuj01, CYT11, WMLA16]. Each

hardware/software can be seen as a module within the discrete event simulator. Moreover, each module can be made up of a simulator/emulator, and its synchronizations and communications are carried out through messages. Both modules can be described at different abstraction levels depending on the precision/speed trade-off that the user can assume.

In a discrete event simulator, state changes occur only through events [Fuj01]. The simulation run time increases with the number of triggered events. Therefore, a usual approach to improve simulation speed is to use higher abstraction levels with fewer events. As already mentioned, the best abstraction level to verify both software/hardware mixed designs is the *transaction level*, which is detailed enough to check designs with timing accuracy but does not consider all the details of RTL/gate-level that would slowdown the co-simulation.

The most common approach to build a hardware/software virtual platform consists of an open-source processor emulator (software) and a digital simulator (hardware). The emulator reproduces the software execution behaviour, such as QEMU [DBK+16]. The digital simulator verifies the behaviour of hardware modules. Some examples of digital simulators are OSCI SystemC kernel, QuestaSim by MentorGraphics, or VCN by Synopsys. SystemC enables modelling the hardware modules from the algorithm level to Register-Transfer Level (RTL) and there is an open-source simulator.

In this approach, the processor emulator is divided into of an *Instruction Set Simulator (ISS)*, which interprets guest instructions and executes them on a host. It also involves a *Bus Functional Model (BFM)*, which models the external processor interface and its connection with the surrounding hardware. Simultaneously, the hardware simulator behaves as the simulation kernel to distribute the simulation events in hardware/software modules (see Figure 2.8). Both software and hardware modules can introduce a high workload, implying a very slow simulation for complex systems. However, its use is very suitable during the early stages of the design flow.

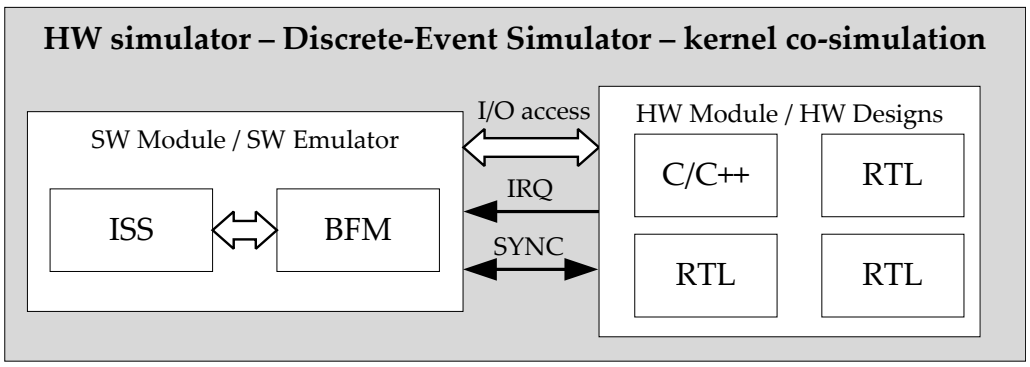

Figure 2.8: Block diagram of the general setup of DES-based HW/SW co-simulation virtual platform.

FPGA-based SoC devices increase the complexity of hardware/software mixed designs and, hence, their verification. Also, a trade-off between the timing accuracy and the

speed of the simulation is required. When software and hardware share information, it is necessary to simulate the hardware design located in the FPGA to verify software binaries. The reason is that the software results will largely depend on the hardware results. For mixed hardware/software designs, the difference of timing accuracy between two components is usually three or more orders of magnitude. A difference like this requires increasing the global timing accuracy of co-simulation, increasing the number of events to process.

SystemC, proposed as IEEE standard in 2011 and based on C++, has become the most common languages to build hardware/software interfaces for co-simulation virtual platforms. Some examples are QBox [DBK+16], COREMU [WLC+11], Simics by Wind River Systems [MCE+02], or PetaLinux by Xilinx. These co-simulation virtual platforms allow instantiating RTL modules modelled in SystemC and using the OSCI kernel (DES) of SystemC as hardware simulator and kernel of co-simulation. However, the SystemC kernel runs the simulation sequentially and has code limitations that prevent it from parallelizing its kernel [Dom16, BMC16]. Therefore, it introduces a penalty on complex FPGA-based SoC systems with multiple CPUs and RTL modules running in parallel. Contributions to parallelize SystemC are presented in [BSS+11]. However, these contributions have not yet been included in the SystemC Standard [IEE11]. Also, their use is limited to the academic field.

To reduce long co-simulation run times, some tools such as PetaLinux or QBox have delegated the responsibility to the software emulators they adopted. This way, they focus on improving the interface, memory management, or ecosystem. Like COREMU, others have chosen to create multiple modules to instantiate copies of the same software emulator in different operating system threads and control their progress. However, they have not presented a follow-up of the work, and their impact has been diminished.

Table 2.1 shows the current most commonly used open-source software (SW) emulators. They are OVPsim, Unicorn, Gem5, and QEMU. The main reasons for their success are their high portability to different architectures, and their updated repository and extensive documentation. This has led companies such as ARM, Xilinx, or Synopsys to adopt them for their products.

There are two critical factors to differentiate each emulator, the accuracy and the speed at which it executes. Accuracy can be at the functional level, the target instruction level or the target processor cycle level. Typically, the higher the level of detail, the slower the speed of the emulator. However, the speed of the emulator is also affected by the quality and type of engine used. If the engine uses virtualisation, the speed is usually higher.

OVPsim is a proprietary virtual platform emulator released by Imperas, which simulates complex multiprocessor platforms with arbitrary local and shared-memory topologies [Imp08, DXZ+13, LP15]. It offers open-source CPU models and free Application Programming Interfaces (API) to build processor, peripheral, and platform models. OVPsim's

Table 2.1: Comparison of most used software emulators.

SW Emulator	License	Engine	Accurate	Speed	Platforms Supported	Refs
Simics (WindRiver)	Proprietary	KVM: Multi-thread	Function	●●●●○	●○○○○	A
OVPSim (Imperas)	Open /Proprietary	DBT: Single-thread	Instruction	●●○○○	●●●●○	B
QEMU-DBT	Open	DBT: Single-thread	Instruction	●●●○○	●●●●●	C
QEMU-KVM	Open	KVM: Single-thread	Instruction	●●●●●	●○○○○	D
QEMU-MTTCG	Open	DBT-Parallel: Multi-thread	Instruction	●●●●○	●●●○○	E
Gem5	Open	DES: Single-thread	Cycle	●○○○○	●○○○○	F
Unicorn	Open	DBT: Single-thread	Function	●●●○○	●●●○○	G

A: [MCE+02, CYT11]; B: [Imp08, CLP14, LP15]; C:[Bel05, LP15]; D: [Bel05, MB16]; E: [CBBC17]; F: [BSS+11, BGOS12, MCJW17, AAKMK18]; G: [ND15, JWLA19].

main advantages are extensive documentation and support for different processor architectures. However, OVPsim does not allow building cycle-accurate models but rather instruction-accurate models [CLP14]. Moreover, [CLP14] shows that OVPsim is slower than QEMU. Also, it supports fewer target processors.

Gem5 is a microarchitecture simulator. Nevertheless, it supports a full-system mode, which enables it to be used as an emulator. It provides cycle-accurate precision since it is based on a DES. Therefore, its simulation speed is not as high as QEMU or OVPSim because it does not take advantage of a Dynamic Binary Translation (DBT) engine or the Kernel-based Virtual Machine (KVM) engine. Although the use of KVM is extended to virtualize desktop machines [KDB+18], its use to verify embedded devices is not very extended.

QEMU is currently the open-source instruction-accurate software emulator adapted to the highest number of guest processors. Initially, it was created using a sequential execution. However, the increase in the number of CPUs (or cores) has forced researchers to look for alternatives such as the KVM engine or the Multi-Threaded Tiny Code Generator (MTTCG) engine. These options speed up emulations of multi-core platforms like symmetric multi-processors operating systems [QEM], parallelizing the load of QEMU in the host CPUs. Multiple works have been published describing its use to build high-performance hardware/software co-simulation environments [CYT11, LP15, DBK+16]. Instruction-accurate is considered enough for functional verification of MPSoCs [CLP14].

As a simplified version of QEMU, Unicorn [ND15] improves the framework and flexibility of QEMU, thus making a safe software emulator. Nonetheless, it offers fewer features than QEMU, and it has been ported to only a few hardware platforms. Its use is limited to

research purposes [JWLA19]. Two main approaches have been used to integrate QEMU as software emulator into the co-simulation virtual platform: mono-process approach and multi-process approach.

In the mono-process approach used in [DBK+16, CKCL18], QEMU is included as a dynamic library in the virtual platform. Thus, the software emulator and the hardware simulator run in the same process, allowing the synchronization and communication mechanisms to be implemented as simple function calls.

The multi-process approach is based on the use of an inter-process communication mechanism. The last versions of PetaLinux toolset by Xilinx include a new feature to connect QEMU with an external simulation environment, such as SystemC. This approach integrates simulators (QEMU and SystemC-DES) as different OS processes, sending transactions and synchronizes time through UNIX sockets. Such a solution provides flexibility. Nevertheless, the main weakness is the limited simulation speed it provides. This applies specifically to systems that present high Input/output acceses (I/O) rates in which the interprocess communication overhead significantly reduces the co-simulation speed. This is an important aspect when the synchronization message rate is high. Another disadvantage of the QEMU emulator is that the multi-host thread feature is not supported, forcing the reduction of simulation speed and sequential execution. Therefore, its use for the verification of target platforms running multi-core operating systems is limited.

QBox or "QEMU in a box" [DBK+16] is an in-progress solution to develop a QEMU version equipped with a TLM2.0 external interface to facilitate its integration in a hardware simulation environment. This virtual platform allows co-simulation using SystemC, and it provides shared libraries that contain QEMU-based CPU models. Currently, QBox only supports three types of ARM CPUs. Although it is a desirable open-source solution for heterogeneous architectures, currently, it does not support the multi-host thread for fast OS simulations (similar to PetaLinux).

2.2. Digital electronics for controlling power electronic converters

In the last few decades, Power Electronics has acquired great prominence as a multidisciplinary technological and scientific area that studies the control and conversion of energy flows using electronics as a reference. These areas include applications from microelectronics to large power systems, where the aim is usually to increase energy efficiency, either by reducing consumption or avoiding unnecessary losses through the use of different control techniques.

The background of power electronics has been widely studied and developed in the last decades and is described by authors such as Mohan, Bose or Trzynadlowski, and

others ([MUR09],[Bos97],[Trz10]). Currently, there are many research areas of interest such as: smart grids, electrical machines, industrial drivers, power converters, renewable sources, storage systems, energy transportation, automation technologies, control systems, robotics, cloud computing or Internet-of-the-Things applied to industry [IEC19]. Figure 2.9 shows an example of the most popular areas of power electronics [Soz17]. The current thesis focuses on making contributions both in *embedded software and hardware design* and *circuit simulation* areas (highlighted in Figure 2.9 with grey sections).

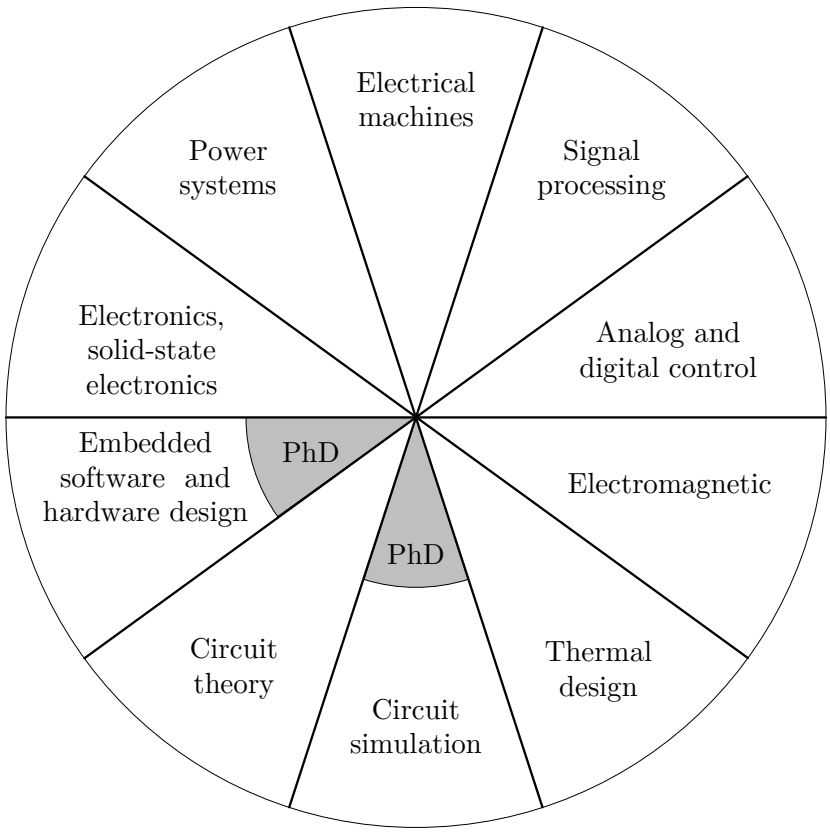

Figure 2.9: Main areas of power electronics and contributions of the thesis.

The basic power system consists of a power circuit/equipment called the *power plant*, which transfers the energy, and the *control system*, which measures and controls the power plant. Figure 2.10 [Soz17] shows the diagram of a basic power system. The control system, also called a controller, is based on real-time feedback/feedforward and measures the state of the power plant using voltage and current sensors, both at the input and output. It then processes the measurements and generates the necessary excitations or changes in the plant to meet the desired control reference or set points. The control system is usually implemented on the *control board* that includes all the digital and analogue circuits for measuring, processing and driving the signals that control the power plant. Examples of the power plant are converters, electrical machines, power generators, power loads or power transmission networks.

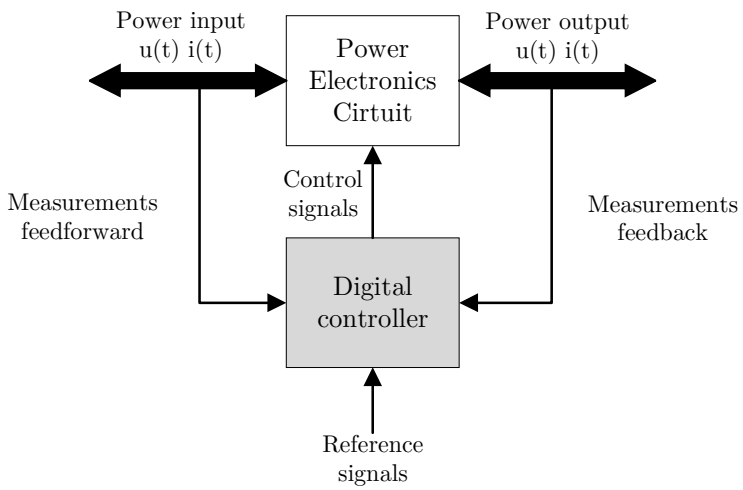

Figure 2.10: Basic power electronics system.

2.2.1. Control system development

Control system designs for high power systems are considered critical systems that must be reliable. In addition, such systems have certain particularities. Generally, the following aspects must be taken into account [ELM14]:

- Time accuracy. The frequency of the grid is usually in two ranges, 50 or 60 Hz. One of the main goals of control techniques is to reduce the losses generated by harmonics in the grid. The prominent harmonics are usually in the Hz to kHz range. To detect these harmonics, it is necessary to sample the power plant at frequencies below 1 MHz. Also, it is required to have processor units with clock frequencies in the order of GHz in order to be able to process the measurements in real-time. The synchronisation between the acquisition system and the processing unit is crucial to meet the time requirements. Such synchronisation is usually imposed by the control frequency with which the control algorithm was designed. Any fault of synchronisation or latency added to the control frequency can generate disturbances that make the system unstable in a steady state.

- Acquisition resolution. Power plant measurements must be transformed from an analogue to a digital world using Analog-to-Digital Converter (ADC) devices. The number of bits in these ADCs limits the accuracy of the sensors, constraining the resolution that the control algorithm can have. Currently, ADCs of up to 20 bits are commonly used. These higher numbers of bits can be obtained using sigma-delta. As a consequence, the latency of ADC increase. In power electronics applications the low latency in the acquisition system is vital. Therefore, most applications use ADCs with a resolution between 10 and 14 bits.

- Computing precision. Control algorithms are usually designed to operate at a precision limited by the control board, typically 16 or 32 bits (single-precision floating-

point format). Therefore, the control board must be capable of meeting the precision requirements of the control algorithm. Close attention must be paid to the *quantization errors* introduced and their impact on the algorithm control.

- A high number of digital input and output. This type of system is distinguished by having a high number of *input/output* (IO). A device can have hundreds of inputs and outputs. They have to obtain real-time information from dozens of sensors and generate excitations on dozens of drivers and relays driving the power plant.

- Robustness to failure. One of the most important features is the ability of these systems to detect a fault and react before the fault causes damage to equipment or staff. This usually includes monitoring the power supply, proper operating, temperature, and any other faults that the control system may detect and, if possible, record. Fault management should never be stopped and should be redundant.

To meet the above requirements, devices such as low-cost Microprocessors (μP), Digital Signal Processors (DSPs), Field Programmable Gate Arrays (FPGAs), Systems-on-Chip (SoC), or a mixture of the above, have been used [BDD19]. Today's most commonly used option is a mixed solution made up with a processing unit, it means a μP, a DSP or a SoC, coupled to an FPGA. The processing unit run the software implementation of control algorithms, and the FPGA implements the application-specific peripherals, also called hardware accelerators. While the processing unit makes it easier the control algorithm implementation, the FPGA complies with the interfaces, latency, throughput and reconfigurability requirements.

FPGAs have been used extensively over the last decade in power electronics. Initially, they were used as a hardware platform to implement peripherals in tandem systems based on a DSP plus an FPGA. The DSP, usually floating-point, provided the necessary performance to allow real-time execution of the control algorithms. At the same time, the FPGA was used to implement the specific peripherals (Pulse-Width Modulation (PWM) generators, acquisition controllers, etc.) due to the limitations of the DSP. Some authors, such as [ZYW+20], have used a DSP+FPGA system to perform predictive control of a Static synchronous compensator (STATCOM). In [BCR+08], the authors use a DPS+FPGA design to control a back-to-back converter for wind turbines. In [BHR+09], the authors present an industrial interface for grid converters that is also based on a DPS+FPGA design. The advantages of using DSP+FPGA tandem have also been demonstrated in [MMB+13], where the authors use a DSP to include a current prediction algorithm applied to multilevel converters.

The inclusion of hardware multipliers into FPGAs enabled their use in reconfigurable computing applications [NI14] and the hardware implementation of control algorithms. These hardware multipliers have evolved into complete DSP cells, which incorporate all the necessary elements to implement the accumulative product or Multiply?accumulate

operation (MAC) operation, which is the fundamental operation to be performed in a large number of discrete control techniques. Current FPGAs have many DSP cells, which implies high parallelism to meet the most demanding timing requirements. Works such as [MIN11] have pointed out that implementing control systems with switching frequencies above 1 MHz can only be addressed by hardware implementation of the algorithms in FPGAs.

As FPGAs grew in capability, it became possible to incorporate in a single device all the resources needed to implement a microprocessor-based system as described above, giving rise to the *System-on-Chip* or SoC concept. The first FPGA-based SoCs used to include low-performance processors, which could be placed more in the category of microcontrollers. Since the computing performance was relatively low in most applications, they were used as supervisors of the control system. In contrast, the control algorithm was implemented in hardware using the logic resources of the FPGA [BIME13]. In this case, the hardware implementation appears in the SoC architecture as specialised peripherals that communicate with the processor.

The design of these hardware accelerators is a complex process that requires considering multiple factors that define their internal architecture. Some factors are the arithmetic to be used (fixed point or floating point), the size of the internal datapath, the replication of functional units (operators) that allow multiple operations to be carried out in parallel, etc. The selection of these factors can directly impact the performance of the SoC (consumption of logical resources, maximum operating frequency) and on the performance of the control algorithm itself in terms of bandwidth [BC14] or stability [YMK14]. All this turns the design of hardware implementations of algorithms on FPGAs into a complex task, and it usually requires in-depth knowledge and experience in hardware design. These are the main reasons that have delayed the adoption of this type of technology, as pointed out by several authors [RAVPM15].

The introduction at the beginning of the 2010s of *High-Kevel Synthesiser (HLS)* has contributed significantly to easing this situation. An HLS synthesiser (also called behavioural synthesiser) makes it possible to automatically generate the hardware designs of the controller and its subsequent implementation in HDL languages (Hardware Description Languages). HLS tools start from a behavioural description written in C/C++. Then, a set of directives are added, defining the required architecture performance in terms of high-level parameters such as maximum latency or minimum throughput. Some of the most popular HLS tools are Vivado HLS by Xilinx, Catapult by Mentor or HLS Coder by Matlab. The first work describing the use of HLS for the design of power converter controllers dates back to 2013[NLB+13]. In [SMB+13], the implementation of a grid synchronisation algorithm described in HLS is analysed, and a comparison is made with its counterpart version in Very High Speed Integrated Circuit Hardware Description Language (VHDL). The study concludes that HLS designs tend to be more resource-intensive

than designs described in HDL (VHDL, Verilog). In addition, with HLS, there are limitations concerning the latency obtained in the hardware design. However, other works such as [KLG11, MFR20] conclude that HLS can achieve similar power consumption and latency as if HDL languages had been used. There is no conclusion about the best option between HLS or HDL because the result depends on the particularities of each design. The success of the HLS tool depends on two factors: the quality and performance of the synthesiser and directives optimisation or *#pragmas* that are given to the synthesiser.

The processors in FPGA-based SoCs have also been evolving, presenting more sophisticated architectures that offer better performance. The processors integrated with FPGAs support two formats, depending on the method used for their implementation: soft cores and hard cores [Nur07]. *Soft cores* use the FPGA logic for their implementation, so their performance is usually quite limited. For this reason, their use is usually restricted to system management and supervision tasks or to implement control algorithms with low timing constraints [JLU+14]. In *hard cores*, part of the silicon area is reserved for implementing the processor, which allows it to achieve performance similar to an Application-Specific Integrated Circuit (ASIC). The current trend is to also include in silicon all the infrastructure necessary for the processor to operate (standard peripherals, cache subsystem, external memory controllers, etc.). The programmable logic part is used to implement specific peripherals for the application or hardware accelerators. An example is the Xilinx Zynq family (ARM Cortex-A9 dual-core + FPGA) [NI14]. These are certainly high-performance processors, which even include vector computing units, enabling the software implementation of more demanding control algorithms in terms of execution times such as [TPD16]. Another example of the use of Zynq is [HWWR20], where authors use this SoC to control a permanent magnet synchronous motor and analyse its disturbances.

Multi-core SoC devices allow specializing the cores in specific tasks, achieving a balance between flexibility and efficiency. Such is the case of the work described in [PC14] working with a dual-core SoC, in which one of the cores runs an embedded Linux, which facilitates its connectivity with the external world. The other core runs the control algorithm under a real-time operating system (Real-Time Operating System (RTOS)), much lighter than Linux. The implementation of such distributions can be facilitated by multi-core communication management libraries such as OpenAMP. One of the most significant features of the last SoC families is the integration of MultiProcessor architecture or *MPSoCs*, which provides excellent flexibility, allowing the control algorithm to be distributed among several microprocessors. An example of MPSoC is Zynq Ultrascale, including in the same device four ARM Cortex A53 cores, two Cortex R5 real-time processors, and an ARM Mali-400 GPU [ABG+16], which will allow the implementation of more challenging control systems.

Although power control applications are not one of the main markets of SoCs and FP-

GAs (their market share is less than 5%), it has benefited from the development of these devices. In recent years, investment in developing devices that enable realizing artificial intelligence techniques such as machine learning or neural networks has been very significant. An example of this investment is the latest Xilinx family called Versal Adaptive Compute Acceleration Platform (ACAP) [Xil20], whose development cost has been estimated at over a billion dollars. Versal presents an evolution in heterogeneous computing devices. The new SoC architecture incorporates a module called Artificial Intelligent engines (AI) Engines. The AI engines are an array of Very Long Instruction Word (VLIW) processor with 512-bit Single Instruction, Multiple Data vector units (SIMD) vector units and memories. They also include DSPs in the FPGA fabric that support 58-bit and single-precision floating-point operations, greatly increasing the resources available for implementing control algorithms.

In [ZBW21], the authors present a state-of-the-art review of artificial intelligence in the area of power electronics. The authors summarise state-of-the-art in three branches from the application perspective: design (design time reduction, modelling and optimisation), control (control tuning, estimation, fault-tolerant operation, modulation, energy management) and maintenance (system parameter identification, data processing, anomaly detection, diagnostics, lifetime prediction). Although Versal ACAP has been developed for other applications such as data centre, automotive or 5G, it is also a great device to implement cloud/edge computing or machine learning solutions in power electronics applications where the cost of the control system is not critical aspect. If the cost of the control system is critical, System-on-Module (SoM) devices [Xil21a, Ins21] are expected to become industry standard in the next decade. This approach integrates a SoC+FPGA and memories on a single board module, reducing control board design and development costs.

To conclude, the option of simulating the behaviour of the whole control system is extremely interesting. It allows a significant reduction in the effort and time spent during the development and verification of these systems. However, its heterogeneous setup, where part of the functionality is implemented in software and part in hardware, requires the use of different simulators that operate jointly (HW-SW co-simulation) to reproduce the behaviour of the different parts of the system and the interaction between them [CLP14, PVL+17]. In the simulation, the SoC interaction with the power plant to be controlled must be added. Typically, the power plant behaviour is simulated using a continuous-time simulator such as Matlab/Simulink.

2.2.2. Control system verification

The control system infrastructure has to acquire data from the sensors, analyse it, detect possible faults, and drive devices like contactors or Insulated Gate Bipolar Transistor (IGBT) of the power plant. In addition, communication functionalities are usually

integrated for monitoring. Thus, the usual architecture used for the implementation of the control system is shown in Figure 2.11. In this architecture, the design is divided into three stages: *acquisition, controller execution* and *output/driver management*. Typically, the controller stage is further divided into multiple hardware modules and software modules. Therefore, such systems usually use multiple modules interconnected sequentially. In other words, a failure in acquisition impacts the behaviour of the control algorithm and can lead to incorrect behaviour of the plant. Thus, its verification is complex. The modules and their interconnections must be taken into account to verify the operation of the whole control system.

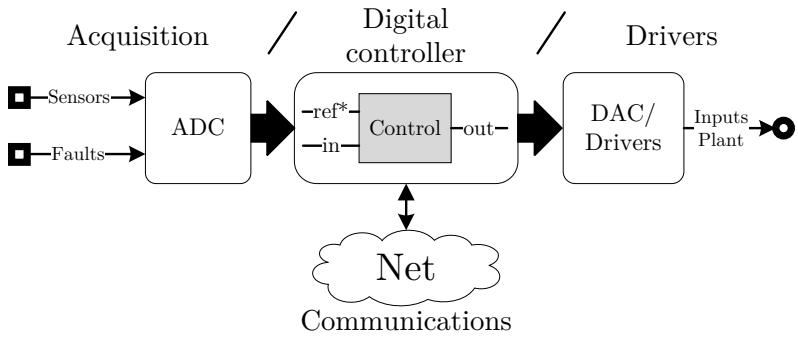

Figure 2.11: Typical controller block diagram in power applications.

The verification of the control systems must be carried out under very controlled conditions and is an essential step in developing the power plant. Usually, they control power equipment operating at thousands of kilowatts. An unexpected control system failure can cause irreparable damage to power equipment and presents a high risk to staff.

The most extended technique for testing software is debugging. The main drawback of verifying control algorithms in power electronics is that they are implemented in real-time systems. Consequently, it is not possible to stop the full control logic to apply debugging techniques. This debugging could stop the execution of the controller, leaving the power plant out of control.

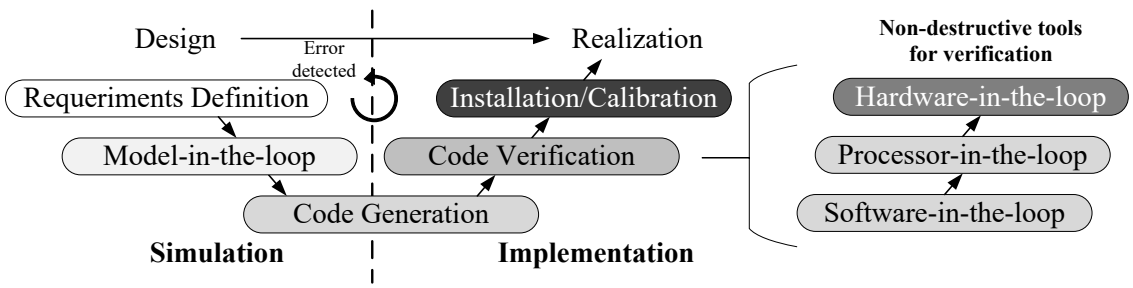

Figure 2.12: Design and implementation flow of a control system in power electronics.

To reduce the risk of failure during the controller design and implementation, techniques have been standardised to verify designs by simulation or emulation of the controller and power plant gradually. The standardisation includes a design and implementation

methodology based on the V-Model used in hardware and software system development. That methodology is described in ISO26262 as part of the IEC 61508 standard Functional safety of electrical/electronic/programmable electronic safety-related systems [IEC10]. It is an evolution of the classical waterfall model by performing a review at each stage and allowing errors to be detected before reaching the final code version. Figure 2.12 shows the design and development flow specialised for control systems in power applications.

This methodology is based on two main stages: *simulation* of the plant and the control algorithm and *implementation* of the control system. In turn, each stage is divided into multiple phases that help to develop the control system gradually. The jump from one phase to another always involves verification, which allows detecting fails early. Whenever an error is detected, the simulation stage must be repeated to check the impact of the error and fix it.

The simulation stage involves the preliminary study, modelling the plant, modelling the controller, and simulation of the whole system. Then again, in the implementation stage, it programs the controller and the verification and installation of the control system is performed. The verification process is usually based on the use of *non-destructive systems*. It means, different systems that allow a closed-loop verification of the controller in safe environments. These systems also help to debug the control system safely.

The initial phase of any design is based on the study of the necessary specifications and theoretical approach. The duration of this phase depends on the difficulty of the used control technique and the plant requirements. In this phase, no infrastructure is required, and no code is developed.

Once the power plant and control system have been defined and studied, model-based simulation tools are used to study the plant and simulate its behaviour with the controller. This technique is called *Model-in-the-Loop (MIL)*. The modelling of the power plant and its reaction to specific excitations can be done theoretically. However, the huge productivity of model-based simulation tools has made it the most widely used option. Companies such as PSIM [PSI20] or Mathworks with Simulink [Mat20] have offered mature simulation environments that are widely used in the literature [Plu06]. Nowadays, simulation is a necessary step in designing control algorithms to understand and verify their performance and behaviour.

In the MIL phase, one of the most critical steps is the modelling of the plant. Multiple techniques are associated with plant modelling to ensure that the plant model behaves precisely like the real plant. However, the most commonly used option is to build the plant model using predefined models from the simulation tool, which has been previously verified. These predefined models are the main added value of this type of tools.

Once the MIL simulation of the whole system, power plant plus controller, has been completed, the requirements to be taken into account for implementing the control system are proposed. Only the simulation tool is needed for this phase. Until now, the phases are

focused on the design. However, the following steps are focused on the implementation of the controller on a control board.

When the control algorithm has been designed, it must be decided how it can be implemented. That is, which parts will be implemented in software (CPU) and which parts will be implemented in hardware (FPGA), taking into account the actual constraints of the control board.

The next step is to program and implement the control system on the control board. This step is often referred to as *integration phase* and is the most problematic step. Here all the low-level details are taken into account. This is when all the problems of moving from a theoretical system to a real system arise. The integration phase is the one that often causes the most delays in project delivery. The integration of the whole system must be well verified. Not only must be checked the correct implementation of the software and hardware but also their communication and the timing requirements are met.

As mentioned before, when working with high powers, the verification of the control system implementation is a dangerous step. Typically, the operation of the control board is verified against a non-destructive emulation system to reduce risks (see Figure 2.11).

In literature, the most used non-destructive emulation system is based on *Hardware-in-the-loop* (HIL) techniques (Figure 2.13a) [PMIG19, JHT+20]. The first HIL was a flight simulator in 1910 made by Sanders "Teacher". Throughout the last century, they have been used in countless industries such as defense, aerospace, energy, automotive and transportation, maritime, etc. However, it was not until the 1990s that they became commercially available. Its broad applicability has led to multiple definitions, and nowadays, there is no standard definition of the system. It was not until 2019 that the term was included in the IEEE taxonomy [IEE20]. Although more than a dozen of definitions can be found, they all have in common that they describe a system that makes it possible to represent the behaviour of a physical system in real-time. Hence, some authors refer to HILs as emulators or real-time simulators.

In the area of power electronics, HIL systems use complex algorithms to represent the physical models of the plant and require detailed hardware designs. HILs require precise hardware designs that typically guarantee a simulation step in the order of microseconds (e.g. for high-speed power switches). Thus, it allows to emulate the power plant and interact with the control board in real-time. Such temporal constraints can only be achieved by HIL systems using FPGAs. Furthermore, FPGAs also bring a great deal of reconfigurability and flexibility to the verification process.

Currently, companies such as OPAL-RT, dSPACE or National Instruments have developed mature products capable of emulating the expected behaviour of the power plant. Nevertheless, there are some constraints. The power plant must be built using pre-defined models.

The main drawbacks of HILs are their high cost, and the test can only be done at a late

development stage. There are also limitations in the timing accuracy of the power plant [KC08] and compatibility with the control board [BMV13]. Usually, the control board are not standard solution and they interface must be compatible with HIL equipment. That makes a slow integration.

Other drawbacks of the HIL reported by authors are that it works as a black box, i.e. there is no detailed internal information about the emulated power plant [Ima14]. Also, the simulation speed is usually constant and does not allow for acceleration, slowing down or pausing [Ima14]. It is in real-time, and this affects the debugging capability of the control board.

International projects, such as ERIGrid [VRN+19], have proposed linking HILs from different laboratories, sharing equipment costs. Other authors, such as [TIM+19], have developed low-cost HILs called Embedded Real-Time Simulator (eRTS). eRTSs integrate the power plant simulation model into SoC+FPGA devices. In this case, a C-equivalent code of the discretised plant model is obtained. Later, it is compiled on the FPGA using HLS tools to deal with the complexity of the simulation models emulated [BSD+20]. However, some authors such as [MOBD18] have concluded that HLS has limitations about the size of the power plant that can be synthesised in the HIL. Then, its main limitation is the capacity of the FPGA itself.

There are cheaper alternatives to the use of HILs that allow verification of controller and power plant operation at the same time. The most commonly used in the literature are Processor-in-the-loop (PIL), FPGA-in-the-loop (FIL) and Software-in-the-loop (SIL).

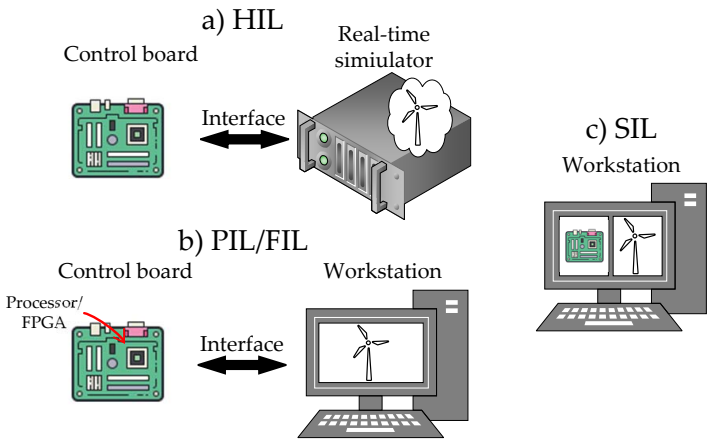

Figure 2.13: Non-destructive tools for verification in power electronics: a) HIL; b) PIL/FIL; c) SIL.

Some simulators allow connecting to the control board and share messages. This technique is called *Processor-in-the-loop* (PIL) or *FPGA-in-the-loop* (FIL) (Fig. (PIL/FIL) (Fig. 2.13).

The PIL technique is used to communicate with a processor, while FIL communicates with an FPGA. Both techniques are similar. Communication with the processor or FPGA is performed, and the software/hardware module to be verified is installed. This module

can be obtained using HLS tools, or it can be implemented manually. Once this code has been installed on the control board, the simulation advances, generating the module inputs. The input data is then sent, and the module is executed on the control board. As a result, the outputs are sent to the simulator, which was waiting, continuing the simulation. From the simulator's point of view, the control board is a black box.

The first restriction of these systems is the simulation speed. The simulation speed is limited by the slowest module, either the simulator (workstation), the processor (PIL) or the FPGA (FIL). The speed will depend on the workload of each module and the latencies of the communication channel between the simulator and the control board.

Both PIL and FIL are great options for verifying modules and provide a useful way to make a preliminary study of the performance (resources and timing) of specific software/hardware module on the final platform. Some works have taken advantage of these types of techniques to implement onboard control quickly. In [GTF+19] authors verified the performance of their control algorithm for a Doubly Fed Induction Generator, or [CNVT12], where they analysed the performance of different controllers Linear Quadratic Gaussian controllers to minimise the impact of wind turbulence on wind turbines.

To carry out the verification with PIL or FIL, the simulation tools usually provide all the utilities or toolboxes that deal with the communication between the control board and the simulator. Hence, it comes the second limitation: the compatibility. The control board must have the communication channel supported by the simulator. This may require the development of a different interface to the original one to communicate the control board with the workstation. Then the simulated code is not the same as the final version. Also, efforts are required to manage the temporal synchronisation between the control board and the workstation. Without full synchronisation, no statements about temporal conformity are possible.

Traditionally, *Software-in-the-loop* (SIL) systems (Fig. 2.13c) have simulated the source code equivalent to the control algorithm and the plant on the same host (workstation). The main feature of this technique is that the whole simulated system is on the same verification tool. Thus, it has high visibility of the whole system, and it can access any simulator or design from the same process. In addition, it allows working with different levels of abstraction, adapting the code to the needs of the verification and providing flexibility to changes. Many works have benefited from this technique [CZCV13, TYH+19, RMdK19]. However, the workload of the simulator is so high that it is often a slower option than PIL/FIL or HIL. Authors such as [DGK07] have defined the main limitations of SIL. Some of them include synchronisation problems with the plant, the scalability of the auto-generated code and the poor reusability of the code.

Table 2.2 compares the main features of HIL, PIL/FIL and SIL. This figure includes concepts such as software/hardware debugging capability, source code and plant changes flexibility, numerical and temporal accuracy of the power plant simulations, simulation

speed, compatibility with different manufacturers and interfaces, cost of simulation infrastructure, and reusability of the code verified in the simulation phase.

Table 2.2: Comparison of non-destructive tools for verification in power electronics.

.	**HIL**	**PIL/FIL**	**SIL**
Tipe of verification	Full-control board	Specific SW/HW modules	Specific SW/HW modules
Price (cheap)	○ ○ ○	● ○ ○	● ● ●
Compatibility	● ○ ○	● ○ ○	● ● ●
Verification speed	● ● ● (real-time)	● ● ○	● ○ ○
Timing Accuracy (power plant)	● ○ ○	● ● ○	● ● ●
Flexibility to changes	● ● ○	● ○ ○	● ● ●
Debug capabilities	● ○ ○	● ● ○	● ● ●
Code re-usability	● ● ●	● ○ ○	● ○ ○

Except for HIL, all other techniques either do not verify the full system code or have to add extra functionality to enable verification, like communication support. In addition, universal translators are often used to generate code from simulation models. Therefore, the auto-generated code is not easily translatable to executable code on board.

The SIL technique is presented as the most advantageous option. However, in order to extend its use, its deficiencies need to be improved. These challenges focus on increasing the simulation speed and testing the whole system (software and hardware simultaneously) to offer a full re-usability.

Once the control system has been verified, the testing phase starts against the real plant. All parts of the control system are gradually checked for correct operation, i.e. sensors, safety devices, power plant elements, control board and control algorithm operation, both open and closed control loop, communication signals, etc., are verified. This phase usually takes a long time as there are many devices and functionalities to install, test and calibrate.

When the whole system has been verified, the final version of the source code is obtained. Then, the final operation is checked by running the power plant at the nominal design power. Any faults detected must be analysed and addressed through an in-depth simulation analysis.

2.3. QEMU as a software emulator

QEMU (Quick-EMUlator) is an *open-source* fast processor emulator. It is a virtualisation platform that allows cross-platform execution of software applications like bare-metal or operating systems. Initially, it was made using *Dynamic Binary Translation* (DBT), based on Just-In-Time compilation (JIT) compilation, to achieve fast emulation speed

[Bel05, QEM]. Dynamic binary translation enables running binary code generated for a target (or a guest) processor (e.g., ARM-Zynq Board) on another host processor, which has an entirely different instruction set (e.g., Intel i9). This means that it translates the binary code associated to the target CPU into the instructions of the host CPU. Then, it executes the translated instructions on the host CPU.

Since 2005, many features have been added to QEMU, such as peripheral support, support for different architectures, virtualisation and performance improvements. Nowadays, QEMU offers multiple *modes of operation*:

- User mode. It can run simple guest user-space processes in the host system, which can have a different instruction set architecture than the guest system. In user mode, QEMU emulates only the CPUs. It executes guest instructions, captures system calls, and sends them to the host kernel. Therefore, it is possible to run x86 binaries on ARM without emulating the full system.
- Full-system mode. In this mode, QEMU emulates a complete system, including peripherals. The QEMU's default translation engine is DBT or parallelized-DBT, called Multi-Threaded Tiny Code Generator (MTTCG).
- Virtualisation. KVM and Xen hosting. In virtualisation, QEMU sets up and supervises the KVM and Xen engines to emulate the guest code. In this case, QEMU deals with all virtualisation aspect that can not be emulated (peripherals or input/output management). The guest code execution is a task of the host virtualisation support. The virtualisation requires host hardware support to execute the guest code directly, so it only works if the guest has the same instruction set architecture as the host.

QEMU supports a wide range of target processors (like x86, ARM, MIPS, or RISC), PCI peripherals, serial ports, graphic cards, and more services. In 2012, Xilinx included QEMU in its PetaLinux tool suite to verify embedded operating systems. It currently supports emulation of MPSoCs Xilinx families as Zynq or Zynq UltraScale and Versal. However, even though these families have multi-core microprocessors, QEMU uses a *single host-core* for multi-CPU emulation due to the QEMU main loop limitations.

The contributions of this thesis have focused on enabling the use of QEMU with external simulators at the same time to build a co-simulation virtual platform. It has used parallelised engine of QEMU to support the fast-emulation of MPSoCs systems. The parallelised engine of QEMU is based on DBT, and therefore, it is crucial to understand how the DBT engine works.

Figure 2.14 shows an example of the translation process using DBT. In QEMU source code DBT is called Tiny Code Generator (TCG). In this case, we start from an ARM guest architecture and emulate it on an x86 host. The guest instruction to be translated and executed is `wfe` (wait for an event). In essence, the translation process consists of

two phases. First, it translates the guest code into an intermediate code (also called *micro-operations* [Bel05]). Then, it translates the instructions from the intermediate code to the host code. Finally, it executes the host instruction sequence.

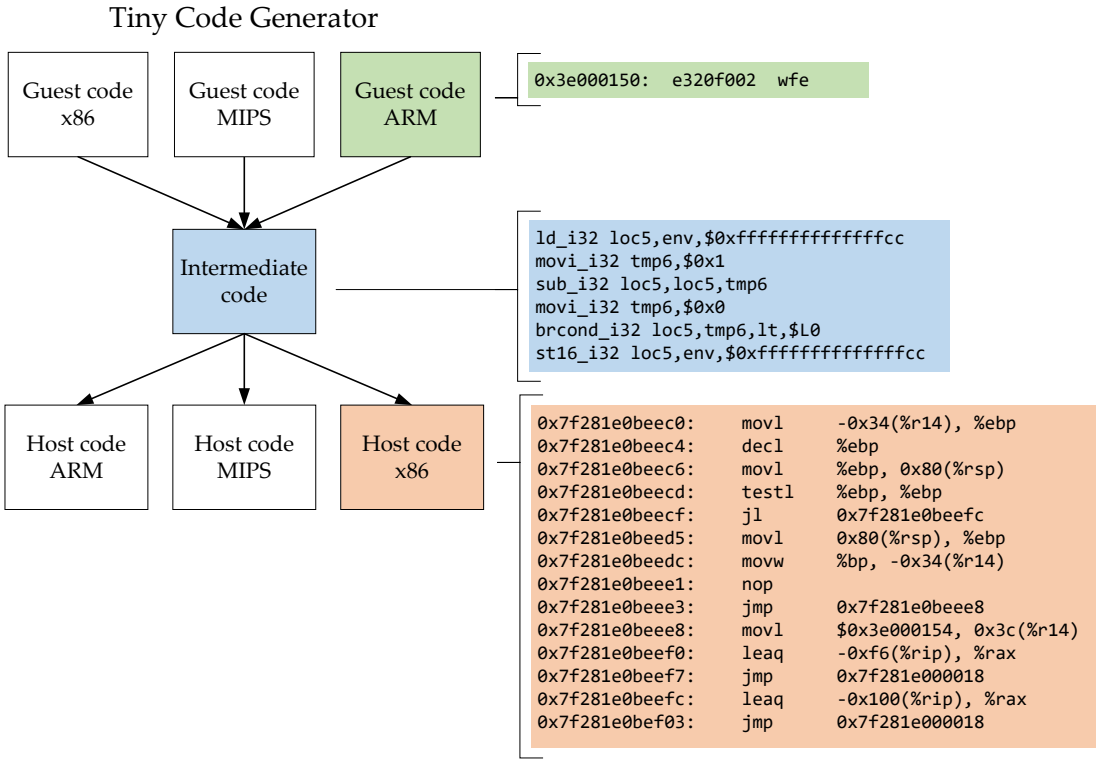

Figure 2.14: Ejemplo de traduccion de TBs.

In order to speed up the emulation, the guest CPU (also known as virtual CPU or vCPU) instructions are organised in basic *Translation Block (TB)*. A translation block is a guest instruction sequence that executes without branches, except at the end of the block. The end of the block is defined by a jump instruction (branch instructions or call sequences) or a breakpoint detected in the guest code. Two sections are added to the host TB in the translation process: the prologue and the epilogue. These sections are a mechanism to access in execution time or runtime to QEMU internals variables. One of these variables is the guest CPU instruction counter. Updating and reading this variable at execution time makes it possible to limit the QEMU execution with a maximum number of guest instructions executed.

An important aspect of QEMU is the use of *TB caches*, which considerably increases its performance. These caches store the translated TBs allowing their use at the execution phase without spent time in the translation phase. Also, the translated TBs can be linked through called *TB chains*. When executing a TB that links to another TB via a chain, both TBs are executed sequentially.

Figure 2.15 shows the simplified DBT loop of a guest CPU in QEMU. Each time a new TB is read, it is checked if its translation already exists in the cache. If it does not exist, it is translated. When the TB cache is full for simplicity sake, a flush is performed,

and all TBs are deleted. If the TB exists in the cache, it is checked if it can be linked to the previous TB. If so, a TB chain is created. The entire TB chain is then executed.

There are multiple conditions in which the process of executing the TB chain can be ended [Bel05]. The most usual ones are:

- When a self-modifying code guest event is detected. In this case, the TB is invalidated.
- When an asynchronous interrupt is detected in the guest. This event causes the TB chain to be broken. When the execution process is finished, it is checked if there is any pending interrupt.
- When an exception is detected in the guest. In such case, the whole TB is restored, i.e. the TB is re-translated in a mode that saves the state of the guest CPU before the exception.

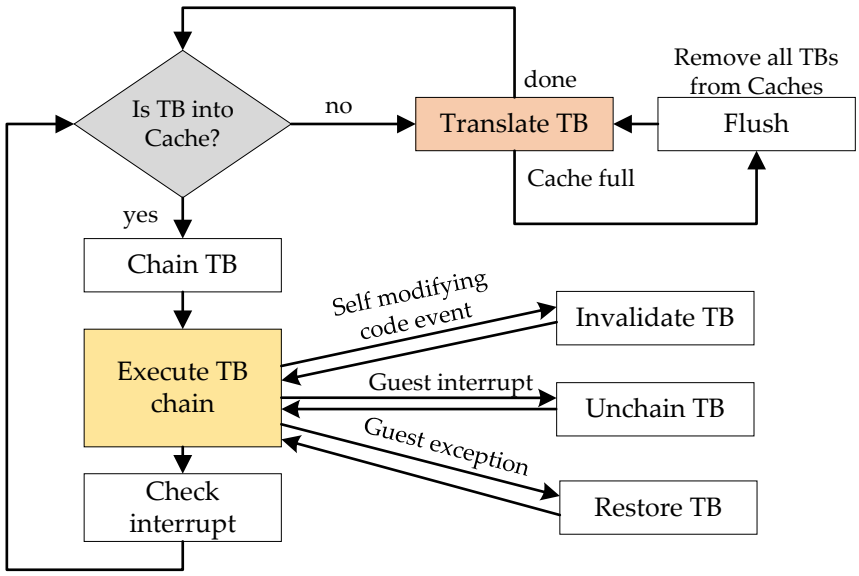

Figure 2.15: QEMU DBT flow.

The increase in the number of CPUs per chip has made obsolete the sequential emulation of QEMU. In response, different works have provided solutions to parallelise QEMU emulation. COREMU [WLC+11], and PQEMU [DCHC11] are the two most cited works in the literature. COREMU uses private code caches, creating a cache for each guest CPU. PQEMU's authors use both private and shared code caches. Although they show a speedup of around 4x, none of these papers addresses the problems of cross-ISA emulation. Some of these problems are the difference in memory models or the semantics of atomic operations. HQEMU [HHY+12] speeds up DBT by frequently translating guest code into optimised host code. This translation is performed in parallel with code execution. In Pico [CBBC17], the authors present the basics of what will be the official project

to parallelise QEMU's DBT. In this work, they studied the translations of atomic instructions across different instruction set architecture. They also present a new mechanism for managing TB chains in caches, which increases the efficiency of TB lookup in multi-core applications. This new TB reordering is based on the use of hash tables, named as *QEMU Hash Table (QHT)* by the authors.

2.3.1. Multi-Threaded Tiny Code Generator

In response to the current trend of increasing the number of CPUs or cores per device, a parallelized-DBT project was launched. The Multi-Threaded Tiny Code Generator (MTTCG) project was included in QEMU 2.9. It allows the translator of QEMU to run one host thread per guest CPU or virtual CPU or guest CPU (vCPU). Some platforms, such as PetaLinux, use this feature to speed up the OS emulation [QEM]. MTTCG incorporates some of the contributions presented in [CBBC17].

Currently, it is a work-in-progress project designed to provide multi-threading support for the emulation of multi-core systems. Some changes were introduced in the translator code to migrate from a single-thread solution (where vCPUs are executed sequentially in a single host thread) to a one thread per vCPU solution. These changes mainly affect the code dealing with the memory consistency, the execution of atomic instructions, the management of the blocks translated by each vCPU, and dirty page tracking [DBK+16, CBBC17]. Figure 2.16 shows the internal QEMU architecture using the MTTCG mode, in which there is a direct assignment of each vCPUs to each thread in the host CPU. In addition to the threads assigned to each vCPU, additional threads are also reserved for memory management and the execution of the QEMU main loop.

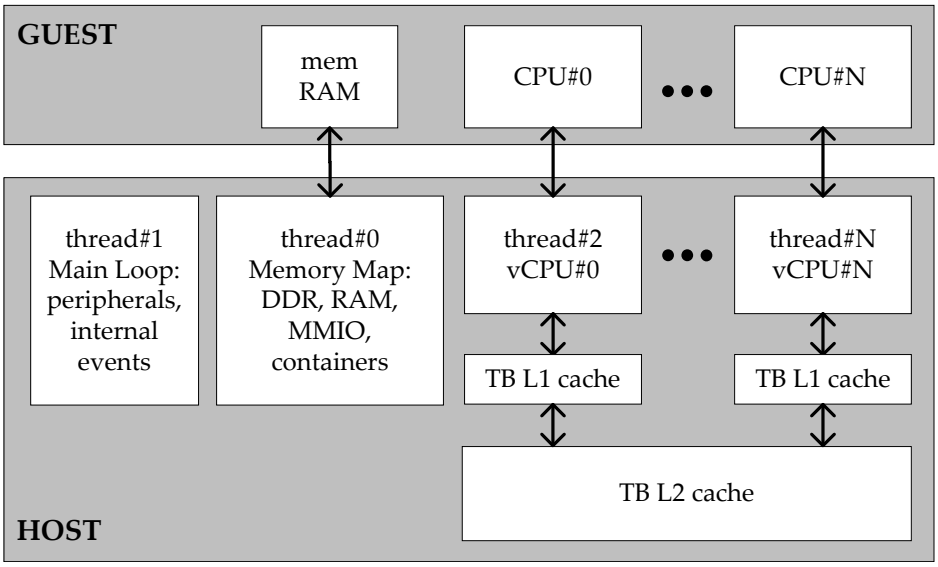

Figure 2.16: QEMU-MTTCG internal architecture.

QEMU-MTTCG achieves high performance through the parallelisation of the high emulation workload. In addition, to reduce the code translation operations, once a trans-

lation block has been translated for a given vCPU, this translation can be reused by other vCPUs in need to execute that piece of code. For this purpose, a cache hierarchy that centralises all translation blocks translated for each vCPU has been added [CBBC17].

Figure 2.17 shows the flow of the QEMU-MTTCG translation-execution loop. Similar to Figure 2.14, the guest code is packed into TBs. Then, each instruction is translated into its equivalent host code. Once the TB is translated, it is stored in the caches. Like a real multi-core processor, there are multiple caches. The level 1 cache is local to each vCPU, and the level 2 cache is shared between all vCPUs. In the same cache, links (also called chains) are created between the TBs [CBBC17]. These links allow TBs that have already been translated to be executed sequentially. For each of the vCPUs, a thread executes the TBs host, taking into account the links. Once each TB has been executed, the guest instruction counter is incremented. Remember that the guest instruction counter is an internal QEMU variable that is accessed in runtime. When the execution is finished, it checks if the next block of TBs has been translated. If so, it reads it directly from the cache. If not, the whole translation-execution process is repeated for the next TB of the guest.

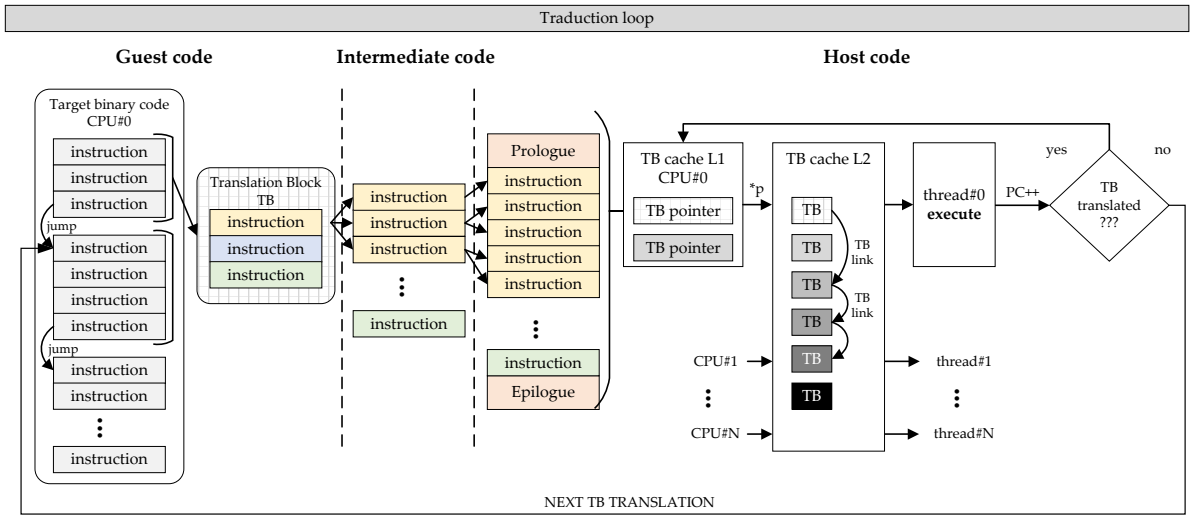

Figure 2.17: Ejemplo de traduccion de TBs.

Although this solution has allowed increasing the emulation performance on multi-core systems, the notion of time remains an unsolved task. When the multi-thread mode is selected, the QEMU instruction counter is automatically disabled. This thesis presents a solution to obtain the software notion of time from the time of each vCPU.

2.3.2. The notion of time in QEMU

QEMU was not developed to model the CPU timing. Instead, it is an instruction-accurate emulator made using a sequential execution. It is accepted that an instruction accuracy is enough to perform a functional verification for most MPSoC applications.

In [KYH16], the authors presented a good summary of the options for temporally modelling a processor. The options cited by the authors are: k-CPI simulation, analytical simulation, sampled simulation, statistical simulation, FPGA-accelerated simulation, hybrid simulation and control-sensitive simulation. Although timing is a field of great interest, few papers have offered solutions to obtain timing modelling of the CPUs in QEMU. In [CYT11] a fast cycle-accurate instruction set simulator based on QEMU, is presented. It models the processor pipeline to obtain cycle-accuracy. It uses a very basic approach to model the data hazard. This is valid for the considered scalar processor (ARM9). However, described techniques are difficult to extend to modern and more sophisticated processor. The work [KYH16] describes a QEMU-based simulator for modern superscalar out-of-order processors. It models the cache controller and the branch predictor to achieve a cycle-approximate accuracy. In [LHL+16] the authors propose to mix OVPSim with QEMU to get timing details of caches and Translation Lookaside Buffers (TLBs). Previous papers only provide specific solutions that cannot be extended to all the machines on the market. In addition, to obtain a cycle-accuracy, these solutions introduce an extra overhead that reduces the emulation speed.

In QEMU and other platforms that integrate QEMU, such as QBox or PetaLinux, this aspect has been simplified, using the number of guest instructions executed to measure the elapsed time. In [KYH16], this option is referred to as *k-CPI*. Therefore, QEMU has *instruction-time accuracy*. The number of instructions executed by QEMU is defined by an internal guest instruction counter that is updated when an entire translation block is executed on the host.

Let us consider the case of a processor with a single vCPU. The time advance can be obtained by Equation 2.1 where c_n is the number of cycles it takes to execute the n-th instruction (MOV, AND, SUB, ADD, etc.). T_{CLK} is the period of the CPU clock, and $k_{cache-n}$ is a factor that considers the influence of cache misses or branch misprediction for the in n-th instruction. N is the total number of executed instructions.

$$\triangle t_{CPU} = \sum_{n=1}^{N} c_n * T_{CLK} * k_{cache-n} \approx T_{CLK} * CPI * N \tag{2.1}$$

This thesis focuses on adding a synchronisation mechanism to QEMU-MTTCG to enable its use in fast hardware/software co-simulation virtual platforms. Since the plan is to use these virtual platforms to validate MPSoC design for application fields that require long simulation times, the simulation speed is a critical factor. Due to the limitations mentioned above of cycle-accurate models and the high simulation speed requirement, some simplifications have been applied to Equation 2.1. Firstly, the cache influence has not to be taken into account, therefore $k_{cache-n} = 1$. Also, the period of the CPU clock (T_{CLK}) is assumed to be constant. The number of cycles for n-th instruction (c_n) can be approximated by its average value (Cycles Per Instruction (CPI)). Although the CPI is

obtained for a specific benchmark, it can be considered a reliable reference to characterise the average performance of the processor. To use the CPI rate, we can assume the time advance is proportional to the number of executed instruction N, with $T_{CLK} * CPI$ being a *constant term*.

QEMU only emulates CPUs concurrently when it uses KVM or MTTCG. Only one thread is available in other setups to perform the translation sequentially using a round-robin approach. In this case, QEMU uses only a variable to count the total number of instructions executed by all vCPUs. In the MTTCG setup, since the vCPUs execute these instructions in parallel instead of sequentially, the total number of executed instructions are not proportional to the elapsed time, causing a timing error. Figure 2.18 depicts this timing error for an example of dual-vCPUs running in single-thread mode (Figure 2.18a), compared with a correct operating in multi-thread mode (Figure 2.18b). Each block represents the vCPU time elapsed for each iteration (A, A', B, B', ...), and the red numbers represent the order in which the blocks are used to increment the total software time. The arrows indicate which block is used to increment the software time at each instant. Please note that in the single-thread, a timing error appears because, for multi-vCPU emulation, the sum of each CPU time is not the total time of the software.

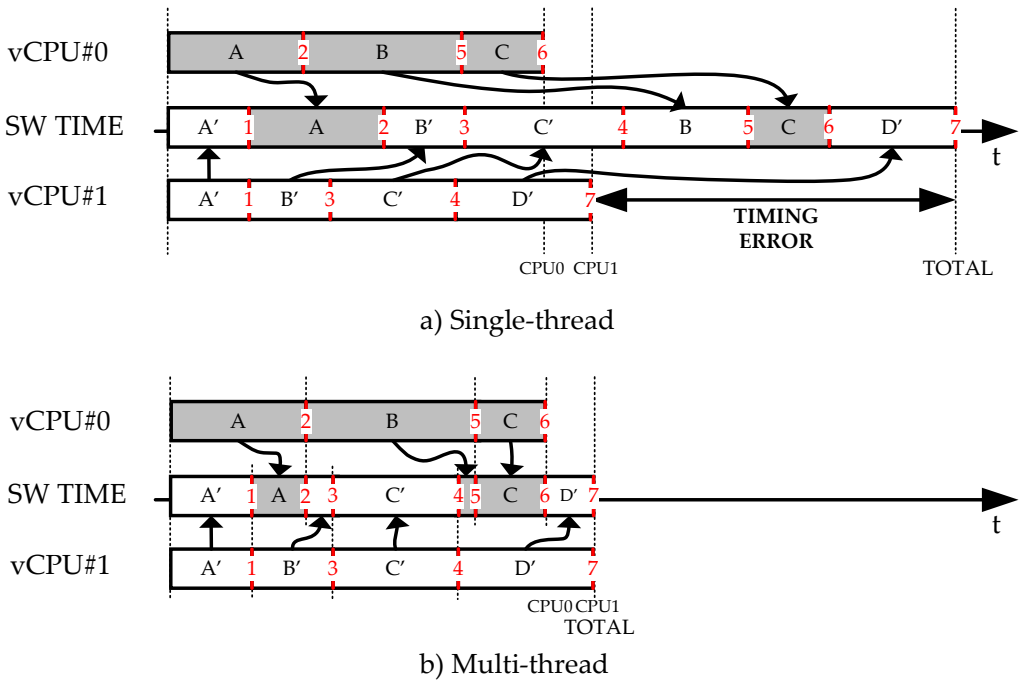

Figure 2.18: Comparison between timing flows for two vCPUs: (a) single-thread mode; (b) multi-thread mode.

Figure 2.18b shows the desired software timing flow when there are more than one vCPU. The software emulator time (TOTAL in Figure 2.18) must be obtained from the vCPU that has executed the highest number of instructions (in this case, vCPU#1).

2.4. SystemC and TLM support

SystemC is a trendy open-source C++ class library standard designed to describe systems and hardware designs. Developed by Open SystemC Initiative (OSCI) in 1999, approved as IEEE standard in 2005 and updated in 2011 [IEE11], it has become one of the most popular modelling languages in the electronic system-level design flow. The OSCI SystemC implementation has its event-driven simulator, also called the kernel. Therefore, it can be used as a DES. Its DES can model time and simulate concurrency and all aspects of hardware: modules, signal, ports and interfaces. Furthermore, it allows to model the hardware with a high abstraction level, and as a result, the simulation speed can be substantially enhanced.

Although SystemC is referred as a language in this book, strictly speaking, SystemC is not a language, it is a C++ library. The concept of language comes because the standard refers to it as a modelling language. However, the standard also clarifies that it is a C++ library.

The main advantage of SystemC is that it allows the use of a single language (C/C+) in multiple phases of design. Thus, SystemC allows verification of the same system using multiple levels of abstraction. Figure 2.19 shows a comparison of the usage of the most common languages in the area of electronic design. As it can be seen, the two languages that provide more flexibility are SystemC and SystemVerilog. SystemVerilog is used in hardware design and offers compatibility with Verilog. The main advantage of SystemC over SystemVerilog is that it allows working at a higher level of abstraction. Therefore, it is a more attractive language for software engineers. Furthermore, using the same language to verify software and hardware modules minimises code bugs in projects where the interaction between software and hardware modules is high.

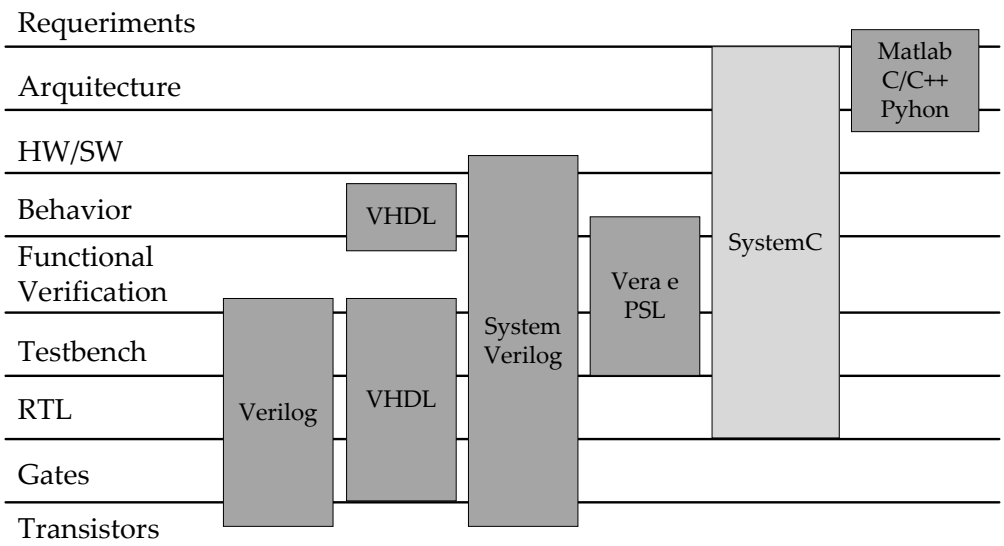

Figure 2.19: Use of language in Electronic Design Automation tools [BDBK10]

The SystemC kernel (DES), also called SystemC's schedule, divides the simulation into multiple stages. The kernel algorithm can be seen in Figure 2.20 and is divided into in three stages: *elaborate*, *simulate* and *cleanup*.

The *elaborate* step initialises the simulation resources as well as all the designs. The *simulate* step runs the SystemC's kernel. The SystemC kernel is based on an events discrete loop. Each event has one or several sensitive processes and a timestamp. This timestamp is rounded to the timing precision of the simulation. The events are ordered in a global event queue. When the next event is triggered, the global simulation time is updated with the event's timestamp, and the sensible processes are executed. If there are several events with the same timestamp, they are triggered as delta cycle events. When all events and delta cycle events scheduled are processed, the simulation ends [BDBK10]. Finally, the *cleanup* step releases the reserved memory and saves the traced signals.

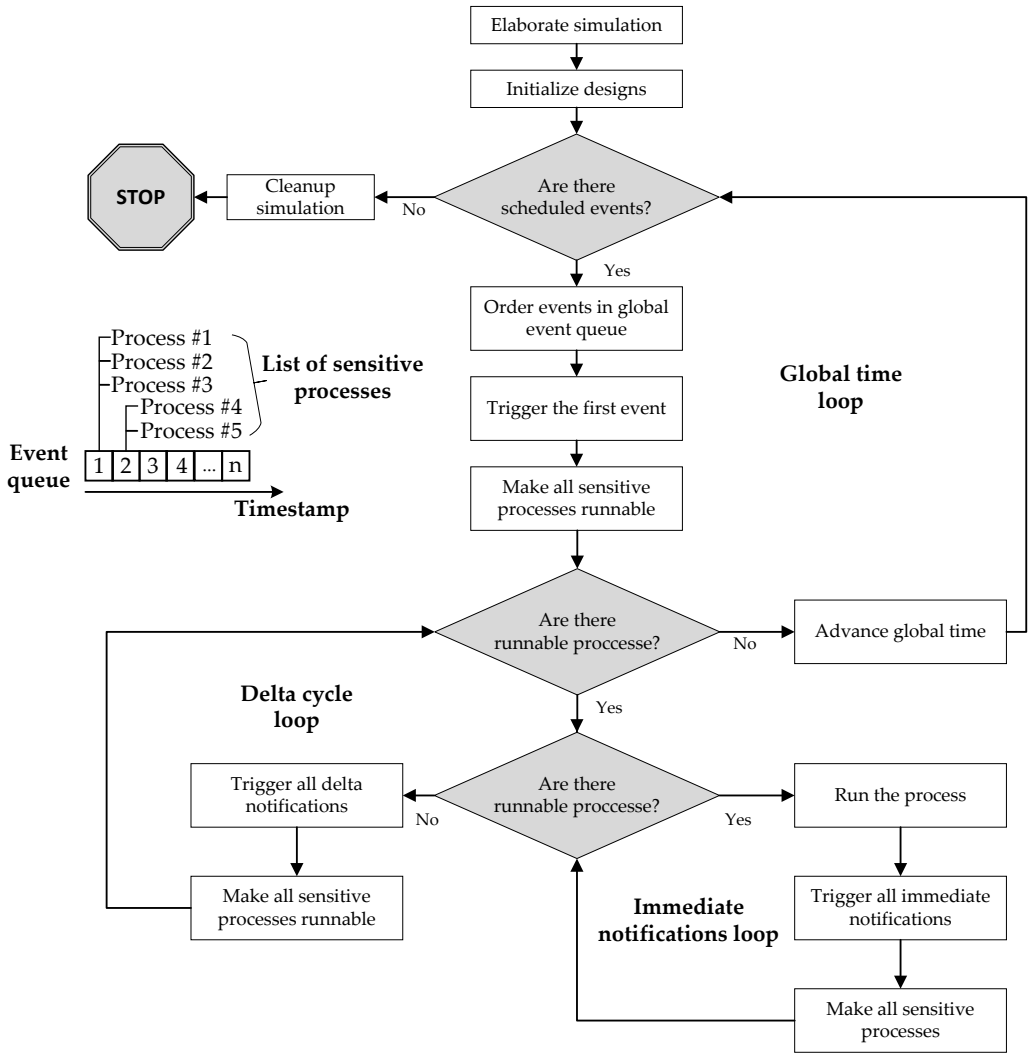

Figure 2.20: Simplified SystemC simulation kernel [BDBK10].

By default, SystemC executes its event loop sequentially. However, the standard does not limit its use to sequential execution. There are numerous obstacles to parallelising

SystemC [Dom16]. Nevertheless, much work has been done to make a Parallel-DES based on SystemC. In [SLPH10], the authors proposed a conservative synchronous simulation approach. In such case, the thread has to wait until other threads have completed the same simulation cycle to continue. They introduce time barriers that impose a limitation on performance. In [Moy13], the authors introduce more information in the event queue about the start and end time of the process to let the scheduler freely spread it across the time. In [DLS17, CSD20], the authors propose a guide to make a conservative asynchronous simulator, also called Out-of-Order Parallel Discrete Event Simulator (OoO PDE). In an OoO PDES, each module is assigned a thread that executes freely unless a causal relationship prohibits it. Since each module can own a local clock, the concept is a great candidate for asynchronous co-simulations. Although it has been shown an increment of simulation performance, the complexity of the proposed synchronisation algorithms is an obstacle to its inclusion in the OSCI SystemC implementation.

The communication mechanism between the SystemC modules is one of its essential features. SystemC relies on the TLM abstraction level to model communications. To avoid possible confusion in the reader with the TLM abstraction level, from now on, we will refer to the libraries that allow making a TLM communication in SystemC as *TLM-library*. TLM-library is an API inside SystemC used to model the communication between modules, exchanges messages as a single step rather than a sequence of cycle-level actions. Its implementation within SystemC has been standardised by OSCI SystemC TLM-2.0 [Ayn09]. It uses direct function calls, like C++, for sending transactions between modules in a structured form. Thus, it aims to reduce events number in the RTL communications (at Register Transfer Level of abstraction), reducing communication overhead between the modules.

The SystemC standard also defines two coding styles using TLM-2.0. It summarises them into two types: loosely-timed coding style and approximately-timed coding style. However, it does not limit operation to these two modes.

The implementation of *Loosely-timed mode (LT)* models is based on Blocking Transactions (BT). Therefore, *communications are blocking*, what mean they block the execution of Master modules (Initiator) until the Slave (Target) responds. From a timing point of view, blocking transactions allow setting the beginning, and the end of each transaction [IEE11]. In the blocking transactions mode, the synchronisation between the modules of the simulation is based on *Temporal Decoupling* to synchronise the modules through a global time slice limit called *Quantum*. Figure 2.21 show an example of temporal decoupling using Quantum time. When any execution of a module exceeds this time limit, it waits for the rest of the modules to continue with the simulation. The temporal decoupling mechanism improves simulation performance by not synchronising all modules in every clock cycles but only in multiples of Quantum. For high HW/SW interaction rates, it requires having a Quantum time of at least the shortest time between two HW/SW

interactions. Small values of the Quantum generate many synchronisations and, therefore, slow down the simulation to exchange for more temporal precision. Otherwise, large Quantum values increase the speed of the simulation, but temporal precision is lost in the simulation [IEE11]. Choosing the correct Quantum value is the key to avoid inconsistencies in the simulation. Works such as [DBK+16, Xil19] have used this type of solution to link software simulators with hardware. However, the authors advise that the value of the Quantum time depends on the type of application, and therefore there is no standardised method to optimise its value.

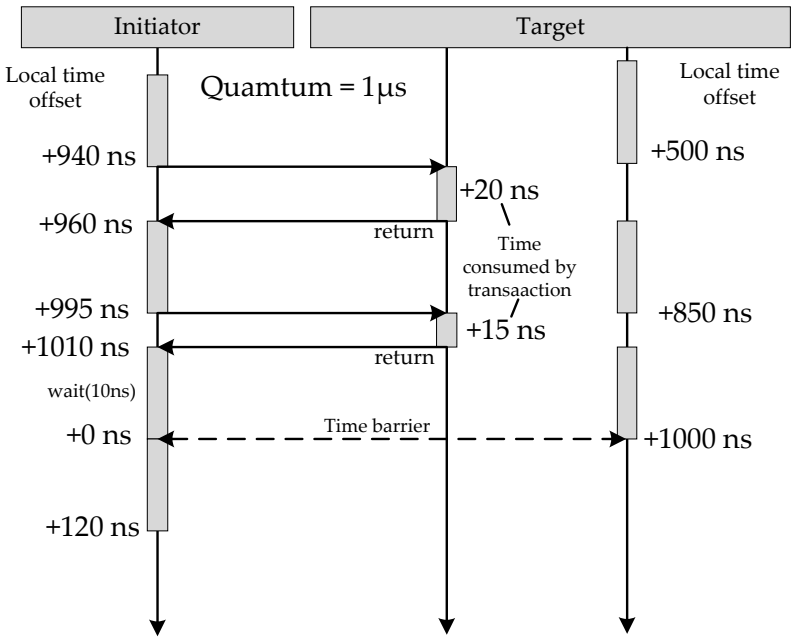

Figure 2.21: Example of temporal decoupling with quantum.

The other coding style defined by the standard is *Approximately-timed mode (AT)*. This type of communication is more elaborated as it includes more timing details and has a *non-blocking transport* interface. Therefore, its use is appropriate for architectural exploration and performance analysis. AT provides *timing points* and *multiple phases* for each transaction. Its main advantage is that it supports the integration of TLM models with cycle-accurate models. However, it reduces the performance of the simulation by incorporating more transactions in the communication process. An advantage is that having more phases in the communication allows incorporating extra functionality. This is the case in [Del17a], where the author relies on AT to develop his implementations of non-memory-mapped protocols based on TLM.

Chapter 3

QEMU External Synchronization Mechanism

Emulation increases the verification-cycle speed, but to verify software and hardware co-designs, it is essential to synchronise the emulator with the external world.

3.1. Introduction

QEMU is one of the most widely used emulators tools for research purposes and the industrial field. However, QEMU support emulation of more architectures. Additionally, from the two previously mentioned options, only QEMU is able to emulate the Xilinx Zynq, UltraScale+ and Versal families, which integrate MPSoC and an FPGA fabric in the same die.

Software code emulation allows verifying the same binary code that it will run on the real platform. The need to emulate multicore embedded devices has motivated some projects to advance in the parallelization of the translator engine of QEMU (Dynamic Binary Translator - DBT). Consequently, a new parallel translation engine called Multi-Thread Tiny Code Generator (MTTCG) has been developed by QEMU contributors.

However, in this new parallel mode, the synchronization of QEMU with an external application (i.e., the hardware simulator) has not been resolved. This synchronization is fundamental in order to include QEMU in a hardware/software co-simulation environment. It should be noted that this tool is highly valuable for the verification of mixed hardware/software designs implemented in heterogeneous embedded devices such as MPSoCs + FPGAs.

The following chapter explains how to enable a synchronization mechanism between QEMU and an external hardware simulator using the QEMU parallel emulation mode (MTTCG). Then, it presents a contribution focus on adding QEMU to the co-simulation virtual platform as a multi-core software emulator.

In particular, the following proposals are presented:

- A procedure to obtain the number of instructions executed by each processor in a multi-core emulation. The notion of time for each virtual CPU can be calculated from this instruction counter, providing an approximately timing processor model.
- A method to break the main loop of QEMU and execute a synchronisation point with the hardware simulator. This is critical as not to affect the main loop performance and achieve fast emulations.
- A method to manage the software timing notion from a hardware simulator point of view in multi-core systems. The synchronisation between the external hardware simulator and the software emulator (QEMU) is essential to get a correct co-simulation.

The version of QEMU used as a starting point has been obtained from the Xilinx QEMU repository [Xil18]. Xilinx has patched the official QEMU version 2.11.1 (2018) to support Zynq-7000 and UltraScale emulation. Throughout this section, some simplified examples of the source code are shown. However, the complete patched code has been omitted for sake of clarity.

3.2. QEMU machine linking

The monitoring of accesses between software and hardware and interrupts that come from the hardware is necessary to ensure synchronisation between QEMU and the external hardware simulator.

In QEMU nomenclature, the machine represents the CPU chosen to emulate or guest CPU or virtual CPU - vCPU. This CPU can be composed of multiple cores, and each machine has its architecture defined in the internal QEMU source files.

The hardware accesses from software are detected by monitoring memory accesses. The user must map the hardware memory range in the QEMU machine source files to monitor them. Also, it is necessary to set up the virtual interrupts manager of the machine to capture any interrupt from the hardware.

An example applied to Zynq-7000 is shown below. In this example, the 1GB memory region reserved for access to the FPGA is mapped. Besides, the pointers to the variables that model the interrupt signal are obtained.

Listing 3.1: Example of MMIO mapping and interrupts linking. Zynq-7000 machine. file://hw/arm/arm_generic_fdt.c

```
1 #define BASE_ADDR 0x40000000ULL
2 #define SIZE_ADDR 0x7FFFFFFFULL
3 #define NUM_IRQS  16
4
```

```
5  void COSIL_interactions_mmap(CPUState *cpu, void *fdt, unsigned long long base_addr, unsigned
       long long size_addr, unsigned long long num_irqs){
6    FDTMachineInfo *fdti;
7    qemu_irq* irqs;
8    // Get interrupts (IRQs) from the Flattened Device Tree
9    fdti = fdt_generic_create_machine(fdt, NULL);
10   irqs = zynq_get_irqs(fdti);
11   fdt_init_destroy_fdti(fdti);
12   //Set MMIO and IRQs
13   COSIL_mmio_map(cpu->env_ptr, base_addr, size_addr);
14   COSIL_irq_map(cpu->env_ptr, irqs, num_irqs);
15   //Register MMIO callbacks
16   COSIL_register_rams();
17 }
18
19 static void arm_generic_fdt_7000_init(MachineState *machine){
20   //Machine definition
21   //...
22   arm_generic_fdt_init(machine);
23   //...
24   //Get and set MMIO and IRQs
25   COSIL_interactions_mmap(cpu, fdt, BASE_ADDR, SIZE_ADDR, NUM_IRQS)
26 }
27
28 static void arm_generic_fdt_7000_machine_init(MachineClass *mc){
29   mc->desc = "ARM device tree driven machine model for the Zynq-7000";
30   mc->init = arm_generic_fdt_7000_init;
31   mc->ignore_memory_transaction_failures = true;
32   mc->max_cpus = 2;
33   mc->default_cpus = 2;
34 }
35
36 DEFINE_MACHINE(ZYNQ7000_MACHINE_NAME, arm_generic_fdt_7000_machine_init)
```

3.3. Hardware/Software interactions

In order to use QEMU as a software emulator in a co-simulation environment, a synchronization is required to enable it running jointly with a hardware simulator or other external modules.

Hardware simulator and software emulator can be seen as modules that exchange messages in the simulation kernel. The messages can be defined based on their nature, modelling physical or virtual interactions. Interaction represents the event in the simulation where both modules have to exchange some information. On the one hand, the *physical interactions* are those that come from real memory accesses between the hardware and software or asynchronous interrupts generated from the hardware to the software. On the other hand, *virtual interactions* send timing and status information between modules for synchronisation or debugging purposes. Virtual interactions share the same concept as *null messages* in literature [Fuj01].

It is essential to define an interface to communicate and synchronise the hardware

simulator and the software emulator whenever there is some hardware/software communication. In every interaction, the state of each module and its time advance must be shared between them. These interactions can be classified into three types:

- Synchronisation points (virtual interactions).
- Input/Output (I/O) accesses between the software emulator (processor) and hardware simulator (physical interactions).
- Asynchronous interrupts (physical interactions) to the processor.

The above interactions are modelled by function calls. This function calls link internal QEMU functions to functions declared in the external simulator. The links are made by dynamic linking. The function calls that have been developed in this thesis can be divided into two types:

- Function calls from QEMU to the hardware simulator (Synchronisation points and IO accesses). These callbacks come from QEMU and execute a function in the hardware simulator. Thus, an interaction pauses QEMU execution and allows an external simulator to obtain state and timing information from QEMU and advance coherently. These functions pause the QEMU execution until they return.
- Function calls from the hardware simulator to QEMU (asynchronous interrupts). These callbacks come from the external hardware simulator. In the particular case of interrupts, QEMU is notified each time some interrupt signal changes of value.

Figure 3.1 shows the different types of callbacks. Also, it can see how they are linked to the outside of QEMU through dynamic linking. In the following paragraphs, each of the modules involved in the interactions is detailed.

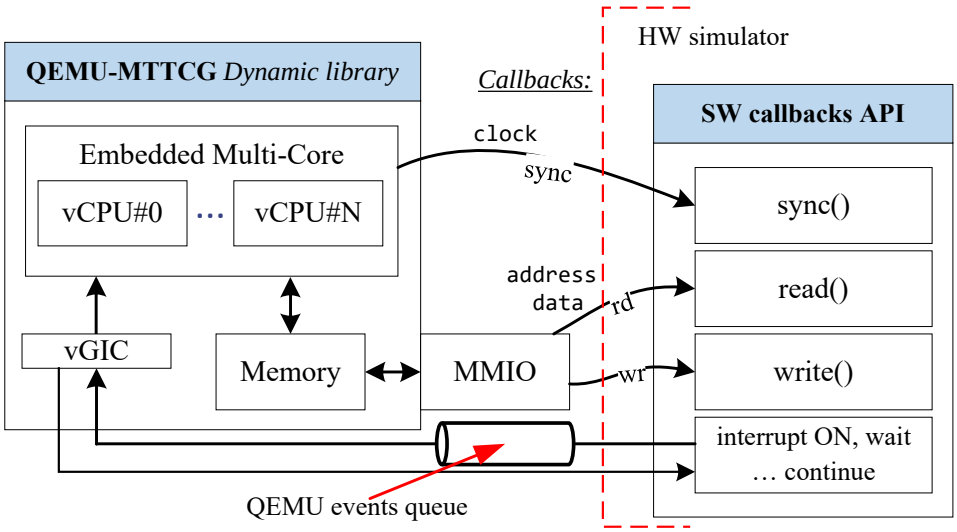

Figure 3.1: Interactions management by callbacks and dynamic linking.

The synchronisation points represent the instants in which the hardware/software modules are synchronised to prevent one simulator/emulator from lagging behind another. They become essential when there are no input/output accesses or interrupts. Unlike input/output accesses and interrupts, this type of interaction does not represent a real physical interaction but is an internal mechanism to synchronise both simulators/emulators. Synchronisation points are the most frequently generated interactions; therefore, they should be as fast and light as possible. They also become a mechanism to control the timing accuracy of the software emulator. The *sync()* callbacks are executed in the hardware simulator and called from the translation execution loops of each vCPU (from QEMU). The *sync()* function reads timing information. In essence, it reads the vCPU local time sent from QEMU.

The monitoring of I/O accesses is based on using the QEMU QEMU Memory-mapped Input/output Object (MMIO) object. The MMIO object has its methods that reproduce the access to the assigned memory range. The MMIO methods receive arguments providing access/transaction attributes (address, size, data, and time). Then, it is notified the hardware simulator about these events. The notification calls the functions *write()/read()* which are executed in the external simulator. Here all transaction arguments are read, and access to the hardware module is performed. The registration method of an MMIO consists of three phases:

1. Type Instantiation: registration of one or more MMIOs (*type_init()* Line:56 and *type_register_static()* Line:53).
2. Declaration of object class: instantiation of the functions that build the MMIO (*.class_init* Line:45).
3. Definition of object class: execution of the functions that build the MMIO (*COSIL_memory_init()* Line:23).

In Listing 3.2 is shown a simplified example of the source code used to instantiate an MMIO. Here, it shows the MMIO methods that are automatically called (*COSIL_read()* and *COSIL_write()*) when a I/O memory access are done by the processor to the hardware simulator.

Listing 3.2: Example of MMIO definition for Zynq-7000 machine. file://hw/arm/cosil_mmio.c

```
1  //-----------------------CALLBACKS-------------------------------
2  static inline uint64_t COSIL_read(void *opaque, hwaddr addr, unsigned int len){
3    //Get address and clock
4    //Do bus access - read()
5    //return read value
6    return rd;
7  }
8  static inline void COSIL_write(void *opaque, hwaddr addr, uint64_t value, unsigned int len){
9    //Get address and clock
10   //Do bus access - write()
```

```
11    return;
12  }
13  //-----------------------DECLARATION CALLBACKS------------------------------
14  static const MemoryRegionOps COSIL_mem_ops[1] = {
15    {
16      .read = COSIL_read,
17      .write = COSIL_write,
18      .endianness = DEVICE_NATIVE_ENDIAN
19    },
20  };
21  //---------------------------------INIT MEMORY------------------------------
22  struct COSILMemory *main_COSILdev = NULL;
23  static int COSIL_memory_init(SysBusDevice *dev){
24    struct COSILMemory *const s = FROM_SYSBUS(typeof(*s), dev);
25    //Definition of interrupts (IRQ)
26    //...
27    //Definition  MMIO
28    s->info.name = "COSIL_memory";
29    // Set MMIO region
30    memory_region_init_io(&s->info.iomem, OBJECT(dev), COSIL_mem_ops, &s->info, s->info.name, s
        ->info.size);
31    // Add MemoryRegion to sysbus
32    sysbus_init_mmio(dev, &s->info.iomem);
33    // Save as gobal variable the main COSIL dev
34    main_COSILdev = s;
35    return 0;
36  }
37  //-----------------------------ObjectClass constructor------------------------------
38  static void COSIL_memory_class_init(ObjectClass *oc, void *data){
39    DeviceClass *const dc = DEVICE_CLASS(oc);
40    SysBusDeviceClass *const sbdc = SYS_BUS_DEVICE_CLASS(oc);// Main system bus
41    // @init: Callback function invoked when the #DeviceState.realized property is changed to %
        true.
42    sbdc->init = COSIL_memory_init;
43    }
44  //---------------------------------TypeInfo----------------------------------------
45  static const TypeInfo COSIL_info = {
46    .name = TYPE_COSIL_MEMORY, //User Type. It include a copy of its parents ObjectClass
47    .parent = TYPE_SYS_BUS_DEVICE, //Type from wich this Type derives from.
48    .instance_size = sizeof(COSILMemory), //Size of the Object instance.
49    .class_init = COSIL_memory_class_init, //ObjectClass constuctor hook.
50  };
51  //----------------------------------TypeImpl register--------------------------------
52  static void COSIL_register_types(void){
53    type_register_static(&COSIL_info);//TypeImpl
54  }
55  //---------------------------------TypeInfo register-------------------------------
56  type_init(COSIL_register_types)
```

QEMU uses event-driven architecture, which means that it processes the emulated devices by means of events. These events are ordered in an *event queue* at runtime. Usually, events are added to the event queue when some description file such as sockets, pipes and other resources is ready to read or write. Another possible source of events is any expired internal timer.

In the case of asynchronous interrupts, QEMU does not check if a hardware interrupt

is pending for every translation block. When QEMU is notified that a new interrupt is triggered, an event is generated and added to the QEMU events queue. This mechanism can be seen in Figure 3.1 and is described in Listing 3.3.

The procedure for binding interrupts to the machine, notifying the QEMU evens queue of an interrupt event, and updating the status of interrupts in the virtual Generic Interrupt Controller (vGIC) is shown below. The registration method of an interrupt in QEMU source code consists of three phases:

1. Setting up an immediate timer. The immediate timer is called bottom-halves (BH) according to the QEMU nomenclature (*qemu_bh_new()* Line:53). This timer allows adding an event to the QEMU event queue, avoiding overflowing the call stack. As an argument, the handler that will be called when the timer ends is given (*update_irq()* Line:12).

2. Definition of the handler that is executed when the interrupt event is attended (*update_irq()* Line:53). When the interrupt event is attended, the *update_irq()* function notifies the virtual GIC of the interrupt signals that changed in level. The current clock of the running vCPU is then obtained, and a callback is sent with timing data to the hardware simulator. Thus, it is notified in hardware simulator that the interrupt has been attended (*COSIL_irq_done_cb()* Line:36).

3. Definition of the callback function to be executed from the hardware simulator (*COSIL_irq_done_cb()* Line:2). This function schedules a bottom-halves or immediate timer in the QEMU event queue. Therefore, an event is created that will be attended as soon as possible.

Listing 3.3: Example of IRQ linking for Zynq-7000 machine. file://hw/arm/cosil_mmio.c

```
1  //------------------------CALLBACK FROM HARDWARE---------------------------------
2  static void COSIL_write_irq(struct COSIL_irq *qirq){
3    assert(main_COSILdev);
4    //Asign interrupt value to interrupt triggered
5    main_COSILdev->pending_irq[qirq->addr / 4] = qirq->data;
6    //Schedule a bottom half for IRQ in QEMU event queue
7    //Scheduling a bottom half interrupts the main loop and causes the
8    //execution of the callback that was passed to qemu_bh_new.
9    qemu_bh_schedule_COSIL_irq(main_COSILdev->irq_bh);
10 }
11 //----------INTERRUPT HANDLER CALLED WHEN AN INTERRUPT EVENT IS ATTENDED----------
12 static void update_irq(void *opaque){
13   uint64_t clk = 0;
14   struct TLMMemory *s = opaque;
15   int i;
16   // Update IRQ level
17   for (i = 0; i < s->nr_irq; i++) {
18     int regnr = i / 32;
19     int bitnr = i & 0x1f;
20     uint32_t data;
21     int level;
```

```
22
23      data = s->pending_irq[regnr];
24      level = data & (1 << bitnr);
25      //file: /include/hw/irq.h - Generic IRQ/GPIO pin infrastructure.
26      qemu_set_irq(s->cpu_irq[i], level);//--> call to vGIC of the machine
27    }
28    //Get current Clock
29    if(use_icount){
30      clk = qemu_clock_get_ns(QEMU_CLOCK_VIRTUAL);
31    }else if (use_multi_icount){
32      clk = edl_cpu_get_multi_icount(last_cpu);
33    }
34    last_sync_icount = clk;// global variable
35    // Notify to hardware simulator that interrupt event was attended and when
36    COSIL_irq_done_cb(COSIL_opaque, clk);
37  }
38
39  //---------------------------------INIT INTERRUPT------------------------------
40  static int COSIL_memory_init(SysBusDevice *dev){
41    struct TLMMemory *const s = FROM_SYSBUS(typeof(*s), dev);
42    //
43    //...
44    //
45    // Set IRQS source from sysbus
46    if (s->nr_irq) { // from machine definition
47      for (i = 0; i < s->nr_irq; i++) {
48        // Request an IRQ source.  The actual IRQ object may be populated later.
49        sysbus_init_irq(dev, &s->cpu_irq[i]);
50      }
51    }
52    // Set IRQ handler for IRQ event
53    s->irq_bh = qemu_bh_new(update_irq, s);
54    //
55    //...
56    //
57    // Save as gobal variable the main COSIL device
58    main_COSILdev = s;
59    return 0;
60  }
```

After the interrupt event is processed, the interrupt is sent to the virtual Generic Interrupt Controller (vGIC). Once this happens, the vCPU is notified of an interrupt exception [QEM]. Nevertheless, there is a delay between the moment when the hardware triggers an interrupt and the moment when the vCPU is notified. This is caused by the large number of events that QEMU must schedule and handle. To solve this issue and avoid timing error, the solution proposed in this thesis blocks the hardware simulator until the QEMU scheduler attends to the interrupt and the vGIC is notified. Figure 3.2 depicts the differences of interrupt management for the real platform versus the QEMU case (Virtual board case), when an Interrupt ReQuest (IRQ) is triggered.

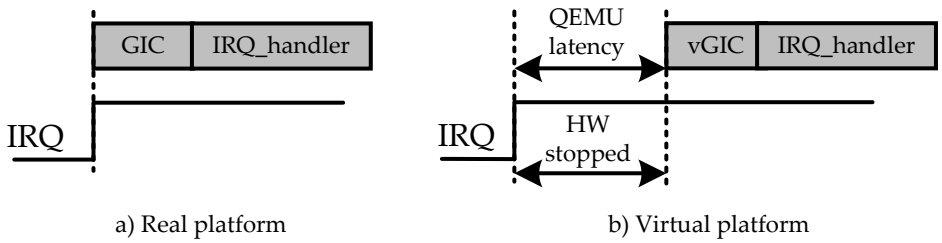

Figure 3.2: Interrupt management: (a) real platform vs. (b) virtual platform cases.

3.4. Implementation of External Synchronization in QEMU using Multi-Thread Tiny Code Generator.

Synchronization points allow the software emulator to send time and status messages to the hardware simulator to maintain successful synchronization in the virtual platform. This section explains the improvements made in QEMU-MTTCG during this thesis in order to introduce the synchronization points.

To carry out this proposal, it is necessary to define three aspects. The first one defines the required changes to obtain the state and timing data of each vCPU. That data must be sent to the hardware simulator. Once the approximate temporal notion can be obtained in MTTCG mode, the next step is to define the method to manage the synchronisation points. Remember that the synchronisation points are virtual interactions and should be as light and fast as possible. The last one explains the place in the QEMU source code where the synchronisation points can be inserted safely. Each one of these three aspects is described below.

3.4.1. Guest instruction counter

As discussed in Section 2.3.2, QEMU uses the k-CPI method to obtain its approximate timing notion. It means it is based on obtaining a counter of the executed guest instructions. This mechanism updates an instruction counter per each vCPU in the execution phase of the Translation Blocks (TB).

To obtain the guest instruction counter, QEMU includes a class called *CPUState* for each vCPU. This structure contains information about the state of the vCPU and the maximum number of instructions that the vCPU can execute sequentially. The maximum number of instructions limits the vCPU advance, allowing QEMU to attend to other events in its event queue. In the QEMU source code, this maximum number of instructions is stored as the *icountMax* variable. *icountMax* is also called by QEMU developers as icount budget.

During the translation phase, *intermediate code* (see Figure 2.14) are added at the beginning and the end of each host TB. These codes made up what it is the prologue

(P) and the epilogue (E) of the TB (see Figure 3.3). The prologue (P) accesses the *icountMax* variable of each vCPU, located in its *CPUState* class and subtracts the size of the TB in guest instructions units. To know the size of the TB, it is necessary to finish the translation phase and runs the epilogue. The epilogue updates the prologue variable setting the TB size.

During the execution phase, the prologue already containing the TB size information. Thus, it can check before executing any TB whether the subtraction of *icountMax* less *TB size* is 0 or less than 0. In this case, it does not execute the TB, it returns to the main loop and rebuilds a new TB with the number of instructions that allows *icountMax* to be 0.

The *difference* between the number of instructions executed before and after the execution phase is what makes it possible to obtain a counter of guest instructions from the QEMU.

In the single thread case, this instruction counter is unique for all vCPUs. Its value is the sum of the guest instructions executed by all the vCPUs. However, as seen in Section 2.3.2 this is not compatible with MTTCG mode.

To obtain the number of instructions executed by the vCPU in MTTCG mode, the proposal presented in this thesis is based on the method applied in the single-thread case, allowing compatibility with the original case. Figure 3.3 shows the mechanism to get a guest instruction counter.

In the start phase of QEMU, a thread is created for each vCPU. This generates an instruction counter in the CPUState class for each vCPU. As the CPUState pointer of each vCPU can be accessed from the prologue, each vCPU updates its instruction counter independently.

Listing 3.4 describes the intermediate code included in the prologue and epilogue of each TB. In essence, it is enabled the access to guest instruction counter of each vCPU from host code. Then, it is possible to obtain how many guest instructions have been executed at runtime for each guest TB.

Listing 3.4: Getting instruction counter for each vCPU in prologue and epilogue. file://include/exec/gen-icount.h.

```
1   static int icount_start_insn_idx; // global
2   //-------------------------------------PROLOGUE-------------------------------------
3   static inline void gen_tb_start(TranslationBlock *tb){
4     TCGv_i32 count, imm;
5
6     tcg_ctx->exitreq_label = gen_new_label();
7     // Init new local variable count
8     if (tb_cflags(tb) & CF_USE_ICOUNT || use_multi_icount) {// also enable fot MTTCG
9       count = tcg_temp_local_new_i32();
10    } else {
11      count = tcg_temp_new_i32();
12    }
```

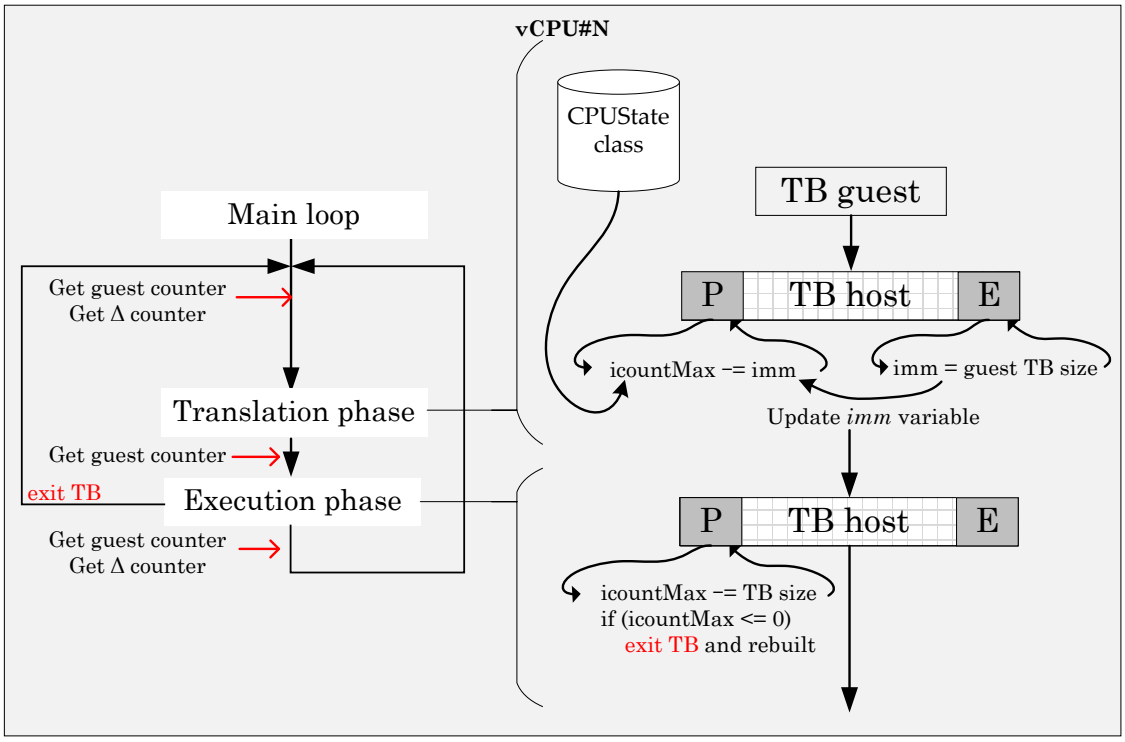

Figure 3.3: Block diagram to define how to get instruction counter in the TB host using prologue and epilogue.

```
13
14      // Load counter for each vCPU from CPUState Class
15      tcg_gen_ld_i32(count, cpu_env,-ENV_OFFSET + offsetof(CPUState, icountMax.u32));
16
17      if (tb_cflags(tb) & CF_USE_ICOUNT || use_multi_icount) { // also enable fot MTTCG
18        imm = tcg_temp_new_i32();
19        // We emit a movi instruction with a dummy immediate argument. Keep the instrution index
20        // of the movi so that we later (when we know the actual insn count)
21        // can update the immediate argument with the actual instruction count.
22        icount_start_insn_idx = tcg_op_buf_count();
23        tcg_gen_movi_i32(imm, 0x00000000);
24        //Sub number of instuction of TB
25        tcg_gen_sub_i32(count, count, imm);
26        tcg_temp_free_i32(imm);
27      }
28
29      //if count is less than 0, goto: exitreq_label: END of TB chains
30      tcg_gen_brcondi_i32(TCG_COND_LT, count, 0, tcg_ctx->exitreq_label);
31
32      // Store counter for each vCPU to CPUState Class
33      if (tb_cflags(tb) & CF_USE_ICOUNT || use_multi_icount) {
34        tcg_gen_st16_i32(count, cpu_env,-ENV_OFFSET + offsetof(CPUState, icountMax.u16.low));
35      }
36      //Free mem
37      tcg_temp_free_i32(count);
38    }
39    //-------------------------------------EPILOGUE-------------------------------------
40    static inline void gen_tb_end(TranslationBlock *tb, int num_insns){
41      if (tb_cflags(tb) & CF_USE_ICOUNT || use_multi_icount) {
42        //Update the num_insn immediate parameter now that we know the actual insn count.
43        tcg_set_insn_param(icount_start_insn_idx, 1, num_insns);
```

```
44        }
45        //
46        //...
47    }
```

Once it has an instruction counter for each vCPU, it is possible to obtain the increment of the number of instructions executed. This increment is equivalent to the variable N in the Equation 2.1. Thus, a temporal notion of each vCPU in MTTCG mode is provided.

Whenever QEMU executes a guest load/store instruction, an I/O access to a set MMIO region may be generated. QEMU can not know when this will happen as it depends on the guest code. When QEMU detects an I/O access in the translation phase, it stops the current TB translation. It then restores the number of un-translated TB instructions in the instruction budget, re-translates the TB again and exits to the main loop. Then, this ensures that the instruction which generates an I/O must be at the end of the TB. Therefore, in the execution phase, the main loop recovers control allowing QEMU to run any I/O event [QEM].

3.4.2. Management of Synchronization Points

Given that each vCPU has its own instruction counter, the simulated time [Fuj01] of each vCPU (local simulated time) can be obtained from Equation 2.1. The main factors that determine the instruction execution speed of each vCPU are as follows:

- Instructions shared between vCPUs. The synchronization in parallel execution of vCPUs is guaranteed by QEMU-MTTCG when an instruction affects multiple vCPUs. As with parallel conservative simulation, the affected vCPUs are blocked until all vCPUs reach the same simulation time. Otherwise, and for most of the instructions, the instructions of each vCPU are executed as fast as possible [CBBC17].
- The intrinsic complexity of the guest code is decisive in the execution speed of each vCPU. A complex structure of the guest code means that QEMU will make more translations and changes to the TB cache table. Therefore fewer executions will be chained, decreasing the performance of the vCPU simulation.
- The workload of the machine/OS on which QEMU runs. Therefore, the speed at which QEMU executes its instructions can change in function on the host OS workload.

The above features produce time differences between the vCPUs. To guarantee that a vCPU with less executed instruction does not update the notion of the time of the software emulator (simulated time), the time values sent to the hardware simulator must be monotonically increasing. Otherwise, it would generate an inconsistency in the time of the hardware simulator. It should be noted that this condition also eliminates the influence

of the number of vCPUs on the total number of generated synchronization points. This is because the number of synchronization points will depend only on the vCPU that has executed the highest number of instructions, achieving the diagram described in Figure 2.18b.

Since multiple vCPUs can trigger synchronization points at very close instants, a threshold has been introduced to control the minimum number of instructions that must be executed before the next synchronization point happens.

For these reasons, the following expression (Equation 3.1) has been included as a condition of to trigger a synchronization point where T_{SW} is the time of the software emulator and is equal to the maximum time of all vCPU; T_{HW} is the time of the hardware simulator; Th_t is the minimum time that the software emulator can freely advance in the co-simulation virtual platform.

$$sync = \begin{cases} 1 & \text{if } T_{SW} = \max(T_{vCPUn}) > T_{HW} + Th_t \\ 0 & \text{if ohterwise} \end{cases} \tag{3.1}$$

3.4.3. Location of the Synchronization Points

The translation-execution loop performs a translation from the guest TBs to the host TBs and its subsequent execution on the host. As the emulation progresses, synchronisation points must be sent to the external hardware simulator. This allows the software emulator and the hardware simulator to be synchronised. Synchronisation points are virtual interactions and only send the local time notion of each vCPU. Figure 3.4 shows the QEMU flow and where to introduce the synchronization point.

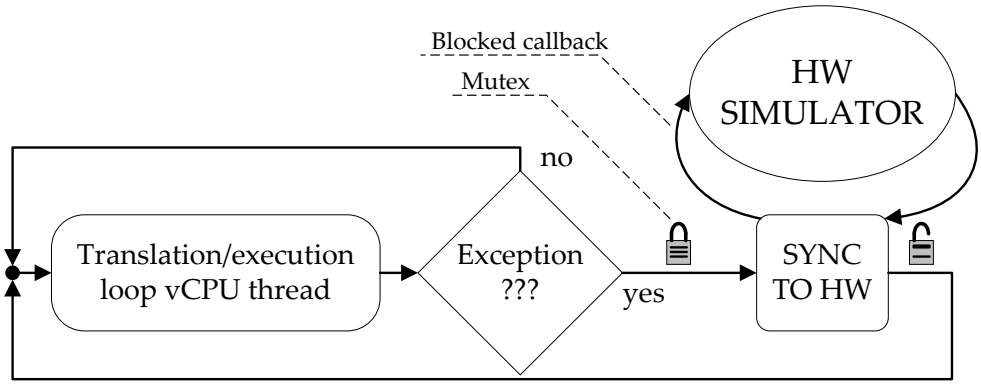

Figure 3.4: Flow chart for the location of the synchronization point.

When an internal/external event needs to break the translation-execution loop, an exception is triggered. The exceptions force to store the state of the vCPU and break the translation-execution-loop in a safe mode. Thus, it allows the remaining processes of QEMU or peripherals to be updated before serving the exception. This is the right time to send a synchronization message to the hardware simulator since the state of the vCPU

is safe and does not change. It should be noted that in the multi-thread case, this point can be reached at the same time by different vCPUs. For this reason, a mutex has been introduced to serialize and protect simultaneous access to the synchronization point. The exceptions of QEMU are triggered by multiple reasons, which are described in QEMU documentation [QEM]:

- Exceeding the maximum limit of executed instructions (*icountMax*). This exception enables carrying out other pending tasks and processing events associated with the management of the whole emulated system. The *icountMax* value is the maximum limit of instructions executed in a translation-execution loop (in Equation 2.1, it equals variable N) and is user-configurable.
- An external interaction toward the vCPU. Which is any signal that is generated outside vCPU flow and modifies its state (i.e., asynchronous interrupts).
- Other exceptions generated by the guest code. Most of them are related to the guest code, the QEMU management of the vCPUs, and the emulated peripherals.

The following code shows the location of the synchronisation points within the QEMU source code. It can be seen that the synchronisation point is located after the translation and execution of the guest TBs, and a mutex protects it.

Listing 3.5: Location of the synchronization point in QEMU source code. file://cpus.c

```
/* Multi-threaded TCG - Tiny Code Generator
* In the multi-threaded case each vCPU has its own thread. The TLS
* variable current_cpu can be used deep in the code to find the
* current CPUState for a given thread.
*/
static void *qemu_tcg_cpu_thread_fn(void *arg){
  //vCPU INITIALIZATION
  //...
  while (1) {
    //...
    //TRANSLATE & EXECUTE TBs
    r = tcg_cpu_exec(current_cpu);
    //...
    //BREAK IF THERE IS SOME EXCEPTION
    //...
    //-------------- SYNCRHONIZATION POINT --------------
    if (COSIL_sync && use_multi_icount) {//if it's enabled
      //MUTEX
      qemu_mutex_lock(&SYNC_mutex);
      //GET INSTRUCTION COUNTER (CLOCK) FOR current_cpu
      int64_t vcpu_clock = qemu_clock_get_ns(QEMU_CLOCK_MULTI_VIRTUAL);
      //MANAGEMENT OF SYNC POINT -- see 3.4.2 section
      if(last_sync_icount < vcpu_clock && (vcpu_clock-last_sync_icount) >= (min_sync_icount)
          {
        //DO SYNC
        COSIL_sync(COSIL_opaque, vcpu_clock, current_cpu->cpu_index, 0);
        last_sync_icount = vcpu_clock;
        last_sync_cpu = current_cpu;
```

```
28          }
29          //MUTEX
30          qemu_mutex_unlock(&SYNC_mutex);
31        }
32        //END TRANSLATION-EXECUTION LOOP
33        //...
34      }
35      return NULL;
36    }
```

In Listing 3.5-Line:21, it can be observed the condition described in Equation 3.1 to execute a synchronisation. Note that Th_t is represented by the configurable parameter *min_sync_icount*.

3.5. Summary

This chapter presents a new proposal for integrating QEMU as a software emulator in co-simulation a virtual platform based on SystemC. The integration enables fast MPSoC + FPGA verification using the MTTCG parallel execution mode of QEMU.

The new proposal is based on a synchronization mechanism between the QEMU and an external hardware simulator. The synchronization mechanism sends messages to the hardware simulator and uses the vCPU local time described for QEMU. Moreover, the mechanism allows setting the timing accuracy of the software emulator with a low impact on the co-simulation performance.

Throughout this chapter, internal details of the synchronisation mechanism have been provided. It explains the method to obtain the timing notion of each emulated guest CPU and the management of synchronisation points in QEMU. It also shows the secure location in QEMU source code where the its execution can be blocked to send a synchronisation point.

Chapter 4

COSIL: Co-simulation Software-in-the-loop

Heterogeneous designs require to use of more flexible tools for verification. COSIL is an open-source tool that advances in this way.

4.1. Introduction

This chapter aims to provide a solution that allows verifying by co-simulation the functionality of the control system and its interaction with the power plant applied to power electronics applications. This solution is presented as an alternative to the use of Hardware-in-the-loop verification platforms. The new verification tool is based on the Software-in-the-loop methodology and virtual platforms to model SoC+FPGA devices and their interactions with the power plant. Thus, the developed tool called COSIL runs the software implemented in the SoC, the hardware designs in the FPGA, and the power plant model simultaneously. COSIL allows working, in a progressive way, with multiple levels of abstraction. It supports functional and temporal simulation techniques, from the most abstract level (Algorithm level) to a level sufficiently detailed to validate hardware designs in FPGAs (RTL − Register-Transfer Level).

COSIL aims to address the limitations found in Software-in-the-loop. As explained in the Section 2.2.2, its main challenges focus on increasing simulation speed and testing the implementation of the whole control system.

To increase co-simulation speed, COSIL takes advance of the high speed of software emulators, among other improvements. Also, COSIL co-simulates the source code of the final implementation without any modification that will run on the embedded processors or the FPGA. Then, it improves the reusability of the source code used in the verification cycle. Table 4.1 summarises the improvements COSIL offers to Software-in-the-loop.

Table 4.1: Comparison of non-destructive tools for verification in power electronics with COSIL

.	**HIL**	**PIL/FIL**	**SIL**	**COSIL**
Tipe of verification	Full-control board	Specific SW/HW modules	Specific SW/HW modules	Full-control code
Price (cheap)	○○○	●○○	●●●	●●●
Compatibility	●○○	●○○	●●●	●●●
Verification speed	●●● (real-time)	●●○	●○○	●●○
Timing Accuracy (power plant)	●○○	●●○	●●●	●●●
Flexibility to changes	●●○	●○○	●●●	●●●
Debug capabilities	●○○	●●○	●●●	●●●
Code re-usability	●●●	●○○	●○○	●●●

The following explains how to use COSIL to verify the control system in power electronics applications. The internal details of COSIL are described. Also, it includes a performance study and results using as a case study a work-in-progress industrial project. The industrial application consists of a control system implemented in a SoC+FPGA device to control a 400 kVA back-to-back (AC/DC/AC) converter.

4.2. COSIL methodology

The design and implementation flow using COSIL is described in Figure 4.1. COSIL is based on the standard design and implementation methodology in power electronics (see Section 2.2.2). First, the design requirements are defined, the power plant is modelled, and the control algorithm is designed using a model-based simulation (Model-in-the-loop - MIL - Simulink). This power plant model will also be used to simulate the power plant (PW) in the COSIL tool.

Once the specification of the control algorithm are obtained, its implementation is designed. Among others, this process determines where it will implement each part of the control algorithm (only software, only hardware or a mix of software and hardware), and the code is generated. COSIL enables the verification of not only the control algorithm but also its integration, and therefore it verifies the whole software and/or hardware infrastructure that makes up the control system. Once the software and hardware are implemented, their operation and impact on the modelled power plant are verified by co-simulation (HW+SW+PW).

The closed-loop link with the model of the power plant allows to use the MIL simulation developed in design phase as a golden reference. Thus, it can be verified that the operation of the final code is correct.

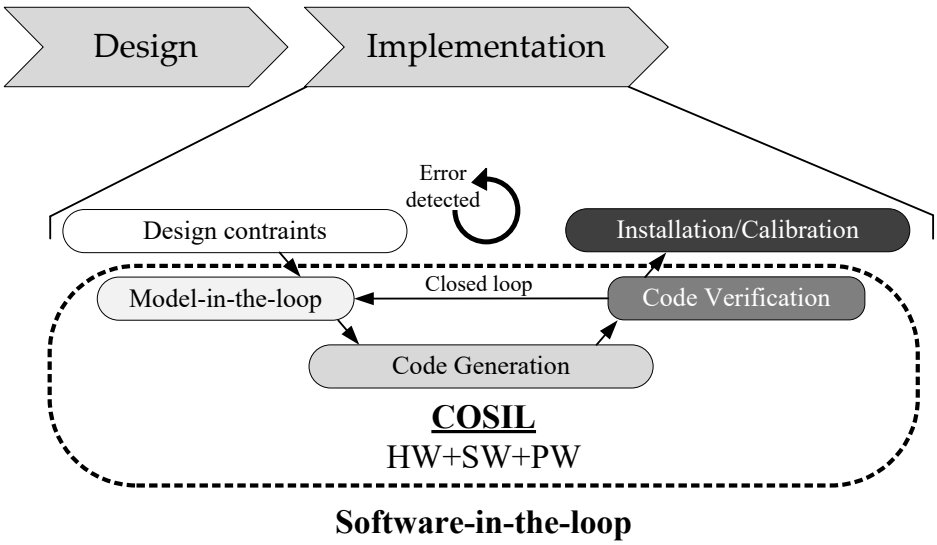

Figure 4.1: COSIL methodology flowchart.

4.3. Platform description

The COSIL verification tool is based on a co-simulation virtual platform managed by a sequential discrete event simulator. It is composed of three modules: power plant simulator (PW – PoWer plant), software emulator (SW - SoC/μP/CPUs), and hardware simulator (HW - FPGA). The simulators/emulator can be seen as modules within the virtual platform, and their synchronisations and communications are performed through messages (Figure 4.2).

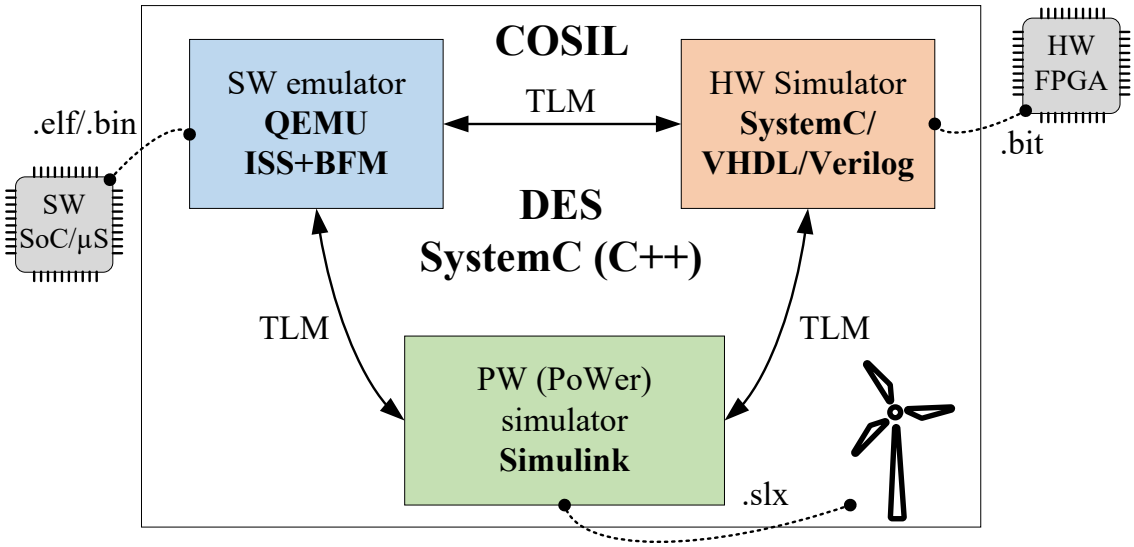

Figure 4.2: COSIL main setup.

These modules are described in SystemC (C++), allowing system-level modelling. This fact facilitates the development of systems in the virtual platform because all modules use the same language, C/C++ code. Furthermore, the SystemC kernel (DES) synchronises all modules. Thus, the environment works in a mono-process setup. It means that all

modules (SW, HW and Power plant (PW)) are executed in the same process from an operating system point of view. Therefore, it is unnecessary to use typical the communication mechanisms found in multi-process applications such as Inter-Process Communication (IPC). This avoids the simulation speed penalties introduced by IPC, replacing IPC with simple function calls [DMB19].

Building full co-simulation in a single process makes easy to get a complete visibility of the status of all simulators. Therefore, it simplifies debugging and tracing data from any point in the power plant, hardware design or software application.

COSIL is based on the Transaction Level Modeling (TLM) abstraction level as a trade-off between abstraction and accuracy. Therefore, the number of events generated in the communication between modules is reduced. TLM reduces the effort of modelling communication and increases the speed of simulation. It models the communication between modules, performing messages like a memory read/write as a single step rather than a sequence of cycle-level events [IEE11].

In COSIL, multiple levels of abstraction are allowed, and RTL designs can be mixed with functional algorithms described in C/C++. At the most detailed abstraction level, COSIL supports instruction-accurate timing for software and cycle-accurate timing for hardware. The power plant timing accuracy depends on the discretisation frequency used with the power plant simulator. Typically, the discretisation frequencies are limited by the behaviour of faster devices such as IGBTs.

Considering the current design work methodologies in power electronics applications, this thesis has used the leading verification tools in each field to improve productivity. Thus, COSIL allows linking the designs with software tools such as Vitis, SDK, Petalinux, Vivado by Xilinx, Eclipse, or Simulink by Mathworks to the SW, HW and PW modules. Thus, it facilities the verification of the final implementation code, which is later installed on the control board, with the power plant model from Simulink (see Figure 4.2).

When the main goal is to verify hardware modules described in VHDL/Verilog/SystemC, the QuestaSim simulator can be used [Sie] as a backplane for the whole co-simulation. QuestaSim is one of the most popular and commercially widespread mixed-language DES, and it supports mixed description simulation where different modules can be described using VHDL/Verilog/SystemC. Note that the use of RTL modules considerably increases the simulation execution time.

The architecture of COSIL can be seen in Figure 4.3, which details the modules that make up the platform. The communication channels that can be used between different modules are detailed. Figure 4.3 will be used to explain COSIL internal details.

As mentioned above, COSIL performs a HW+SW+PW co-simulation mixing three domains: the software domain, the hardware domain and the power plant domain. Each domain can be made up of multiple modules.

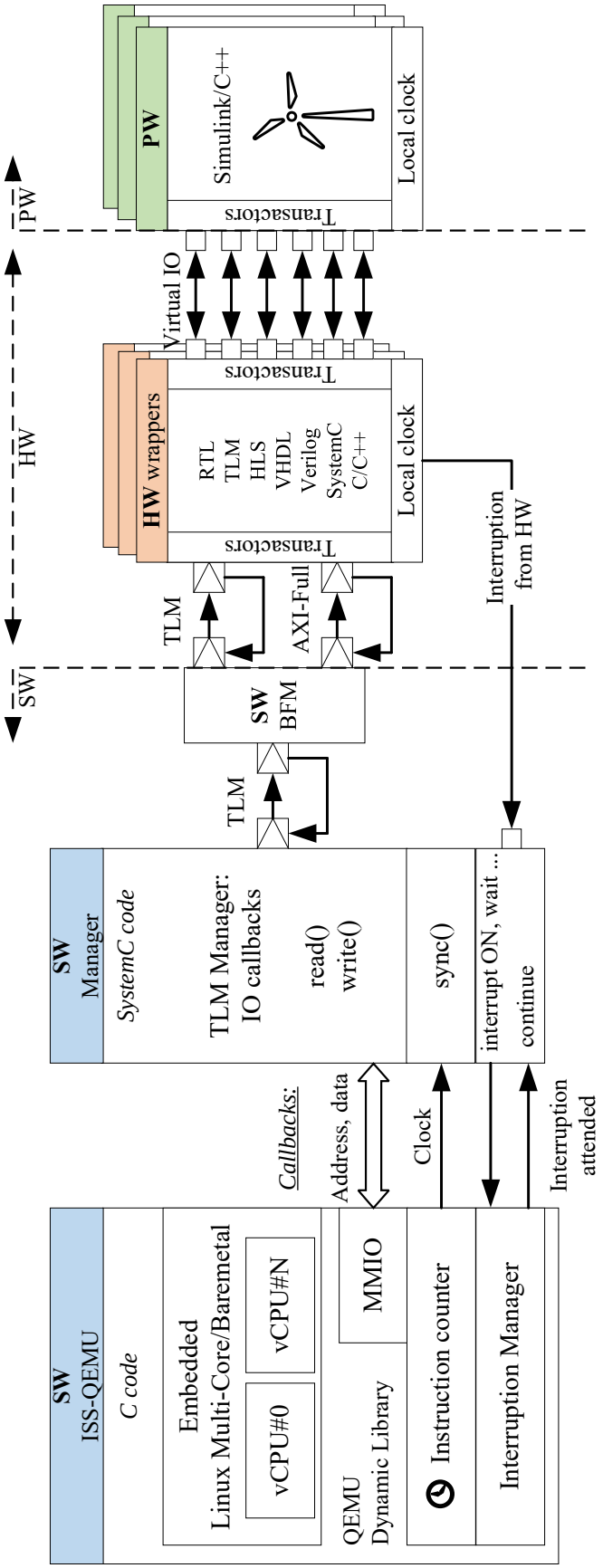

Figure 4.3: Internal architecture of COSIL. Details of SW, HW and PW modules.

4.3.1. SW domain

Modelling the behaviour of a microprocessor is a complex task. The use of emulators addresses this complex task. COSIL uses the QEMU tool (Quick EMUlator). Currently, QEMU is the most compatible open-source software emulator on the market [QEM]. QEMU allows working with a multi-thread translator core. The use of parallelised emulators helps to improve the performance of SIL systems. Therefore, both multi-core operating systems and bare-metal applications can be verified. A virtual CPU (vCPU) is created for each guest CPU to translate and execute the instructions.

The SW module can be seen as a chip based on an ISS (Instruction Set Simulator) and a BFM (Bus Functional Model). The ISS reproduces the behaviour of the instructions executed by the processor. Otherwise, the BFM reproduces the external interface of the processor without providing any internal detail. The external interface is composed of the buses that communicate the processor with the FPGA and the interrupt inputs [SG]. The BFM is implemented as a state machine controlled by an API. When the ISS detects memory access, it accesses the BFM API to perform the access.

The ISS includes the processor emulator (QEMU). The ISS is also called *SW-QEMU* in this book. It is possible to replace the ISS with a SystemC module (*SW-SystemC*) that functionally represents the software behaviour. In literature, this option is also called host Host-Compiled Simulation [Ger10]. Then, it eliminates the workload of QEMU and achieves a faster simulation of the hardware architecture. In this case, it does not verify the same software code, but only the code sections that it wants to verify.

QEMU includes some modules to detect situations where it is required to synchronise its state with external simulators. These situations are typically interactions with the hardware, such as input/output accesses, interrupts or synchronisation points (see Section 3.3). The input/output accesses are managed by the MMIO module (Memory Map Input Output). The Interrupt Manager module manages the interrupt request coming from the HW (FPGA). Finally, forced synchronisation points allow share timing notion between SW, HW and PW and running the co-simulation in a synchronised way.

The software timing accuracy is configurable and relates to the maximum number of guest instructions that can be executed before sharing the CPU status with the HW or PW. The number of instructions executed by each vCPU is included in the CPU status information, providing the software execution timing with instruction accuracy. The *Instruction counter* block of the ISS-QEMU provides the time notion of the emulated machine. The timing and synchronisation details of QEMU have been described in Chapter 3.

The BFM decodes the accesses between software and hardware depending on the communication bus used by each hardware peripheral. The following communication mechanisms have been developed: TLM and Advanced Microcontroller Bus Architecture

(AMBA) Advanced eXtensible Interface (AXI). The TLM [IEE11] is intended for designs with a high abstraction level, while the AXI protocol [ARM] is intended for RTL modelled.

Between the ISS and the BFM, there is a submodule called SW Manager (Figure 4.3). The *SW Manager* submodule include an API for the BFM. It manages the I/O accesses between SW and HW and the synchronisation between ISS (QEMU) and the rest of the modules (HW and PW). It is composed of callbacks (read/write/sync) called from QEMU. The callbacks model the interactions or synchronisation. In addition, the callbacks get state information of the vCPUs and the addresses and access data exchanged between SW/HW. This block also manages the interrupts generated in the HW and applies them to the SW emulator.

4.3.2. HW domain

Due to the different levels of abstraction that HW modules can present, a generic module has been made to encapsulate each hardware design. This generic module is called *HW wrapper* and includes blocks called transactors [BP07]. The transactors translate the communication protocols between TLM/AXI depending on the level of abstraction.

Usually, the workload in RTL simulations is very high. A method to relieve this workload is based on reducing the number of events and increasing the simulation speed to raise the level of abstraction in designs. For this purpose, COSIL supports the use of mixed abstraction levels designs.

In the early stages of the design cycle, where functional verification of the system is carried out, code auto-generation tools such as Matlab Simulink Embedded Coder can generate C++ code for the control algorithms. The latter option allows testing the influence and operation of these algorithms in the whole environment (SW, communications and HW) quickly.

In the advanced design cycle, it is possible to integrate RTL modules generated from HLS synthesis tools or code them directly in VHDL, Verilog or SystemC. Thus, the RTL model tested in the virtual platform behaves identically to the one used in the final implementation.

The current trend in power electronics is to use HLS tools to generate RTL models from C code [BSD+20]. Following this trend, it is possible to take advantage of such tools to generate RTL equivalent code in SystemC from the C description of the HW design. The advantage of SystemC code is that it is a compiled language in contrast with interpreted HDL languages (like VHDL or Verilog), reducing the workload and increasing the co-simulation speed [DMB19].

Clock management in hardware modules is critical. In the power electronics area, the control algorithm usually runs periodically depending on the acquisition frequency. It is only at these times that the hardware modules work. When the hardware modules finish,

only events linked to the clock signal are attended. However, until the next acquisition event, the hardware modules are idle. Thus, there are vast unnecessary events generated by the global system clock. Local clocks are used in each HW Wrappers module to suppress these additional events instead of a global clock for the whole system.

Local clocks offer the possibility to only wake up the clock signals of the HW module during the time required to complete its task. Once it completes its task, it turns off the local clock signal. The local clocks are enabled by any interaction with the hardware design and stop when the maximum latency assigned to each hardware design elapses.

4.3.3. PW domain

The PW modules includes the power plant simulation models discretised and automatically translated to equivalent C++ code. It has a local clock whose frequency is the discretisation frequency. The integration of the power plant into the simulation environment is based on Matlab Simulink Embedded Coder. However, it can be based on discretised equivalent models and coded in C++.

As in the HW designs, communications of PW with the HW/SW modules include transactors. These transactors abstract the details of the interface between the HW/SW and the power plant.

The interface between the control board and the power plant has been modelled by immediate TLM channels specially developed for COSIL and called VirtualIO (Figure 4.3 - VirtualIO channels). These channels use double precision floating point to keep compatibility with the native Matlab Simulink precision.

VirtualIO uses immediate TLM transactions. Immediate TLM transactions are also called Direct Memory Interface (DMI)-TLM [IEE11]. DMI-TLM transactions are a type of transaction that has no communication latency. In fact, it uses the slave memory pointer to access directly to slave.

The requirements and libraries supported by Embedded Coder, as well as the limitations and quantisation errors, have been well studied by Matlab [Mat21]. The quality of discretisation depends on the optimisations that have been configured in Embedded Coder. In addition, the result also depends on the version of the C/C++ code generator used. Therefore, the discretisation error will depend on the model, library, frequency, and even optimisation or program version used.

For the sake of simplicity, the Embedded Coder Quick Start interface has been used to generate the PW modules in this thesis. Both regular and Simscape library models have been transformed. No limitations have been found to get the C++ code equivalent to the power plant.

After Embedded Coder Quick Start configures the model for basic code generation, it is possible to do a fine-tune optimisation to adapt the generated code to the high-level

code generation objectives. Some of these high-level objectives are execution efficiency, Random Access Memory (RAM) efficiency, traceability or debugging. In this thesis, the execution efficiency goal has been selected as the main goal is optimizing the co-simulation speed.

4.3.4. Synchronization

COSIL relies on the SystemC kernel to synchronise SW, HW and PW modules. The SystemC kernel is a *sequential DES* kernel, so it executes events chronologically in a *sequential single-process architecture*. COSIL uses a *conservative asynchronous synchronisation* approach as a balance between timing accuracy and performance. Synchronisation must be performed whenever there is an interaction between two different modules (SW, HW or PW), i.e. when there is communication. However, the high number of synchronisations between multiple modules could drastically reduce the co-simulation performance.

Temporal decoupling defined in TLM 2.0 has been used to implement synchronisation in communications. This mechanism allows to reduce the number of synchronisations and thus increase the performance of the co-simulation. However, it introduces a timing error in the interactions. The use of temporal decoupling implies that each module has a local time and it is released when the next *temporal barriers* are overcome. The period of these barriers is called *Quantum Time*. It is the same concept that *look-ahead time* explained in Section 2.1.2. If any module exceeds this barrier, it is blocked, waiting for the rest of the modules to reach that time. It is recommended to check Section 2.4 for more details about temporal decoupling.

The implementation of this type of synchronisation through temporal decoupling involves using a SC_THREAD per module and using different time types. These types are:

- Global time. Simulation time (T_{Sim}). This is the time used to manage the global events queue in the DES. Please do not confuse the T_S parameter with the sampling period typically used in the design of the control algorithm. In that section, it talks about simulation times.
- Local time (T_{SW}, T_{HW}, T_{PW}). This is the local time of each module. The local time is updated with the global simulation time only when it passes Quantum time barriers or when the SystemC kernel executes an event. Local times are not managed by SystemC DES.

Focusing on the SW module, COSIL allows two options for software verification. The timing of the SW module depends on the option chosen.

- SW-SystemC (host-compiled simulator). Designed for quick verification. In this

case, the SW module time is managed by itself. Thus, from the DES point of view, its management is similar to the HW and PW modules.

- SW-QEMU (emulator). Designed for software source code emulation. Here the advance of the SW module is managed externally to the DES. It is QEMU that sends the temporal information of the vCPUs and allows to increase the time of the SW module. Then, the SW local time increment is controlled by QEMU. If QEMU stops, the co-simulation will be blocked because the rest of modules will stop in the next Quantum time barrier.

A practical example of synchronisation in COSIL will be presented below. This example aims to obtain the voltage measurement of the power grid using a SoC and an FPGA. Even though in practice we can have multiple SW, HW or PW modules, to simplify the example, it will be assumed that it has one SW module, one HW module and one PW module. For this purpose, it is necessary to simulate the power grid in a PW module and simulate an acquisition IP located in the FPGA with a HW simulator. The design, therefore, divides the functionality into three environments: CPU (bare-metal) application, FPGA (Acquisition - ADC) and Power Plant (mains/power grid). If the mains voltage value exceeds a specific limit, the ADC interrupts the bare-metal application to warn of a fault. It can be seen the time advance of the co-simulation in Figure 4.4.

For this example, the HW module (Acquisition - ADC) spends 1.5 μs (latency) to read data from the power grid at 200 kHz. This latency symbolises the time it takes to get data from the sensors of the plant connected to the power grid. The PW module is executed at 1 MHz as this is the frequency that has been used to discretise the power plant. Both the HW and PW modules are executed in a loop using the *wait(time)* sentence to indicate the DES when they are going to be executed. For the sake of simplicity, the communication between the HW and PW modules is latency-free. The global simulation time is denoted as (T_{Sim}).

The choice of the Quantum time is critical. The Quantum must be chosen small enough not to swamp the synchronisation behaviour of the co-simulation. In this example, a Quantum time of 5 μs has been chosen to represent possible timing errors. This value has been chosen as it is a multiple of the period of the PW module, the one which advances the fastest. So every five PW executions, all the modules must be synchronised. Some tips for selecting the Quantum time will be given at the end of this section.

Eight points are highlighted in Figure 4.4.

1. The SW manager starts in an idle state waiting for some interaction from QEMU. When QEMU accesses the SW Manager, it sends its time notion. Then, an internal variable of SW Manager called tsw_{var} is increased. This variable accumulates the time increments coming from QEMU ($tsw_{var} = tsw_{var_{-1}} + \Delta T_{QEMU}$). If after the updating of tsw_{var}, it exceeds the next Quantum time barrier, the SW module will

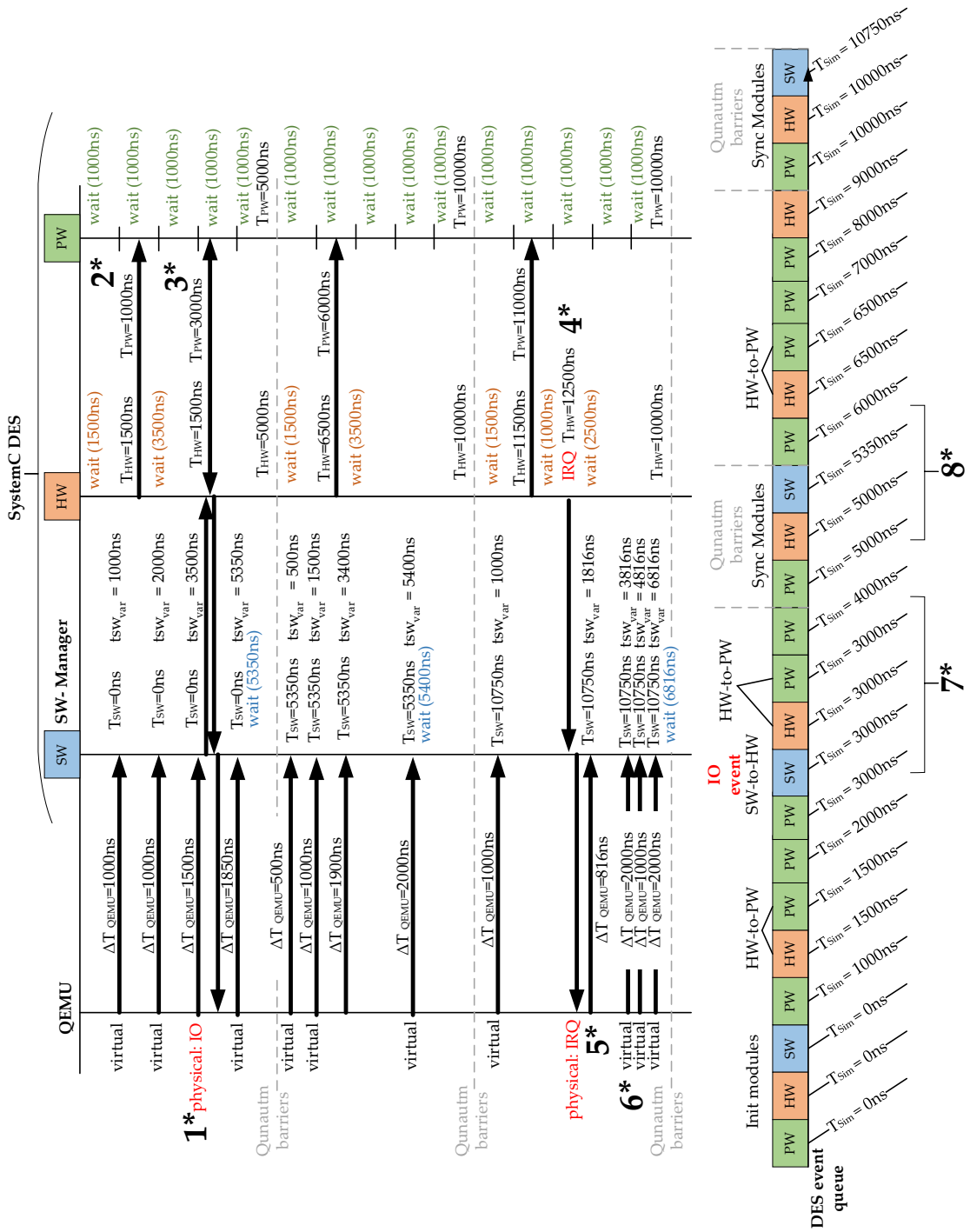

Figure 4.4: Example of syncronisation of a SW, HW and PW module using SW-QEMU.

block until the global simulation time reaches the new T_{SW}. The blocking is done using the *wait(time)* sentence. When the rest of modules reach the accumulated tsw_{var} the SW Manager can continue and the SW local time is updated with tsw_{var}.

Remember that there are two types of access: virtual or physical. Virtual accesses depends on the guest emulated code and the operating system where QEMU is executed, and they are limited by the *icountMax* and Th_t parameters (see section 3.4.3). Physical accesses (IOs and Interrupt Request (IRQ)) are asynchronous and instantaneous (it does not has latency in the QEMU-SW module communication). In the example, an asynchronous IO access occurs. This access is blocking and stops QEMU while the *SW-manager* manages the access.

2. An access is being performed from the HW module to the PW module. An example of this type of access is a reading of the voltage and current sensors by the ADC FPGA-controller. At this moment (T_{Sim} = 1500 ns) the PW was blocked with a wait(1000 ns) sentence. Therefore, from the HW point of view, the PW module time is T_{PW} = 1000 ns. This generates a timing error of 500 ns in synchronisation.

3. There is access from the HW module to the PW module made by QEMU I/O access. Same as case 2, the notion of time between SW, HW and PW is different. This generates a timing error due to a relatively high Quantum time compared to the faster module (PW).

4. The HW module detects an overvoltage in the power grid. Then, it sends an interrupt to the SW module, which notifies QEMU to deal with the interrupt.

5. When QEMU has attended the interrupt event in its *event queue*, it returns its time notion. This does not imply that the interrupt handler started to be executed.

6. QEMU runs on the host operating system. Therefore, the interaction between QEMU and the SW Manager is *not* temporally deterministic. The time advance of QEMU depends on the workload of the operating system where it is executed. However, its time advance (ΔT_{QEMU}) is also limited by *Th* and *icountMax*.

7. In the DES event queue, only the global simulation time is taken into account. Here all events to be processed in the co-simulation are reordered. At this point, although $T_{SW} = 0ns$, $T_{HW} = 1500ns$ and $T_{PW} = 3000ns$, the simulation time is assigned by the last timestamp ot the executed event. Therefore, from the DES point of view, these interactions are performed at time $T_{Sim} = 3000ns$.

8. When the Quantum barrier is overcome, all modules must be synchronised.

The bottom of Figure 4.4 shows how the SystemC DES event queue would be organised for this example. Here, it can be seen how the kernel does not know any aspects of the local time of each module but works with the global simulation time.

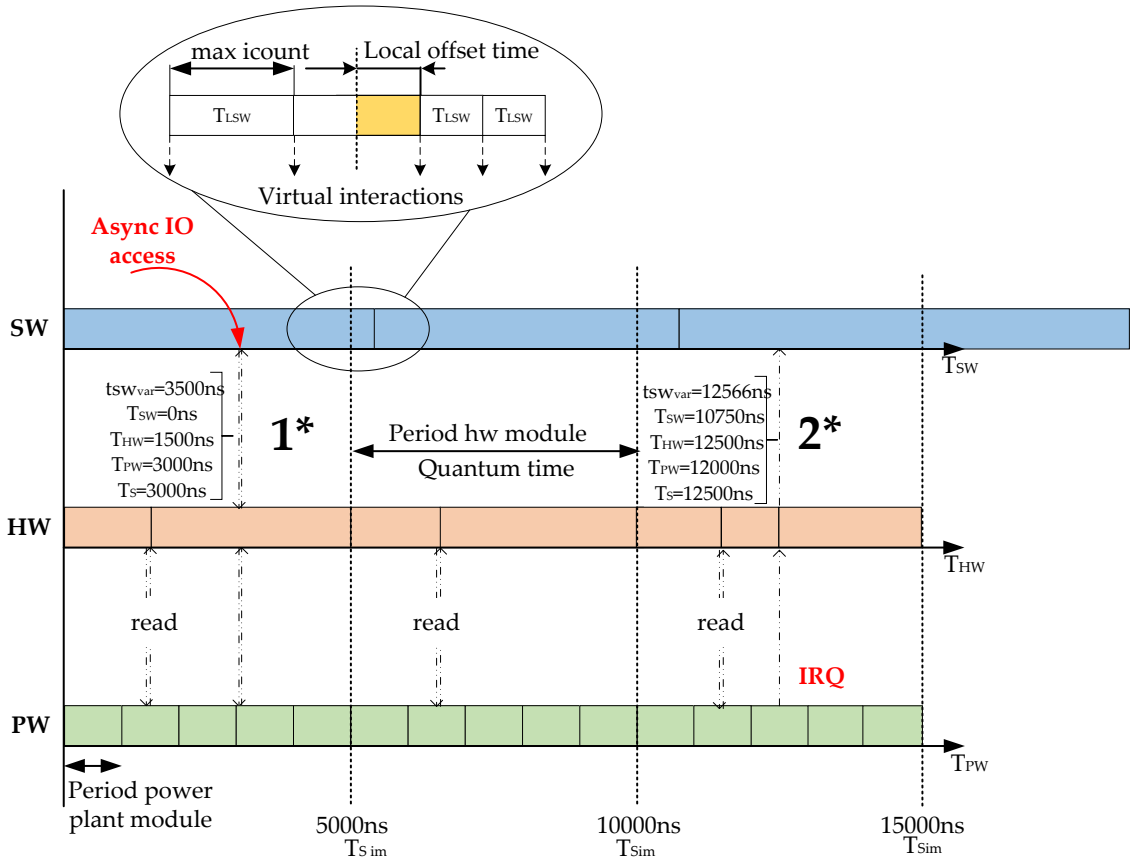

Figure 4.5: Representation of Figure 4.4 from the point of view of DES SystemC.

As it has been shown, there are many cases where timing synchronisation errors appear, i.e. when there is a communication between the modules at different times ($T_{SW} \neq T_{HW} \neq T_{PW} \neq T_{Sim}$). These situations usually happen in synchronization mechanism based on temporal decoupling. Figure 4.5, again represents the example of the previous Figure 4.4. In this case, it shows a timeline of how the simulation would look from DES perspective. In the following, the timing errors that exist when the Quantum is not properly set can be better observed.

1. An IO access is performed from QEMU. This happens at $T_{Sim} = 3000$ns. This is the instant when the DES detects this event. The difference between the global time and the local time of the modules provides the timing error.
   - SW Error: $T_{Sim} - T_{SW} = +3000$ ns
   - Error HW: $T_{Sim} - T_{HW} = -1500$ ns
   - Error PW: $T_{Sim} - T_{PW} = 0$ ns
2. Interrupt is triggered and sent to QEMU. This happens at $T_{Sim} = 12500$ ns since this is the instant at which the HW module executes the event to trigger the IRQ. In this case the timing errors are:
   - SW error: $T_{Sim} - T_{SW} = -1816$ ns

- HW error: $T_{Sim} - T_{HW} = 0$ ns
- PW error: $T_{Sim} - T_{PW} = -500$ ns

Note that the absolute value of all errors is less than the Quantum time (5000 ns). Reducing the Quantum time makes it possible to eliminate these differences as all modules will have to synchronise more times. However, too many synchronisations are also not recommended as it drastically reduces the performance of the co-simulation.

To minimise these timing differences, the following tips are suggested:

- All interactions should be performed in multiples of the fastest module. In Figure 4.4, this would be the PW module.
- Some authors like [Emb10] has commented that Quantum time must be small enough not to swamp the timing behaviour of the system. Around 10-50% of the smallest would be a reasonable time to use as a Quantum time.
- Please consider the required SW time precision taking into account HW and PW modules. A high value of K-CPI will cause the SW to advance too fast and lose timing resolution in the SW. A small value will result in slow progress, and more instructions will have to be executed to advance the same amount of time.

4.3.5. Debugging features

Debugging the SW, HW or PW is highly desired for analysing system functionality. COSIL mixes two environments, QEMU and SystemC. Therefore, it is critical to stop both environments at the same time to perform step-by-step debugging.

Temporal decoupling implements blocking communications. When communication takes place, the initiator is blocked until the target returns control to it. This is precisely the key to being able to block the progress of all modules and allow debugging.

In the case of co-simulation blocking from SystemC, the synchronisation between QEMU and the SW-Manager module will cause the QEMU process to stop, blocking all its threads. Remember that every N instructions QEMU sends a callback to SW-Manager to transmit timing information by means of virtual interactions. If the local time of the SW, managed by the SW-Manager, exceeds any Quantum barrier, the DES will block the whole SW module, including QEMU.

When it wants to block the progress from QEMU, COSIL has two alternatives. The less intrusive option is based on using QEMU's debugging support. QEMU includes a framework to link the *gdb* tool. In this case, when QEMU detects a breakpoint generated by gdb, it generates an exception in the translation-execution loop. Then, a call function is sent to SystemC (in SW module), in which a breakpoint was set up. This allows blocking the advance of the HW and PW modules. Thus, all modules are stopped at the

same instant, allowing step-by-step debugging. However, working with gdb is currently quite tedious.

Another easy but more intrusive approach is to add accesses to memory regions reserved for debugging. Such accesses do not involve any latency. Like the accesses also produce a call function to the SW module, it can fix a breakpoint similar to the gdb option. In addition, using an API that accesses the reserved memory location, it is possible to trace the software variables and see their progress during the simulation.

Figure 4.6 shows an example of the path that is taken to block the DES access when it is accessed from QEMU using memory accesses.

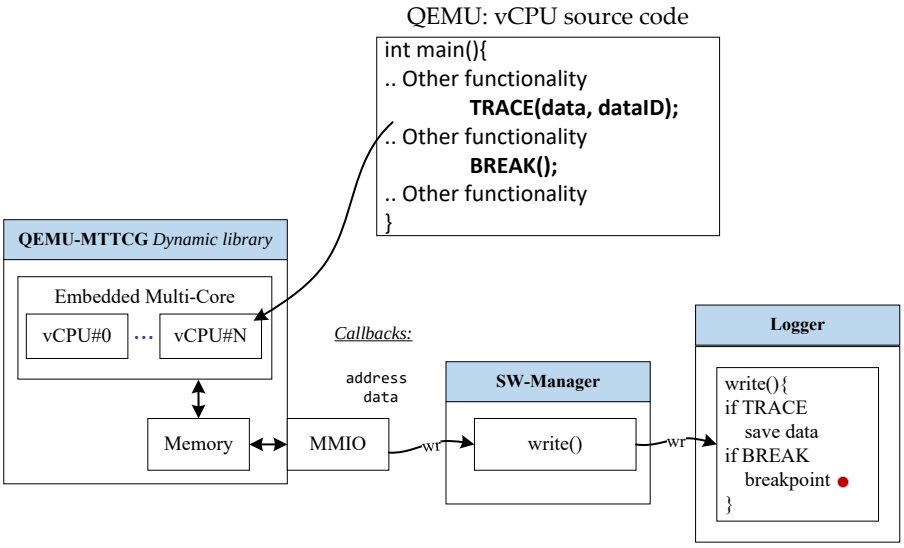

Figure 4.6: QEMU debug flow using code intrusive option.

4.4. Summary

This chapter presents the verification SIL-based tool called COSIL as an alternative to the HIL and PIL systems. Its functionality has been designed for power electronics applications, although its use could be extended to control applications in general. This tool is based on a co-simulation virtual platform that links the hardware and software designs with the power plant, closing the control loop in the same workstation.

The presented tool allows verifying exactly the final HW/SW modules of the multi-core SoC+FPGA based control board without using a HIL system. Thus, it helps to reduce the errors that happened in the code writing phase and integration phase. This new SIL-based alternative provides more debugging and abstraction capabilities in the simulation and it reduces the verification costs

Throughout this chapter, it has been described how to use COSIL and its internal architecture. It has also been explained how the SW, HW and PW domains are synchronised.

Chapter 5

Performance analysis and test

...

5.1. Introduction

This chapter focuses on bringing together all the tests carried out to analyse the thesis contributions. The tests have been grouped into two sections, which have the following goals:

- Analyse the impact on the performance of the proposed modifications in QEMU source code. The proposal supports an external synchronisation with the hardware simulator running in MTTCG mode.
- Test the reliability, performance and synchronisation of the developed COSIL co-simulation tool with a real power electronics application. Such a tool may include QEMU as a software emulator. COSIL eases the verification process of the source code integration of multi-core mixed software/hardware projects in closed-loop simulation.

The tests were performed on a workstation with an Intel Core i7-10750H processor and 16 GB of RAM.

5.2. QEMU external synchronisation mechanism

This thesis aims to integrate QEMU-MTTCG in a co-simulation virtual platform that uses the SystemC kernel as Discrete Event Simulator manager and hardware simulator. The presented external synchronization mechanism is essential to synchronize QEMU-MTTCG as the software emulator with the hardware simulator. Within the virtual

platform, QEMU is integrated into the SystemC simulator as a dynamic library. Besides, a TLM 2.0-based API has been developed to provide the necessary communication infrastructure between QEMU and the hardware simulator.

This test and results section presents the influence of virtual interactions (synchronization points) and physical interactions (Input/Output/interrupt).

A Symmetric Multi-Processing Linux (MPSoC Zynq7000-ARM Cortex-A9 dual-core) has been considered as use case, such as in [AKS16], to test virtual and physical interactions and using different setups. To analyse the performance of virtual interaction, it will run the Linux boot and the ParMiBench benchmark [ILG10], which is specialized in multi-core embedded devices. Then, to test physical interactions, a real project of a hardware/software system will use it.

The simulation performance can be calculated as the ratio between the guest simulation time considered by our virtual platform and the time consumed to run the simulation, also known as wallclock time [Fuj01]. Since the co-simulation virtual platform is executed under an operating system, the running time is not deterministic and can vary. Therefore, each setup runs ten tests, and the average and variance of the host execution time (wallclock time) have been calculated.

5.2.1. Overhead of virtual interactions in co-simulation

5.2.1.1. Definition of methodology

This section aims to define the methodology used to test and analyse the possible impact on the QEMU performance of the contributions. In particular, it will analyse the speed reduction exclusively due to synchronisations points or virtual interactions, which means the number of synchronisations (N_SYNC) in the co-simulation environment. No hardware modules were added to focus on the changes introduced in the software emulator (QEMU). Furthermore, it is checked that the time of the hardware simulator is correctly synchronised with the software emulator.

Three different setups have been considered using a Linux boot to compare the overhead introduced:

- Setup#1: Real hardware platform. Linux boot in the physical system (ZedBoard-Zynq7000). This test shows the real execution time in the real platform and can be used as a golden reference.
- Setup#2: QEMU-PetaLinux. Linux boot using QEMU-MTTCG provided by PetaLinux. This setup shows the execution time using the QEMU version included in PetaLinux without any modifications. This can be considered as the original MTCCG implementation against which our solution will be compared.

- Setup#3: QEMU-VP. (QEMU for virtual platforms) Linux boot using the version of QEMU-MTTCG patched with our external synchronization mechanism. In this setup, QEMU is running from a SystemC-based virtual co-simulation (see Figure 5.1).

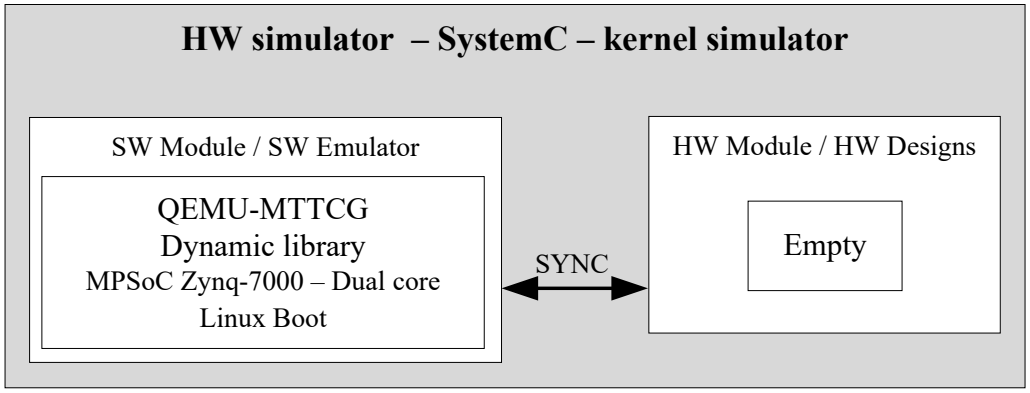

Figure 5.1: Setup#3. QEMU-MTTCG inside a co-simulation virtual platform as software (SW) emulator.

In previous setups, the number of synchronizations performed between the software emulator and the SystemC simulator (hardware simulator) is evaluated. The results are evaluated for different values of the Th_t parameter (see Equation 3.1) and the maximum limit of instructions executed by the QEMU translation-execution loop (*icountMax*, also called icount budget). In addition, the overhead introduced by the number of synchronizations on the simulation speed is analysed.

The ParMiBench benchmark was used in order to measure the overhead when executing other types of applications different from a Linux boot. ParMiBench [ILG10] is a benchmark specialized in multiprocessor-based embedded systems. It is a parallelized version of MiBech [GRE+01] and specialized in the embedded devices. ParMiBench provides four categories: Automation and Industry Control, Network, Office, and Security. In this case, for the Setup#2 the benchmark is executed by directly launching QEMU-PetaLinux code source, which prevents the overhead added by the PetaLinux managing scripts. Table 5.1 summarizes details on these applications and the inputs used for each case.

5.2.1.2. Results

Multiple tests have been carried out using Setup#3, sweeping the Th_t and *icountMax* parameters. The number of synchronizations and the wallclock time consumed in a Linux boot has been measured in each test. These test results are shown in Figure 5.2, using a logarithmic scale for all the axes. As it can be observed in Figure 5.2a, the number of synchronizations can be controlled by the *icountMax* and Th_t parameters, as indicated by Equation 3.1. It can be seen that the number of synchronizations increases exponentially as Th_t and *icountMax* parameters are diminished.

Table 5.1: ParMiBench benchmarks descriptions with input configuration.

Benchmark	Categories	Summary	Configuration
basicmath	Automation	It makes mathematical calculations such as cubic function solving, angle conversions, and integer square root.	Large data set: 1 Giga numbers.
bitcount	Automation	It measures the processor bit manipulation capability by counting the number of bits.	An input of long type (31 bits with 1).
susan	Automation	It is an image recognition application, which detects corners and edges.	PGM picture: 2.8 MB.
patricia	Network	It uses a sparse leaf nodes-based data structure used instead of a full-tree.	Text file containing 5000 IP ad-dresses.
dijkstra	Network	It computes the single-source and all-pairs shortest paths in an adjacency matrix graph.	All-pairs: 160 x 160 matrix
stringsearch	Office	It gets a specific word in several given phrases by using case sensitive or insensitive comparison algorithms.	Input data set size: 32 MB 1024 patterns or keys of length (m): 5
sha	Security	Iterative one-way hash function cryptographic algorithm	-P 2 (Dual-core)

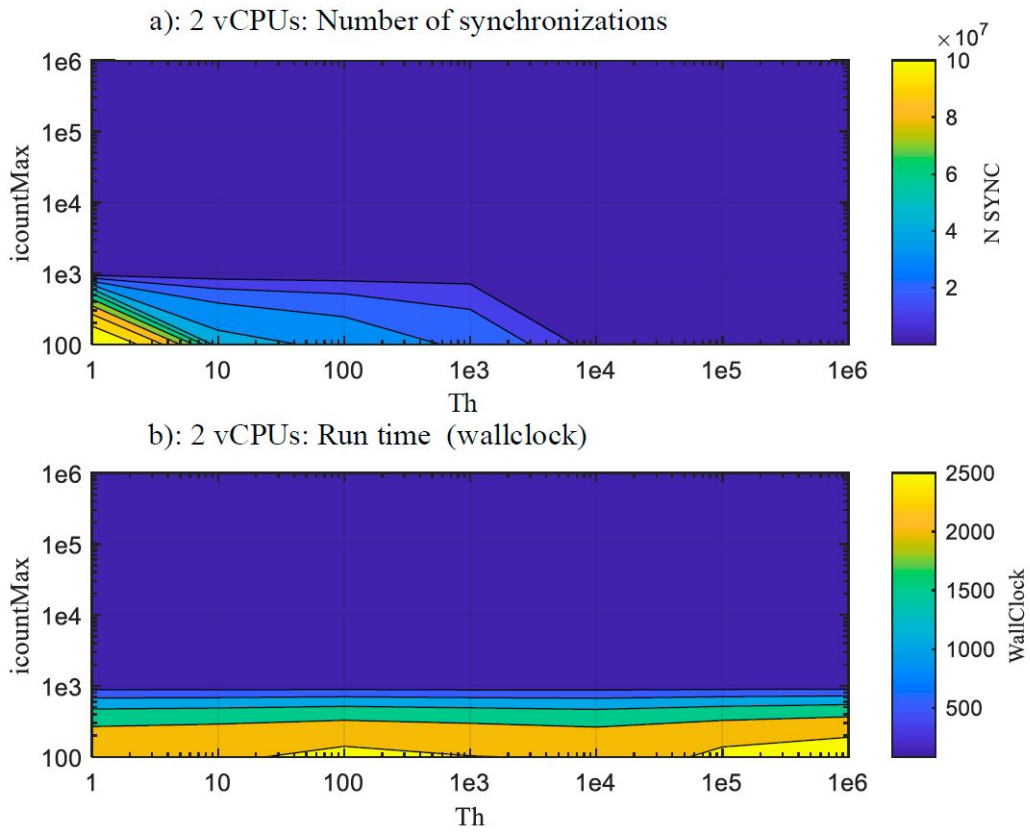

Figure 5.2: Results of the number of synchronization and host runtime in seconds (wallclock) for different setups of Th_t and *icountMax*: (a) number of synchronizations (N SYNC); (b) Wallclock.

Figure 5.2b shows that the *icountMax* parameter is the only one with a real impact on the wallclock time (wallclock in seconds) of the software emulation. Thus, it can be concluded that the value of the Th_t parameter by itself does not influence the host simulation time or wallclock time. Hence, it provides the user with a handy mechanism to define the timing precision of the software emulator. The reason is that *icountMax* forces the QEMU translation-execution loop to break when *icountMax* instructions are executed. This stop breaks the chains between the TBs, decreasing the high performance of the QEMU translator-execution loop, consequently slowing down its simulation speed significantly.

It should be noted that *icountMax* represents the maximum limit of the software timing precision. By default, its value in QEMU is 256. For compatibility reasons and based on the study carried out in this work, this value must not be modified to preserve the high performance of QEMU unless it is necessary to have higher timing precision for the software emulator. In any case, *icountMax* is user-configurable in our virtual platform.

Figure 5.3a shows that the Linux boot execution times for Setup#2 (QEMU-PetaLinux) and Setup#3 (QEMU-VP) with our modifications are similar. Thus, it is

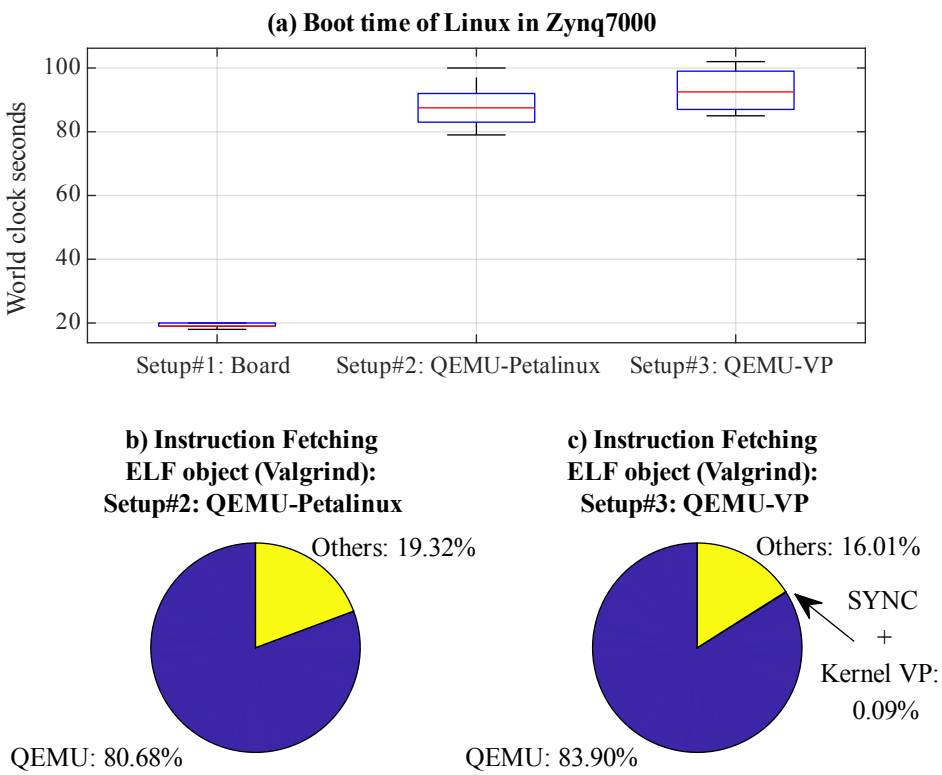

Figure 5.3: Linux boot time (wallclock) for different scenarios: (a) boot Linux time in different plat-forms; (b) Instruction Fetching profiling for Setup#2:PetaLinux; (c) Instruction Fetching profiling for Setup#3:QEMU-VP.

demonstrated that the changes made do not generate a significant overhead for virtual interactions.

In Figure 5.3b and Figure 5.3c, it shows where it spents the CPU time using the percentage of instruction executed during the simulation for Setup#2 (QEMU-PetaLinux) and Setup#3 (QEMU-VP). The data were obtained using the Valgrind tool. As it can be observed, the modifications introduced to allow the integration of QEMU-MTTCG on our co-simulation virtual platform has a small impact on run time, and it does not exceed 0.1% of instructions executed.

Figure 5.4 shows the results running on the Linux platform of the execution time obtained for each ParMiBench application. Tests have been done for Setup#2-QEMU-PetaLinux and Setup#3-QEMU-VP. The figure shows the relative increment of the wall-clock time for QEMU-VP when compared with the QEMU-PetaLinux. Consequently, a result of x% means that QEMU-VP is x% slower than QEMU-PetaLinux. It is concluded that under this type of applications, synchronisation points decrease performance by 5% on average.

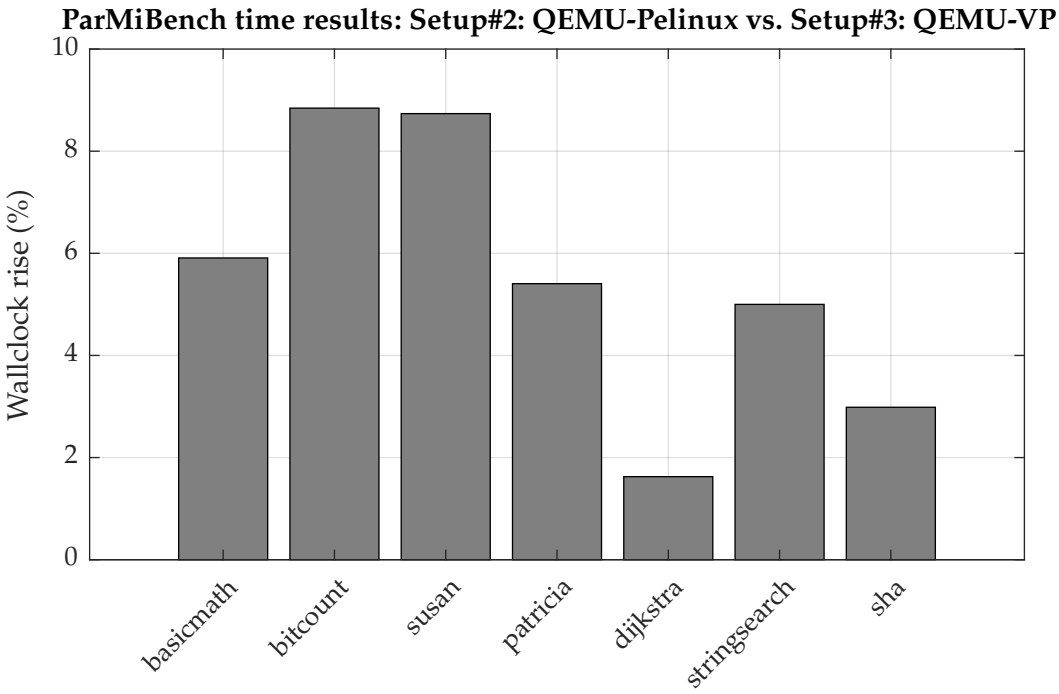

Figure 5.4: Wallclock time comparation for ParMiBench. Setup#2 QEMU-PetaLinux vs. Setup#3 QEMU-VP.

Finally, Figure 5.5 shows the instructions and synchronizations executed by each vCPU in the first 100 seconds (wallclock) of the Linux boot in Setup#3, and it reflects the behaviour described by Equation 3.1. During the Zynq-7000 boot (start-up), CPU#0 behaves as the Master CPU; hence, vCPU#0 is the first to boot the system and later wakes up vCPU#1. This behaviour can be seen in the first 10 seconds of Figure 5.5-top. Figure 5.5-bottom shows what CPU has the preference to synchronize. When vCPU#0 is the only active core, all synchronizations are generated by it. Once the vCPU#1 wakes up,

and the number of instructions executed by vCPU#1 is more significant than vCPU#0, the synchronizations are no longer managed by vCPU#0.

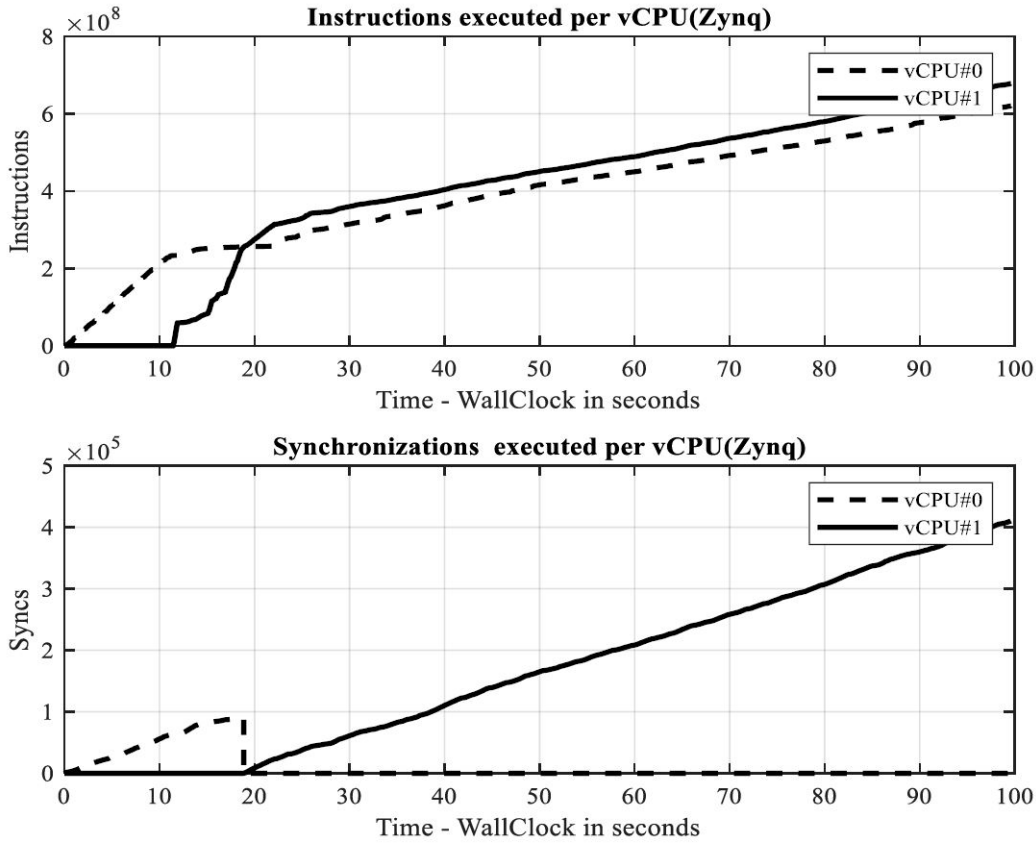

Figure 5.5: Sync management as a function of vCPUs: (Top) instructions executed per vCPU; (Bottom) synchronization executed per vCPU.

5.2.2. Overhead of physical interactions in co-Simulation

5.2.2.1. Definition of methodology

This section aims to define the methodology used to test and analyse the possible impact on the performance of QEMU of the contributions. In particular, it will analyse the speed reduction exclusively due to physical interactions, which means the communication between the software emulator and the hardware simulator, i.e., input/output (I/O) access and interrupts. To remove the effect of virtual interactions in the co-simulation, both $icountMax$ and Th_t remain constant. In detail, it has set 65536 and 10000 instructions to $icountMax$ and Th_t respectively.

In order to analyse the impact of physical interactions, a Zynq (SoC+FPGA) based design has been developed that periodically exchanges information between the software simulator (CPU) and the hardware simulator (FPGA) through input/output accesses and interrupts. In this project, the software emulator accesses to external hardware (peripherals) using a interrupt driven approach. Since the thesis focuses on power electronics applications, a power grids monitoring system has been chosen as a case study.

Using that case study, a sweep of the number of I/Os per time unit has been carried out, increasing the number of writes/reads executed for each interruption (every 100 μs). Thus, it will test the influence on the simulation speed of I/O accesses and interrupts between the software emulator and the hardware simulator.

As the influence is expected to be more significant as the number of I/O accesses increases, profiling techniques have been used again to analyse the percentage of the number of instructions executed at the maximum I/O rate of the sweep. In each test, one second is simulated with a precision of 10 ns in the hardware simulator. The maximum CPU I/O throughput estimated for the Zynq-7000 is 25 MB/s [Xil21b], for this reason, tests have been performed up to this value.

Finally, the synchronisation of the system has been checked by tracing the interrupt signals and synchronisation and I/O events. It also is checked that the interrupt periodicity is kept, as well as when an interrupt is signalled, an I/O event is generated from Linux.

5.2.2.2. Results

The description of the power grids monitoring system can be seen in Figure 5.6 and is based on a mixed hardware/software solution. The hardware/software design is composed of an embedded Linux (dual-core) that runs an application to obtain data about the power grid state. It means the application accesses the FPGA or the hardware when it receives an interrupt (IRQ) from the FPGA. This interrupt indicates that the data obtained by the hardware is ready to be read.

The hardware design is made up of two hardware accelerators described at RTL level and presents a middle-grade complexity. The first accelerator is a signal acquisition module (*IP-ADC*). The *IP-ADC* provides the voltage and current sensors of the power grid. The second hardware accelerator is called *IP-PLL* and receives the sensor data from *IP-ADC*. Then, it calculates the frequency, phase, alpha-beta transformation, and Direct-quadrature-zero transformation of the power grid. The *IP-PLL* uses a Sequence Detector based on SOGI (Second-Order Generalized Integrator), plus a Phase-Locked Loop (PLL) with Backward integration. It has been implemented using High-level synthesis tools, as described in previous works [SMB+13]. When the PLL finishes performing its computation, it saves the results in a dual-port Block Random Access Memory (BRAM) memory. Then, it generates the interrupt to the Linux application, which indicates that the data can be read. Once Linux reads the power grid data from BRAM memory located in hardware, it sends them via MODBUS (Ethernet) to an external supervisor. The FPGA clock frequency is 100 MHz, and the hardware ADC and PLL accelerators run periodically, generating an interrupt every 100 μs. Both hardware accelerators consume 9.54% of Slices, 14.55% of DSPs, and 1.78% of BRAMs memories from the FPGA (Zynq-7000).

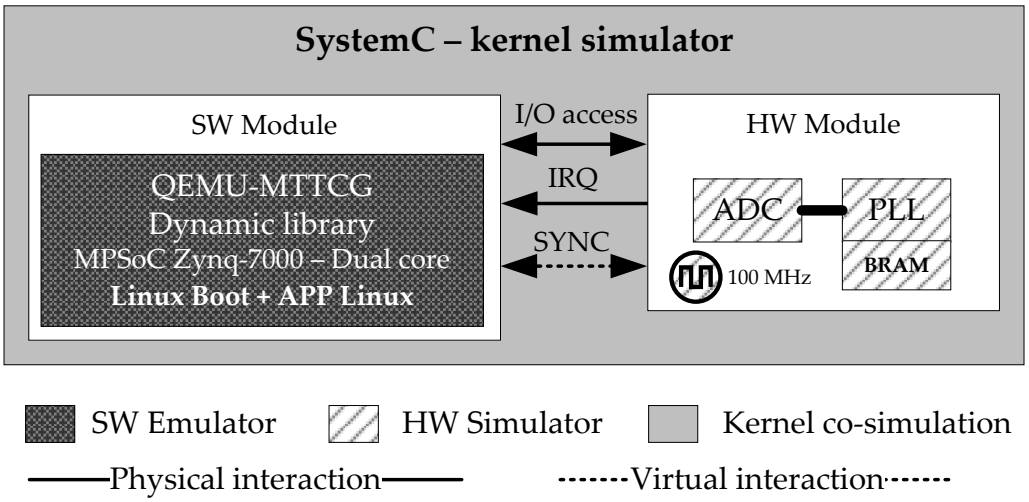

Figure 5.6: Linux + HW setup. QEMU-MTTCG inside a co-simulation virtual platform as software simulator. ADC and PLL IP accelerators in hardware simulator.

The results of multiple tests are shown in Figure 5.7a and, as it can be observed, even though the input/output rate is increasing, there is no appreciable change in the wallclock or running time of the co-simulation. This is because the workload of the hardware simulator and QEMU is higher than the virtual and physical interactions, even though it used the highest input/output rate supported by the real platform.

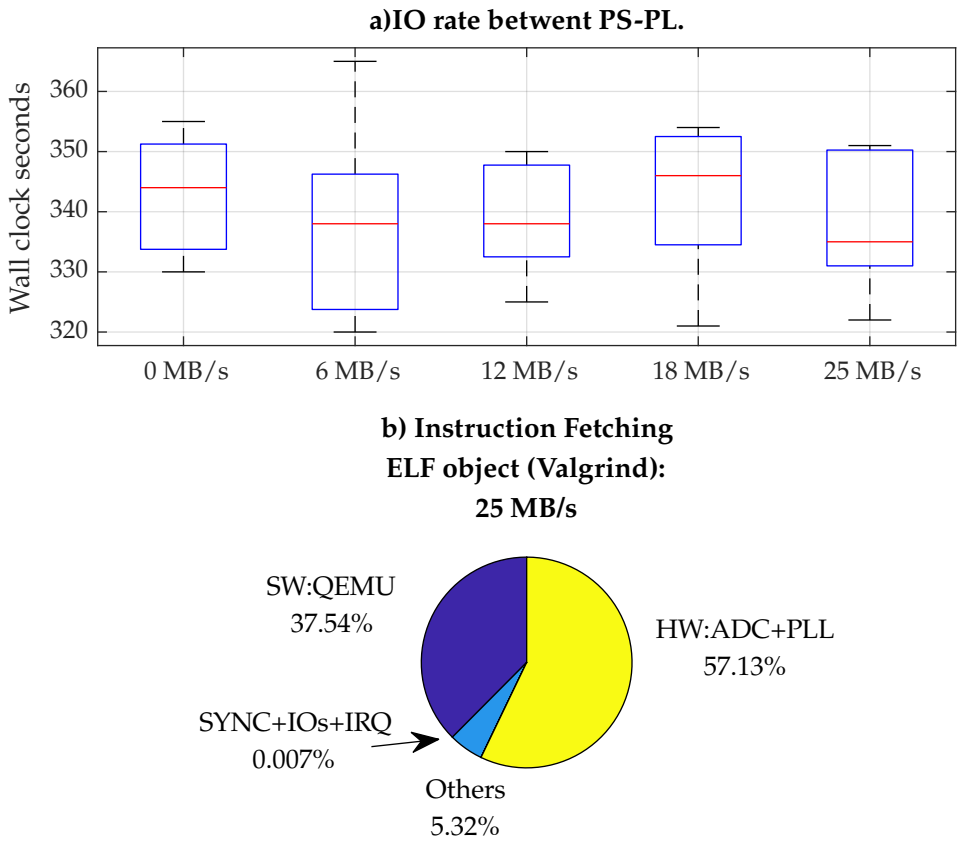

Figure 5.7: Linux + ADC + PLL micro-grid monitoring system results: (a) time consumed to perform the co-simulation (wallclock) for different I/O rates in PS-PL; (b) Instruction Fetching profiling for 25 MB/s I/O rate.

To check the overhead of the input/output accesses and interrupts on the host, profiler tests have been used again. Taking the highest I/O rate (25 MB/s), Figure 5.7b shows that most of the host instructions, which are executed by the virtual platform, are consumed by the hardware simulator.

While, despite having a high in-put/output rate (25 MB/s), only 0.007% of instructions are dedicated to the management of the input/output/interrupt and the synchronization between the software emulator and the hardware simulator. This shows that the synchronization mechanism included in QEMU-MTTCG does not add any appreciable workload for physical interactions compared to the workload of a real hardware/software co-simulation.

Figure 5.8 shows a time frame of the co-simulation results with the input/output events (physical interaction) and synchronization (virtual interaction) events. These are generated in each periodic interruption (100 μs). One read access per 100 μs is executed, which means a 32-bit access per interrupt (40 KB/s). The triangle represents the time instant when an event is generated. The IO_EVENT signal indicates the start of an input/output access event, and the SYNC_EVENT signal indicates a synchronization event between the software emulator and the hardware simulator. It can be seen that for each interrupt, Linux accesses the hardware through an input/output access. Therefore, Linux detects the interruption, wakes up the Application, and executes read access to hardware. The time from when the interrupt is triggered until Linux accesses the hardware is the Linux response time. The large number of synchronization events that are seen allows keeping the software emulator and hardware simulator in synchronised with a timing precision of 1 μs.

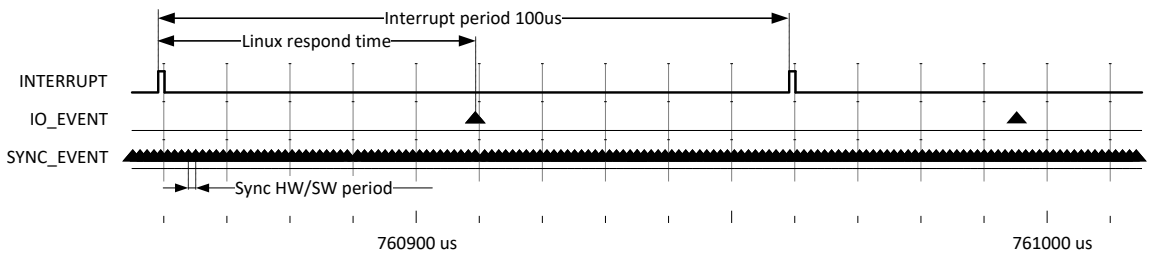

Figure 5.8: Co-simulation frame showing physical and virtual interactions. Example of the number of IO and synchronization events for each interrupt at 40 KB/s IO rate.

5.2.2.3. Conclusions

A quantitative analysis of the proposal and real tests have been carried out to analyse the possible overhead in co-simulation. These experiments test the impact of virtual and physical interactions between software emulator and hardware simulator for multiple setups. It has used a SMP Linux Boot study case, different typical software applications running under a SMP Linux platform and an FPGA-Linux co-simulation of a real industrial application.

The results shown in this section verify that the virtual interactions or synchronisation points do not add any overhead to a Linux boot running in the QEMU-MTTCG mode. In this case, it only increases the execution time by 5× when compared with the real platform. Furthermore, the modification applied in our solution does not introduce a substantial overhead (less than 10% and %5 on average) when running ParMiBench applications. This should be considered acceptable for power electronics applications where the simulation duration is typically around minutes.

The introduced external synchronisation mechanism allows obtaining the virtual CPU that has executed the most instructions, taking this time as the software timing reference. Moreover, the value of the Th_t parameter by itself does not influence the host simulation time or wallclock time. Therefore, *icountMax* parameter provides the user with a handy mechanism to define the timing precision of the software emulator.

The results also show that physical interactions do not add any appreciable workload in co-simulation. The workload of the HW modules is so high that the overhead of physical interactions is not appreciable.

The results show that the new proposal allows using parallelized-QEMU in hardware/-software co-simulation virtual platforms without any substantial overhead. It also allows taking advantage of the parallelisation of the software emulator to reduce the execution time of the co-simulation in real multi-core projects.

5.3. COSIL tests and performance analysis

Usually, the errors mainly happen during the code writing phase or the integration of different modules that make up the whole control system. The COSIL tool has been essential to verify all these functionalities and interactions with the control algorithms. Also, it has considerably shortened the project start-up times, making it possible to find possible errors quickly and without risk.

In order to evaluate COSIL in a work-in-progress industrial project, the validation of a control system of a power electronics converter has been considered. The control system includes control algorithms, hardware modules and interactions between the control board and the converter. As previously indicated, this is a 400 kVA back-to-back converter controlled by a Zynq-7000 SoC+FPGA (dual-core Cortex A9+Artix-7) based platform. This converter acts as a grid emulator to test photovoltaic converters. Therefore, it must work with different power setups, such as grid-forming, grid-following, and active/reactive power source/load. Different faults in the grid, like balanced and unbalanced voltage dips, must be tested in the grid-forming operating mode.

5.3.1. Power plant: back-to-back converter

Figure 5.9 shows a picture of the converter, and Figure 5.10 shows the electrical schematic of the converter. This converter has two Voltage Source Converter (VSC). VSC1 works as an active rectifier for this specific application, while VSC2 operates as an inverter. Depending on the configuration, it will behave as a voltage source or as a current source. VSC1 is connected to the grid via an L-filter. VSC2 has an LCL filter. The second L corresponds to the leakage inductance of the isolation transformer connected to the output. The Device Unter Test (DUT) is connected to the output of VSC2 through the insulated transformer.

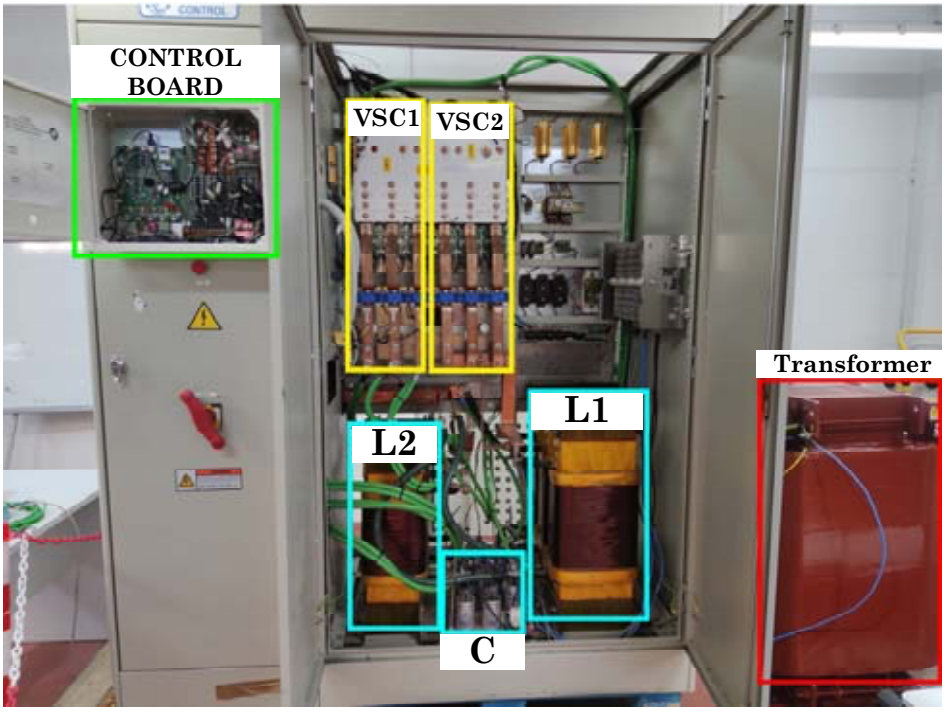

Figure 5.9: Back-to-back converter 400KVA.

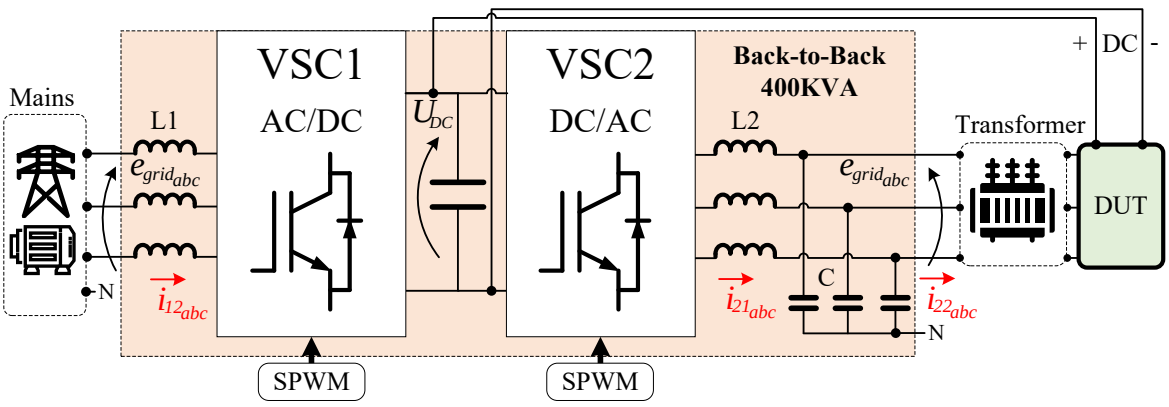

Figure 5.10: Electric scheme of the back-to-back converter.

5.3.2. Control algorithm

The control algorithm has been developed and verified in model-based simulation jointly with the power plant. The VSC1 uses a classical control of an active rectifier. It is based on two external loops for DC-bus voltage and reactive power control, and the internal current control loops have been implemented on dq-axes. Therefore, the VSC1 control maintains the setpoints defined by the DC bus voltage and reactive power. As a result, the PWM (Pulse-Width Modulation) references of VSC1 are obtained.

Concerning VSC2, the control algorithm is determined by its operation as a grid emulator and the disturbances generated to the Device Under Test (DUT). Focusing on the voltage-source operation, different types of control algorithms have been evaluated: cascaded controllers (voltage controller + current controller), voltage-only controllers, alpha-beta axis controllers, dq-axis controllers. COSIL has been a crucial tool to verify the control solutions before testing the converter. As a consequence, the experimental tests are speedy and safe.

5.3.3. HW/SW architecture of control system

The HW/SW architecture designed and implemented is shown in Figure 5.11. An asymmetric multiprocessor architecture solution (based on OpenAMP) has been used for the software. In OpenAMP architecture, the CPU#1 executes the control algorithm, and the CPU#0 runs an Embedded Linux, which offers all the communication and control services to the user.

In the HW (FPGA), the following modules have been implemented:

- Input/output interface of the converter.
- The whole acquisition stage (Figure 5.11 - ADC).
- Generation of switching frequency of the IGBTs and control frequency using a periodic interrupt (Figure 5.11 - IRQ). The possibility of reconfiguring the control sampling frequency and all associated frequencies is a desirable feature in a grid emulator converter to which a wide variety of DUTs can be connected.
- PLL (Phase-Locked Loop) to obtain the power grid phase and frequency. VSC1 and VSC2 use the PLL. The PLL is implemented in the FPGA because it allows to reduce the workload in software (Figure 5.11 - PLL).
- PWM generation. Different modulation techniques have been implemented. First, a modulation technique for DNPC (Diode Neutral Point Clamped) converters with zero sequence injection have been programmed as a default option. Subsequently, ANPC (Active Neutral Point Clamped) techniques have been developed. The ANPC modulation technique optimises IGBT losses. The modulation technique is selected

according to the working point, displacement power factor and amplitude modulation index.

- Control of faults produced by overvoltage or short circuit in the IGBTs (Figure 5.11 - Fault).

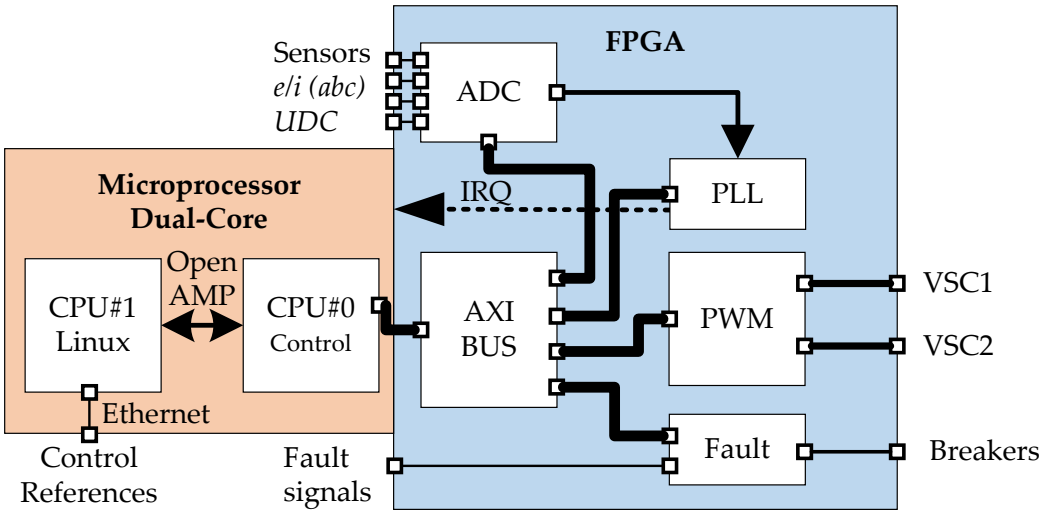

Figure 5.11: HW/SW architecture of implemented control system.

5.3.4. Tests analysis

In order to verify the implemented HW/SW architecture, a co-simulation of three seconds of the converter is carried out. The tests are performed using a full degree-of-freedom (FDOF) controller (voltage-only controller) [GGS01].

Figure 5.12 shows the COSIL platform results for the co-simulation of three seconds. In the first second, the DC bus voltage is raised to 900V. Then, 230V RMS - 50 Hz is generated at the output of VSC2. From the instant (1.5s) and for 500 ms, a three-phase voltage dip of 50% is performed without a ramp. This test has been repeated in the MIL simulation (Power Plant+Control), COSIL (SW+HW+PW), and the back-to-back converter. The source code of software and hardware used in COSIL and the real tests are the same. The acquisition functionality of the control board has been used to read the real signals.

As it can be seen, the implemented source code follows the control reference of VSC2 voltage. Figure 5.13 shows a zoom of the instant at which the gap is realised in MIL, COSIL and real converter. Note that there are some differences between the simulations and the actual results. These differences are due to the deviations between the plant modelled in MIL and the real plant. In addition, the acquisition frequency in COSIL and MIL is 200 kHz, while it is 80 kHz for the real power plant. This is an example of the high accuracy of COSIL, which can verify the code functionality more accurately than

even the real system. The reduction in frequency means that the real results do not show the high-frequency ripples in the signal peaks observed in the MIL and COSIL case.

The differences between MIL and COSIL are mainly due to quantisation and discretisation errors. Matlab works with double-precision floating-point by default. In contrast, the implementation of software and hardware in the control board works with single-precision floating-point. Furthermore, the power plant, which in MIL is simulated as a continuous element, is discretised in COSIL. Despite these errors, it can be seen that the results are practically similar, which demonstrates that the implementation of the software and hardware is correct. Then, COSIL allows verifying exactly the final HW/SW implementation of the multi-core SoC+FPGA boards without using HIL systems.

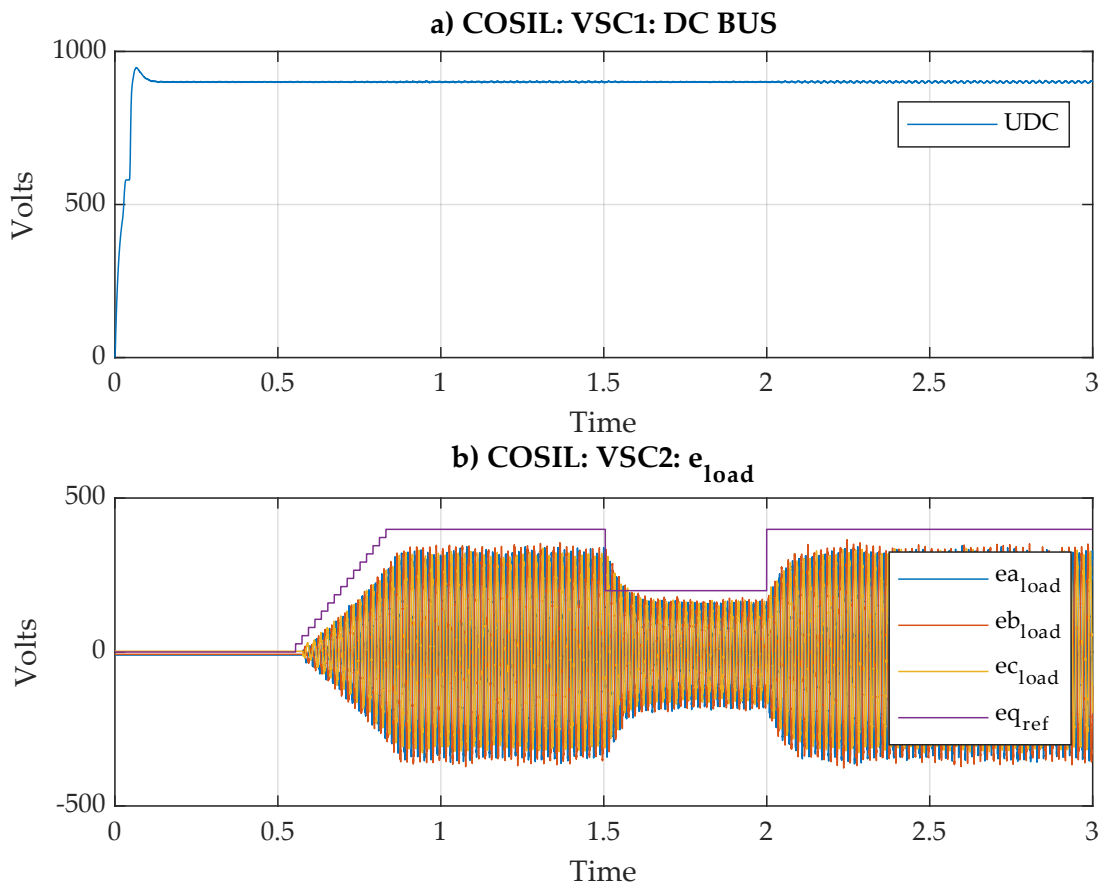

Figure 5.12: COSIL results. Start phase and voltage gap of 50%.

Figure 5.14 shows the results obtained directly in QuestaSim during the start-up of the converter. Here, it can be observed the status of the power plant and if the system (SW+HW) can control the converter successfully. The showed waveforms are:

- *contactor*. Contactor status. The contactor must open when the DC bus has been preloaded. When it closes, it connects the converter directly to the power grid or mains.

- *ea grid*. The voltage level of the grid (230 Vrms). Phase A. Here it can be observed that the PW module is correctly generating the electrical grid to which the device under test (DUT) will be connected.
- *current i12a*. Input current to VSC1. Initially, the DC bus preload can be observed. In order to increase the speed of the DC bus precharge, the value of the resistors of the precharge circuit is reduced. Once the precharge is done, it waits for a safe time and starts switching the IGBTs of VSC1 to continue raising the DC voltage level. At that point, it can be observed the current required to maintain the DC bus.
- *UDC BUS*. DC voltage level of the bus. It is shown how the precharge is performed. The precharge value raises the bus to 565 VDC. Later, when the modulation is enabled in VSC1, it can be seen the DC bus follows the control references reaching 950VDC in 50VDC jumps.
- *UDC REF* and *ea load REF*. Control references for the desired DC and AC voltage values.
- *ea load*. AC output voltage of VSC2. The output level is 0 until the SW decides to enable VSC2 switching. The *ea load REF* reference introduces a jump from 0V to 230Vrms. The aim is to check in the simulation that the programmed control algorithm can achieve an immediate step from 0 to 230Vrms.
- *current i21a*. Output current before the VSC2 filter. Here we can see how a purely resistive load of 20 kW is supplied. The current starts to be generated when the voltage level is present. That is when it begins to switch the IGBTs of VSC2.

The results in Figure 5.14 demonstrate the SW code and all the IPs to be implemented on the FPGA (ADC, PWM, PLL, Faults) work properly.

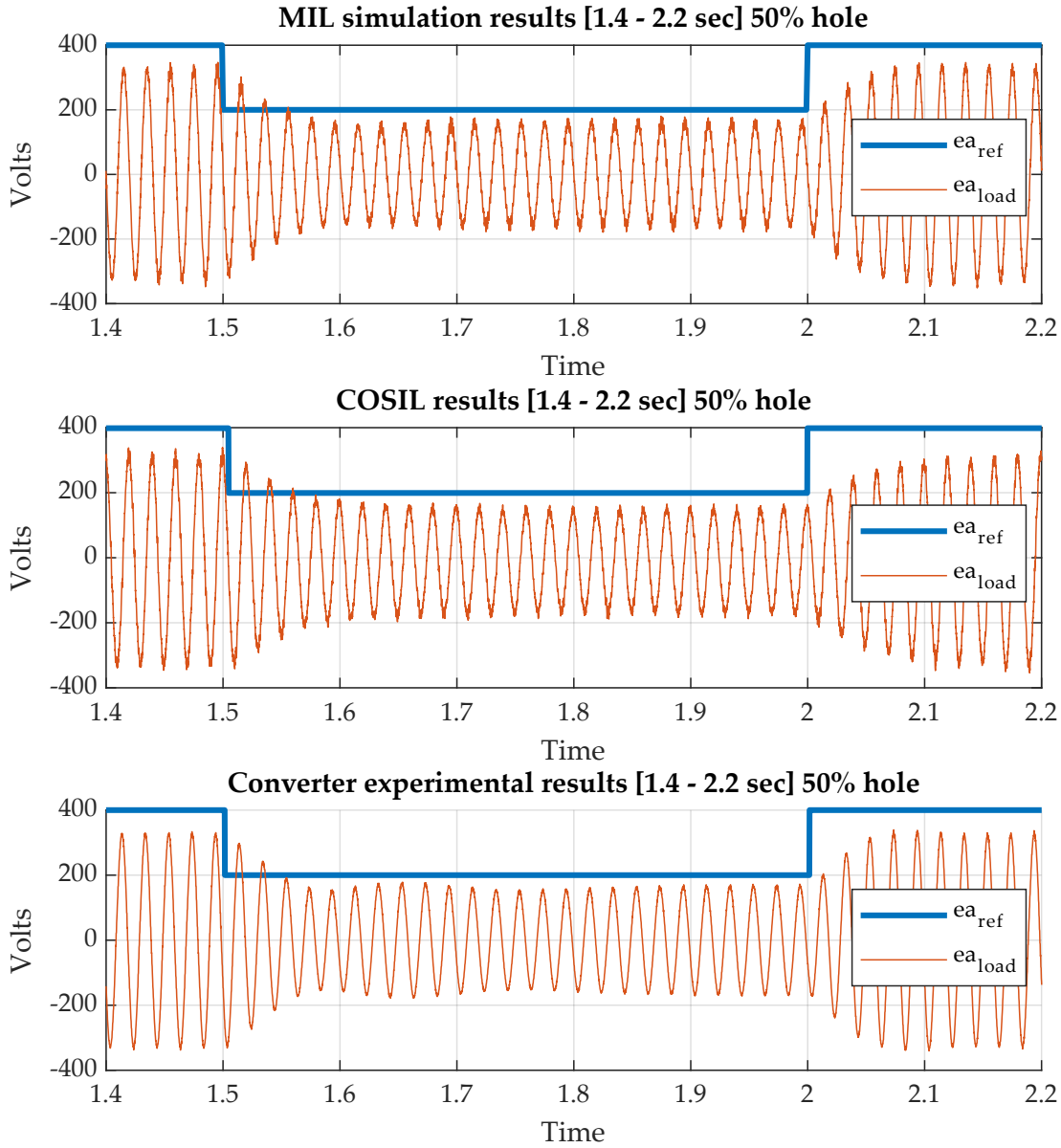

Figure 5.13: Results comparison. Model-based simulation (MIL - Simulink) versus COSIL (SW+HW+PW) versus real test. Zoom at the gap of 50%.

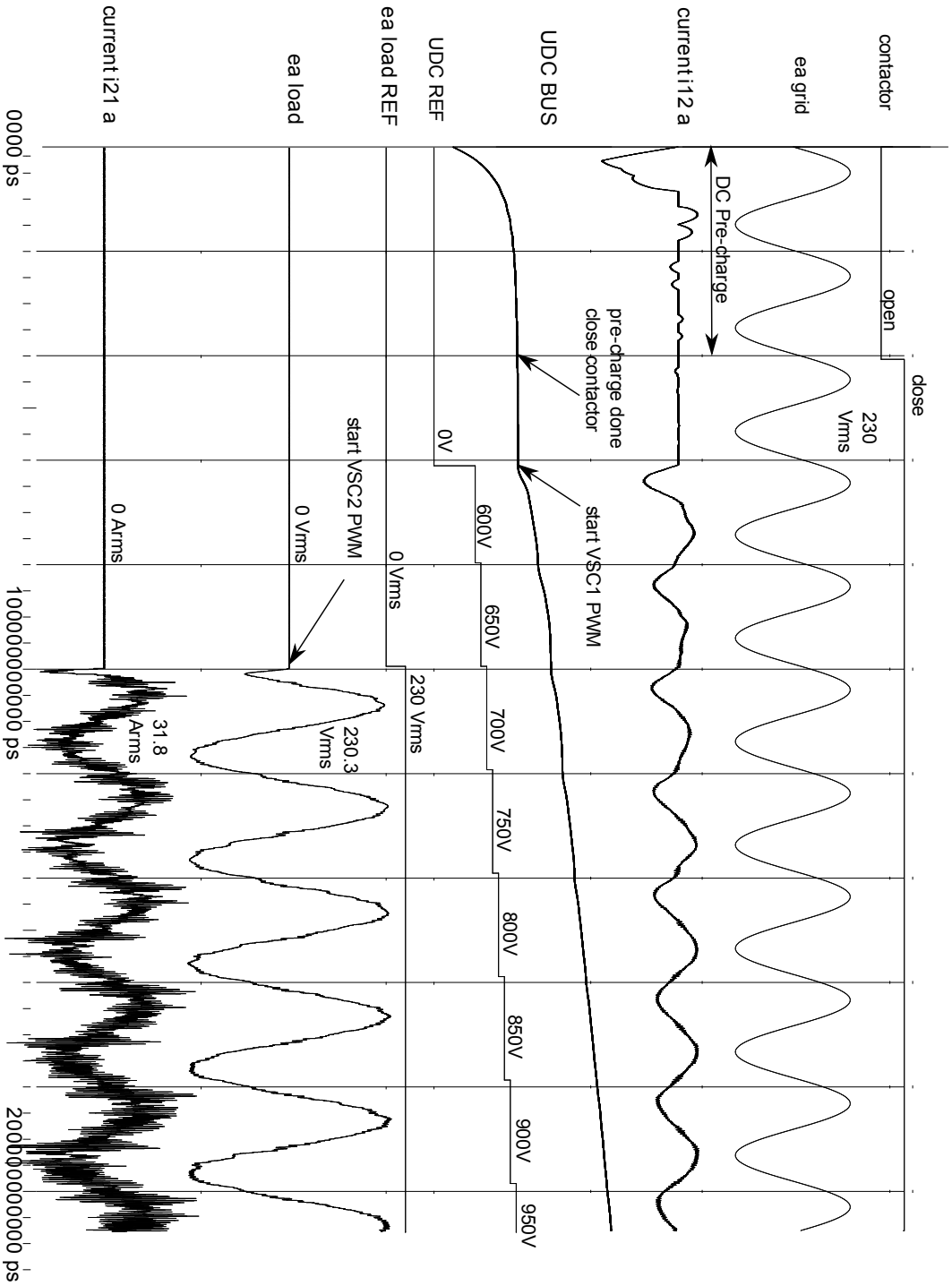

Figure 5.14: Co-simulation results. Back-to-back converter start-up and PW module status.

5.3.5. COSIL performance and the use of abstraction levels

The performance of a verification tool is vital to its usability. The less time the tools take to obtain the results of the simulations, the less time is consumed in the verification phase of the project. In this section, a performance study of the COSIL platform using the control system of the converter at different levels of abstraction has been carried out. To analyse the impact on the co-simulation performance of each SW, HW and PW module, the abstraction level of each module will be changed. Therefore, the co-simulation execution time for each configuration will be analysed. Furthermore, the results will be compared with the execution times that would be necessary to simulate each module independently using other reference tools.

In COSIL, the power plant is simulated using the equivalent discretised SystemC(C/C++) model. In order to focus on the performance and workload of software and hardware, all the following tests will run the same SystemC PW module.

The software has been verified using three configurations. The first one is called SW-SystemC. This configuration does not verify the same code used for the final implementation. It uses compiled code for the host processor, including the compiled code in a SystemC process. Therefore, it uses a functional equivalent version. However, this option is optimal for fast verification of the hardware on the FPGA, reducing the workload introduced by the software emulator (QEMU). The second configuration uses QEMU and emulates only a CPU running the control algorithm and the interaction with the hardware. Finally, the last option has included a multi-threaded emulation verifying an asymmetric OpenAMP (Linux-Baremetal dual-core) system.

From the hardware point of view, the IPs or hardware accelerators, shown in Figure 5.11, have been modelled in SystemC, working at the TLM abstraction level. As an example of mixing abstraction levels, simulations have been carried out mixing the ADC and PLL IPs modelled in SystemC (TLM level) with the PWM and Fault IPs modelled in VHDL (RTL level).

Finally, the time spent simulating three seconds with the MIL simulation has been analysed. In addition, a simulation of only the PWM IP has been performed in VHDL to check the time it takes to perform a three-second simulation at the RTL level.

In all the following tests, three seconds of simulation has been performed. The software has an timing resolution of 1000 instructions executed per core (dual-core). The timing resolution on the hardware (FPGA) is cycle-accuracy (10 ns), and the timing resolution of the power plant (PW) is the used discretized frequency (50 us).

Table 5.2 shows the different options and abstraction levels that have been simulated and the simulation times consumed. It can be seen that the COSIL platform allows simulating the behaviour of the whole system (HW+SW+PW) using mixing different levels of abstraction. In these tests, times in the range 1.17 minutes (even faster than

the MIL simulation with Simulink) to 108.22 minutes have been obtained (simulating an OpenAMP Linux+Baremetal system with IPs in SystemC and VHDL and the power plant discretised in SystemC). This range of times is proof that the platform can be adapted to users' needs for abstraction and debugging. Note that just simulating 3 seconds in VHDL of the PWM IP takes 73.8 minutes, while on our platform, we can co-simulate all software and hardware (CPU+FPGA) plus the power plant in just 10 minutes more. It is an example of the optimised performance of the platform.

Table 5.2: COSIL time results using different abstraction levels for the same design

SW	HW	PW	Time	Abstraction level
–	–	MIL/Simulink	4.35 min	●●●●●
–	Only PWM in VHDL	–	73.8 min	○○○○○
SW-SystemC	SystemC/C++	SystemC/C++	1.17 min	●●●●○
QEMU(single-core) baremetal	SystemC/C++	SystemC/C++	11.65 min	●●○○○
QEMU(single-core) baremetal	ASC+ PLL* in SystemC PWM+fault in VHDL	SystemC/C++	81.29 min	●○○○○
QEMU(dual-core) OpenAMP	SystemC/C++	SystemC/C++	34.32 min **	●●○○○
EMU(dual-core) OpenAMP	ASC+ PLL* in SystemC PWM+fault in VHDL	SystemC/C++	108.32 min	●○○○○

*PLL in SystemC exported from Vivado HLS; ** The boot Linux time has not been take into account.

5.3.6. Synchronization results

Figure 5.15 shows the synchronisation results obtained with QuestaSim. The events generated when any communication between QEMU and SystemC takes place are shown here. The bottom figure is a zoom of a time interval in the upper figure. In this example, it has chosen the minor period between the HW and PW modules as Quantum value. Then, the value of Quantum is equal to the discretisation period of the power plant (1 μs).

The figure above shows that the I/O accesses from QEMU are performed when an interrupt is detected. This interrupt is triggered every 16 acquisitions runs of the ADC IP. In this Figure, it also can be seen the differences in the duration of the interrupts. The differences in the duration of the high level interrupts dependent on the time it takes QEMU to handle the interrupt. The period $T_{control}$ between each rising edge of the interrupt is what sets the control frequency. This control frequency is used by the control algorithm (bare-metal). When the bare-metal executes the interrupt, it accesses the HW to turn off the interruption source.

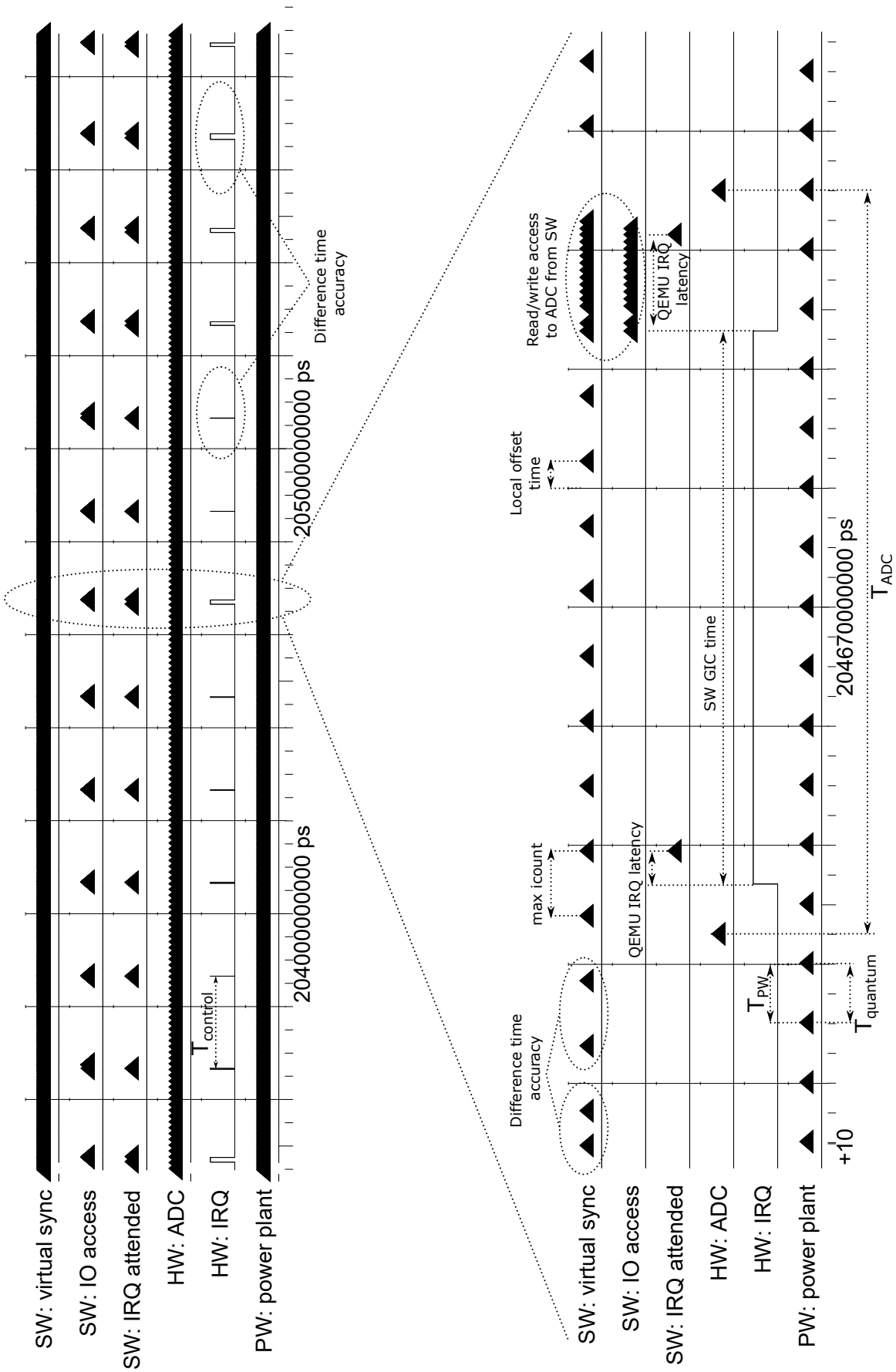

Figure 5.15: Synchronization results. Communication events between SW, HW and PW modules.

In the Figure above, the synchronisation between the SW, HW and PW modules can be seen in more detail. Some comments are described below:

- The icountMax parameter sets the maximum time that can be spent to send a virtual interaction. This maximum time depends on the configured K-CPI value.
- It can also be noticed that virtual interactions are not performed periodically. They depend on the execution of QEMU.
- When the SW module exceeds the Quantum barriers (which coincide with the PW events), it waits until the rest of the modules have advanced the same amount of time as the SW module. The difference between the time that it has to wait and the time of the passed Quantum barrier is called Local offset time in the Figure.
- When an interrupt is triggered, QEMU takes a time (latency) to notify the HW that it has read it. It then sends the interrupt to the CPU's interrupt controller so that it can be addressed. The QEMU latency is the sum of the time it takes for QEMU to handle an interrupt level change and the time it takes for the emulated CPU's interrupt controller to process the interrupt. Each time the value of the interrupt signal changes, QEMU notifies SystemC when it has updated the level change.
- When the CPU handler detects the interrupt, it triggers access to the ADC to read the sensor data.

These results show that timing errors between simulation and module times are minimal when chosen an appropriate quantum time. Furthermore, it is observed that the system is correctly synchronised by executing an interrupt every 16*TADC and reading the ADC data from the bare-metal application.

5.3.6.1. Conclusions

This section presents a SIL-based verification tool called COSIL as an alternative to the HIL and PIL/FIL systems. Its functionality and performance have been tested in power electronics applications, although its use could be extended to any generic applications in general. This tool is based on a co-simulation virtual platform that links the hardware and software modules with the power plant, closing the control loop in the same workstation.

The presented tool allows verifying exactly the final HW/SW implementation and integration of the multi-core SoC+FPGA without using a HIL system. Thus, it helps to reduce the errors that happen during the code writing phase and integration phase. This new SIL-based alternative reduces the verification costs and supports more debugging and abstraction capabilities in the simulation.

The results show that the functionality of the system is the same as the MIL simulation and corresponds to the real results of the equipment under test. It has been demonstrated

that COSIL, working in a simulated environment, allows even more numerical and temporal accuracy than the real platform.

Performance tests also show that COSIL allows working at multiple abstraction levels depending on the user's needs. For a real application, the execution times range from minutes to hundreds of minutes. It has again been verified that the HW modules workload is higher than that of the other modules. This suggests that more effort is needed to parallelise the HW simulator to increase the co-simulation speed.

Finally, the synchronisation results show that the timing error in synchronisation can be controlled by the Quantum time and *icountMax* parameters. It is found that the errors between simulation and module times are minimal if the appropriate Quantum value is chosen.

Although the performance of co-simulation using multi-threaded emulators has been improved, SIL-based solutions, like the COSIL tool, can not currently achieve the real-time execution of HIL systems for complex designs. However, it is a significantly cheaper solution competing in performance and debugging features with MIL simulations.

Chapter 6

Conclusions and future work

Software-in-the-loop could democratise the verification of critical systems.

The main goal of this thesis is to improve the verification process of control systems in power applications. However, its use could be extended to control applications in general.

In order to improve it, a study of the current verification alternatives for control systems has been carried out, and the most advantageous option has been chosen. Then, it presents the verification SIL-based tool called COSIL as an alternative to the HIL and PIL/FIL systems. COSIL is an open-source co-simulation virtual platform that links software emulators with hardware and power simulators. Thus it can close the control loop in the same workstation.

Thanks to COSIL, it is possible to verify software/hardware designs implemented in MPSoC+FPGA devices. Multi-core operating systems and bare-metal applications interacting with hardware designs on the FPGA can be verified. An advantage is that it allows using multiple levels of abstraction and adjusting the complexity and performance of the co-simulation to the user's needs.

The execution speed of the verification tool has been a priority, so multi-core emulators (QEMU) have been used, and a synchronisation mechanism between the QEMU and an external hardware simulator has been provided. The mechanism gets its timing information from the number of instructions executed by each CPU and decides when to send the information to the external hardware. Results show that the proposed mechanism does not have an impact on the performance of the QEMU.

To test the proposed solution, the implementation of a control system for a 400 kVA back-to-back converter has been simulated using COSIL. This converter operates as a grid emulator for testing photovoltaic converters. The implementation has been based on a HW/SW architecture where the FPGA designs have been mixed with multi-core OpenAMP Linux-Baremetal systems.

To conclude, the results demonstrate that COSIL is a useful tool for verifying critical control systems. This new SIL-based alternative also reduces the verification costs and supports more debugging and abstraction capabilities than HIL systems. The contributions aim to address the main limitations of SIL and offer greater code re-usability and simulation speed, repositioning SIL as a competitive option to HIL systems and competing in performance with MIL simulations.

6.1. Thesis conclusions

Co-simulation techniques

Co-simulation is not a novel concept. In fact, some authors have been working with it for decades. With the rise of heterogeneous architectures and the mixing of software/hardware designs, its use is becoming more common. It is essentially based on synchronising specialised simulators or emulators to build a virtual platform that behaves like the real platform.

A co-simulation tool has two main features: the synchronisation between the simulators or emulators and the levels of abstraction it supports.

The wide range of simulators has resulted in a whole theory for managing the synchronisation of simulators. There are solutions for synchronising simulations distributed across multiple machines, parallel or sequential. This thesis has been based on using a conservative asynchronous sequential co-simulation. This option is a trade-off between speed and timing accuracy, allowing each simulator to advance freely until specific instants in which everything must be synchronised. These instants are repeated periodically. This solution is optimal for power applications, as it usually runs periodically the same control algorithm.

The choice of a sequential co-simulation is a consequence of the base language used to develop the co-simulation platform. This language is SystemC (C++). SystemC includes a simulation kernel based on DES (Discrete Event Simulator). Although it allows the development of hardware simulators with an open-source license, its main limitation is execution speed. It was intended to simulate concurrent designs, but the execution of its kernel is sequential. Although some authors have proposed promising ways to parallelise the SystemC kernel, these have not yet been standardised.

Using SystemC also allows working at practically all levels of abstraction of the design/verification methodology in digital devices. This makes the verification process more manageable and adapts the debugging requirements, speed and timing accuracy to the user's needs.

The timing errors in the synchronisation must always be limited. In our case, conservative asynchronous synchronisation allows modifying the frequency with which the

simulators are updated. This option is helpful to adapt the timing accuracy.

The proposed software/hardware/power co-simulation solution is based on synchronising an emulator with a simulator: QEMU (Quick Emulator) with SystemC-DES. With SystemC-DES simulator, it can simulate hardware and power modules. To guarantee synchronisation between QEMU and SystemC-DES, it is necessary to implement a SystemC module that advances according to the QEMU's clock. This advance is limited and is blocking so that the other SystemC modules can advance sequentially. The link between QEMU and SystemC-DES allows SystemC and QEMU to be stopped at the same time. This supports the step-by-step debugging of all modules.

In this solution, the timing errors of synchronisation between QEMU and SystemC-DES depend on the maximum time allowed for each module to advance. Although this time is customisable, it is constant for the duration of the co-simulation. Typically the advance limit has been estimated for each design as it depends on the specifications of each design. An attractive approach to reduce timing errors is to change this limit dynamically depending on the number of executed events.

Software emulators

Emulation has become one of the most popular techniques used by engineers to verify software. Not only does it allow testing the software functionality almost in real-time, independently of the architecture, but it also enables the emulation of peripherals and everything attached to the microprocessor. Another of its significant features is that it allows verifying precisely the same code that runs later on the control board without any modification.

Nowadays, there are many emulators available. The most famous is QEMU, perhaps because it is open-source, and many companies have picked it up. QEMU is based on an instruction-by-instruction translation of the guest code to run on a host. Its efficiency is based on reusing the instruction blocks previously translated, so it can skip the translation step and execute the instructions directly.

The parallelisation of QEMU's TCG translation kernel has been a performance improvement. The considerable growth of QEMU in the last five years has made it a very mature project. However, the complexity of its source code makes it a challenge to add features. Perhaps the code distribution should be re-evaluated, and it may be simplified its internal architecture to continue growing at a sustainable rate.

The most complex aspect of QEMU is its timing. Currently is not available the timing models of all microprocessors, peripherals and memories across the market. This is a drawback to include the software emulator in a co-simulation environment. The absence of timing models of microprocessors has forced the search for original ways to obtain their timing notion. Today it is still a field in need of growth.

Currently, counting the number of instructions executed by each CPU provides sufficient timing accuracy to synchronise QEMU with an external stimulator. However, the temporal modelling of the memories devices (like caches) could provide more resolution with a minimum penalty in performance.

Verification in power electronics

In power electronics applications, power plants are composed of three parts: power equipment (capacitors, inductors, diodes, IGBTs, motors...), sensors and protections (contactors, relays, wiring) and a control system (control board). The control system is the brain of the equipment and is becoming increasingly intelligent. Highly sophisticated control algorithms are programmed in it.

More and more, control systems are embracing advances in other areas such as artificial intelligence or telecommunications. An example is the trend to use MPSoCs+FPGAs to develop the control algorithms and the equipment monitoring.

The verification of digital MPSoCs and FPGAs is a complex and time-consuming process. The current trend to include heterogeneous chip architectures has made the verification cycle even more complex and time-consuming. The current solution for manufacturers is to simplify the design process by adding synthesisers. These synthesisers, such as HLS, generate HDL code (VHDL, Verilog, SystemC) from C code. Thus, the use of typical HDL digital design languages will gradually decrease and be replaced by those typically used in the software.

Power equipment is costly and is called critical systems, i.e. it cannot fail. This is why their verification is so important. A failure can lead to an explosion or damage to staff. A solution is to use non-destructive equipment that emulates the behaviour of the plant and allows the control system to be tested. Such equipment is called Hardware-in-the-loop. However, the high features of Hardware-in-the-loop equipment make it expensive.

Software-in-the-loop could overcome the limitations of Hardware-in-the-loop. This movement from a hardware world to a software world has two implications. First, it increases verification times. It is difficult for a workstation to compete on speed with a HIL equipment. HILs equipments often use racks of FPGAs. The second is increased flexibility, debugging capabilities, reduced cost and reuse of the verified code.

Software-in-the-loop could provide low-cost tools to verify control systems efficiently and democratise the verification of critical systems. This thesis aims to provide the basis for this route.

In order for Software-in-the-loop to compete with Hardware-in-the-loop, it is necessary to mix three domains: software (CPU), hardware (FPGA) and real-world (power plant). Virtual co-simulation platforms are currently one of the potential solutions. These platforms allow to emulate software, simulate hardware designs and simulate plant models.

The challenge is to synchronise the three domains without compromising the performance.

The most commonly used option to synchronise them is based on choosing a soft synchronisation that allows fast co-simulations. A potential solution to the efficiency of this co-simulation is the use of the cloud. This could distribute, parallelise and increase the emulation speed at a relatively low cost.

6.2. Future work

A set of proposals for future work is shown below. Such future work attempts to address the main limitations found in COSIL and QEMU throughout this thesis, as well as to extend their features and introduce new branches of research.

Accelerate co-simulation speed

One of the disadvantages of COSIL is precisely its slow execution speed compared to real-time systems. There are many works that have offered some solutions to increase the speed of the simulation. In summary, three solutions have been proposed: 1. Parallelise the SystemC kernel [DLS17, Del17b]. 2. Use of dynamic synchronisation in co-simulation [Del17b]. 3. Use the high performance of cloud services to run distributed co-simulations.

The SystemC kernel can be used as a hardware simulator. The drawback is that the current open-source OSCI implementation is sequential, while the hardware modules to be verified are often parallel. Some useful works have managed to parallelise the SystemC kernel [DLS17, VPSH14, VS16]. Applying the contributions of these works to COSIL would be to go faster.

As has been discussed throughout this book, COSIL uses an asynchronous conservative co-simulation. Its implementation is based on the use of the Quantum time limit. At these time limits, all simulators must stop and synchronise. In the OSCI implementation of the SystemC kernel, the Quantum is static. Throughout a co-simulation, there may be times when there is much interaction between the simulators and high synchronisation accuracy is required, and times when there is minimal interaction between the simulators. In the latter case, the Quantum could be extended to avoid any penalties for unnecessary synchronisation. A more lax solution called dynamic Quantum is proposed in [Del17b]. In particular, this work proposes calculating the Quantum at the beginning of the simulation by saving the timestamps of the first synchronisation events. Although this is an promising solution, it would only be effective if the synchronisations were periodic. This is the case for most power electronics control applications. It would be interesting to evaluate how to modify Quantum to minimise the number of synchronisations in the co-simulation.

Lastly, the current trend of some companies like Amazon or Microsoft is to offer cloud services. This means that users do not have to spend money on a high-performance

workstation but only worry about having a stable internet connection. The software tools are executed on servers in the cloud, and the memory and computational capacity are increased.

This option is exciting as, in addition to increasing the overall performance of the co-simulation, it would allow it to be distributed by parallelising the simulators. Obviously, this parallel distribution would have to manage the synchronisation between the simulators correctly. By distributing or parallelising simulators, penalties for communication between simulators will be encountered. However, as seen throughout this thesis, for power electronics applications, the main workload is not provided from the communications or synchronisation between software/hardware/power domains; it is provided from the emulation of the CPUs and the FPGA modules simulation.

Improve timing accuracy

CPU time modelling is one of the most complex tasks for an emulator. In fact, not every emulator supports this feature.

The timing notion of QEMU from the number of executed instructions provides determinism and approximate timing of the software progress. However, it is not sufficient for detailed timing analysis of the software. Commonly, other simulators such as gem5 are used to achieve cycle accuracy. Multiple works have provided solutions to increase the accuracy of QEMU (see Section 2.3.2). All of them impact the performance of QEMU to some degree. The penalties appear when using helper functions to stop the translation and execution loop and add some functionality. How to reduce the impact of these helper functions could be an interesting work.

In particular, the paper [KYH16] explains how to model the CPU timing by modelling the instruction cache miss, data cache miss and branch misprediction. The authors claim that they can achieve cycle accuracy with an 8% error in estimation at the cost of a 30% penalty in emulation speed. It would be interesting to analyse how to optimise this modelling and apply it to the parallel mode of QEMU.

Develop a full open-source tool

Cost is a significant factor in the choice of using a particular software tool. This thesis has been trying to provide a solution that does not depend on proprietary tools, thus extending the use of COSIL. However, due to the limitations of the current open-source simulators, it was necessary to use tools such as Matlab or Questasim to develop specific modules and simulations of the platform.

Currently, the software emulator (QEMU) and the hardware simulator (OSCI SystemC kernel) are open-source. However, the hardware simulator only support hardware module written in the native language of its kernel, SystemC (C++). To verify mixed hardware

modules (e.g. VHDL+SystemC), it was necessary to use mixed simulators such as QuestaSim. It is expected that Vivado Simulator by Xilinx will support mixed simulations, and the reason is that many of their hardware models are starting to be described in SystemC. The current open-source mixed simulator performance evaluation is a pending task to try to offer open-source solutions for mixed simulations.

Although there are companies that have tried to generate C code from the plant models, the current performance of Simulink is hard to overcome. Its toolboxes have been on the market for years and allow the generation of C/C++ code from almost all its components. It would be useful to develop a C code library of basic discretised power electronics elements. This would facilitate the generation of C/C++ code for a complex plant and not depend on Simulink for that task.

Enable the use of QEMU multi-instance

As heterogeneous architectures use are on the rise, many MPSoCs include multiple processors of different models. The UltraScale family from Xilinx is a good example. QEMU can only emulate a CPU type per instance. To emulate UltraScale, two QEMU processes must be launched in the OS. Thus, to emulate both processors simultaneously, it is necessary to build an interface between the two QEMU instances [Xil19].

Multiple QEMU instances are not currently enabled in COSIL and are a desirable feature to emulate the latest devices on the market. The standardisation of this interface is vital to extend its use to all architectures and guest machines supported by QEMU. A critical part that must addressed is the synchronisation between multiple instances of QEMU.

Add more co-simulation features

This thesis provides the basis for the COSIL tool, but many desired features have not be implemented.

These include memory accesses from hardware to software. These accesses are typically done through Direct Memory Accesses (DMA). In the latest version of QEMU, an API has been included to manage DMA accesses. It is necessary to study its limitations and analyse how to access CPU memory from the hardware simulator.

Finally, it would be desirable to use typical software development platforms such as Eclipse to include breakpoints in the code. This would avoid modifying the source code with internal functions and would be more convenient to debug. The tracing of software variables throughout an emulation is an exciting option to analyse its evolution. Currently, it is available with COSIL by introducing sentences in the original source code. However, it should be possible to do this without having to modify the source code. A possible solution would be to create a memory address table where these variables are stored in

QEMU. This table should note the pointers to the cache memories in order to be able to analyse their progress. In this way, an engine could be created to monitor the evolution of the software variables within QEMU.

Bibliography

[AAKMK18] Anas Ahmad Abudaqa, Talal M. Al-Kharoubi, Muhamed F. Mudawar, and Armin Kobilica. Simulation of ARM and x86 microprocessors using in-order and out-of-order CPU models with Gem5 simulator. In *2018 5th International Conference on Electrical and Electronic Engineering (ICEEE)*, pages 317–322. IEEE, may 2018.

[ABG+16] Sagheer Ahmad, Vamsi Boppana, Ilya Ganusov, Vinod Kathail, Vidya Rajagopalan, and Ralph Wittig. A 16-nm Multiprocessing System-on-Chip Field-Programmable Gate Array Platform. *IEEE Micro*, 36(2):48–62, mar 2016.

[AKS16] Mohammad Alian, Daehoon Kim, and Nam Sung Kim. pd-gem5: Simulation Infrastructure for Parallel/Distributed Computer Systems. *IEEE Computer Architecture Letters*, 15(1):41–44, jan 2016.

[And05] Jason R. Andrews. *Co-verification of Hardware and Software for ARM SoC Design.* 2005.

[ARM] ARM. AMBA AXI and ACE Protocol Specification.

[Ayn09] J. Aynsley. *OSCI TLM-2.0 language reference manual.* Number July. 2009.

[BC14] Simone Buso and Tommaso Caldognetto. A non-linear wide bandwidth digital current controller for DC-DC and DC-AC converters. In *IECON Proceedings (Industrial Electronics Conference)*, pages 1090–1096. IEEE, oct 2014.

[BCR+08] Emilio J. Bueno, Santiago Cóbreces, Francisco J. Rodríguez, Álvaro Hernádez, and Felipe Espinosa. Design of a back-to-back NPC converter interface for wind turbines with squirrel-cage induction generator. *IEEE Transactions on Energy Conversion*, 23(3):932–945, sep 2008.

[BDBK10] David C. Black, Jack Donovan, Bill Bunton, and Anna Keist. *SystemC: From the Ground Up.* Springer US, Boston, MA, 2010.

[BDD19] Blaabjerg, Dragicevic, and Davari. Applications of Power Electronics. *Electronics*, 8(4):465, apr 2019.

[Bel05] Fabrice Bellard. QEMU, a Fast and Portable Dynamic Translator. *USENIX Annual Technical Conference. Proceedings of the 2005 Conference on*, pages 41–46, 2005.

[BFP06] Nicola Bombieri, Franco Fummi, and Graziano Pravadelli. On the evaluation of transactor-based verification for reusing TLM assertions and testbenches at RTL. In *Proceedings -Design, Automation and Test in Europe, DATE*, volume 1, pages 1–6. IEEE, 2006.

[BGL+15] David Broman, Lev Greenberg, Edward A. Leey, Michael Masin, Stavros Tripakis, and Michael Wetter. Requirements for hybrid cosimulation standards. In *Proceedings of the 18th International Conference on Hybrid Systems: Computation and Control, HSCC 2015*, pages 179–188, New York, NY, USA, apr 2015. ACM.

[BGOS12] Anastasiia Butko, Rafael Garibotti, Luciano Ost, and Gilles Sassatelli. Accuracy evaluation of GEM5 simulator system. In *7th International Workshop on Reconfigurable and Communication-Centric Systems-on-Chip (ReCoSoC)*, pages 1–7. IEEE, jul 2012.

[BHR+09] Emilio J. Bueno, Álvaro Hernández, Francisco J. Rodríguez, Carlo Girón, Raúl Mateos, and Santiago Cóbreces. A DSP- and FPGA-based industrial control with high-speed communication interfaces for grid converters applied to distributed power generation systems. *IEEE Transactions on Industrial Electronics*, 56(3):654–669, mar 2009.

[BIME13] Imen Bahri, Lahoucine Idkhajine, Eric Monmasson, and Mohamed El Amine Benkhelifa. Hardware/Software codesign guidelines for system on chip FPGA-based sensorless AC drive applications. *IEEE Transactions on Industrial Informatics*, 9(4):2165–2176, nov 2013.

[BMC16] Denis Becker, Matthieu Moy, and Jérôme Cornet. Parallel Simulation of Loosely Timed SystemC/TLM Programs: Challenges Raised by an Industrial Case Study. *Electronics*, 5(4):22, may 2016.

[BMV13] Andrea Benigni, Antonello Monti, and Ravinder Venugopal. Advancements and challenges of a multi-platform real time simulation lab for power applications. In *IECON Proceedings (Industrial Electronics Conference)*, pages 5358–5363. IEEE, nov 2013.

[Bos97] Bimal K. Bose. *Power Electonics and Variable Frecuency Drives: Technology and Applications*. First edition, 1997.

[BP07] Felice Balarin and Roberto Passerone. Specification, synthesis, and simulation of transactor processes. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 26(10):1749–1762, oct 2007.

[BSD+20] Andrea Benigni, Thomas Strasser, Giovanni De Carne, Marco Liserre, Marco Cupelli, and Antonello Monti. Real-Time Simulation-Based Testing of Modern Energy Systems: A Review and Discussion. *IEEE Industrial Electronics Magazine*, 14(2):28–39, jun 2020.

[BSS+11] Nathan Binkert, Somayeh Sardashti, Rathijit Sen, Korey Sewell, Muhammad Shoaib, Nilay Vaish, Mark D. Hill, David A. Wood, Bradford Beckmann, Gabriel Black, Steven K. Reinhardt, Ali Saidi, Arkaprava Basu, Joel Hestness, Derek R. Hower, and Tushar Krishna. The gem5 simulator. *ACM SIGARCH Computer Architecture News*, 39(2):1, aug 2011.

[BST92] D. Becker, R.K. Singh, and S.G. Tell. An engineering environment for hardware/software co-simulation. In *[1992] Proceedings 29th ACM/IEEE Design Automation Conference*, pages 129–134. IEEE Comput. Soc. Press, 1992.

[CBBC17] Emilio G. Cota, Paolo Bonzini, Alex Bennee, and Luca P. Carloni. Cross-ISA machine emulation for multicores. In *CGO 2017 - Proceedings of the 2017 International Symposium on Code Generation and Optimization*, pages 210–220. IEEE, feb 2017.

[CKCL18] I-Hua Chen, Chung-Ta King, Yao-Hua Chen, and Juin-Ming Lu. Full System Emulation of Embedded Heterogeneous Multicores Based on QEMU. In *2018 IEEE 24th International Conference on Parallel and Distributed Systems (ICPADS)*, pages 771–778. IEEE, dec 2018.

[CLP14] Filippo Cucchetto, Alessandro Lonardi, and Graziano Pravadelli. A common architecture for co-simulation of SystemC models in QEMU and OVP virtual platforms. In *2014 22nd International Conference on Very Large Scale Integration (VLSI-SoC)*, pages 1–6. IEEE, oct 2014.

[CNVT12] H. Camblong, S. Nourdine, I. Vechiu, and G. Tapia. Comparison of an island wind turbine collective and individual pitch LQG controllers designed to alleviate fatigue loads. *IET Renewable Power Generation*, 6(4):267, 2012.

[CSD20] Zhongqi Cheng, Tim Schmidt, and Rainer Dömer. SystemC coding guideline for faster out-of-order parallel discrete event simulation. In *Lecture Notes in Electrical Engineering*, volume 611, pages 99–114. 2020.

[CYT11] Ming Chao Chiang, Tse Chen Yeh, and Guo Fu Tseng. A QEMU and SystemC-based cycle-accurate ISS for performance estimation on SoC development. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 30(4):593–606, apr 2011.

[CZCV13] Chia-han Yang, Gulnara Zhabelova, Chen-Wei Yang, and Valeriy Vyatkin. Cosimulation Environment for Event-Driven Distributed Controls of Smart

Grid. *IEEE Transactions on Industrial Informatics*, 9(3):1423–1435, aug 2013.

[DBK+16] Guillaume Delbergue, Mark Burton, Frederic Konrad, Bertrand Le, Christophe Jego, Bertrand Le Gal, and Christophe Jego. QBox : an industrial solution for virtual platform simulation using QEMU and SystemC TLM-2 . 0, 2016.

[DCHC11] Jiun Hung Ding, Po Chun Chang, Wei Chung Hsu, and Yeh Ching Chung. PQEMU: A parallel system emulator based on QEMU. In *Proceedings of the International Conference on Parallel and Distributed Systems - ICPADS*, pages 276–283. IEEE, dec 2011.

[DDC+19] Marc Duraton, Koen De Bosschere, Bart Coppens, Christian Gamrat, and Madeleine Gray. *HiPEAC Vision*. 2019.

[Del17a] Guillaume Delbergue. *Contribution à l'amélioration des plateformes virtuelles SystemC/TLM : configuration, communication et parallélisme*. Theses, Université de Bordeaux, 2017.

[Del17b] Guilleaume Delbergue. *Advances in SystemC/TLM Virtual Platforms: Configuration, Communicacion and Parallelism*. PhD thesis, University of Bordeaux, 2017.

[DGK07] Stephanie Demers, Praveen Gopalakrishnan, and Latha Kant. A Generic Solution to Software-in-the-Loop. In *MILCOM 2007 - IEEE Military Communications Conference*, pages 1–6. IEEE, oct 2007.

[DLS17] Rainer Dömer, Guantao Liu, and Tim Schmidt. Parallel simulation. In *Handbook of Hardware/Software Codesign*, pages 533–564. Springer Netherlands, Dordrecht, 2017.

[DMB19] E. Diaz, R. Mateos, and E. Bueno. Virtual Platform of FPGA based SoC for Power Electronics Applications. In *2019 IEEE 28th International Symposium on Industrial Electronics (ISIE)*, pages 1371–1376. IEEE, jun 2019.

[Dom16] Rainer Domer. Seven Obstacles in the Way of Standard-Compliant Parallel SystemC Simulation. *IEEE Embedded Systems Letters*, 8(4):81–84, dec 2016.

[DXZ+13] Dexue Zhang, Xiaoyang Zeng, Zongyan Wang, Weike Wang, and Xinhua Chen. MCVP-NoC: Many-Core Virtual Platform with Networks-on-Chip support. In *2013 IEEE 10th International Conference on ASIC*, pages 1–4. IEEE, oct 2013.

[ELM14] ELMG. Three Key Issues to Watch out for in the Digital Control of Power Electronics. Technical report, 2014.

[Emb10] Embecosm. Embecosm Guidelines for using TLM 2.0 temporal decoupling, 2010.

[Eur09] European Commission. Decision No 406/2009/EC of the European Parliament and of the Council of 23 April 2009, 2009.

[Eur12] European Commission. EU NER 300 programme, 2012.

[Eur13] European Commission. EU Horizon 2020, 2013.

[Eur18a] European Commission. Directive (EU) 2018/2001 of the European Parliament on the promotion of the use of energy from renewable sources, 2018.

[Eur18b] European Commission. Directive (EU) 2018/2002 of the European Parliament on energy efficiency, 2018.

[Eur19] European Commission. Fourth report on the State of the Energy Union, 2019.

[Eva19] Stephen Evanczuk. Embedded Markets Study. Technical report, Embedded, 2019.

[FLV14] John Fitzgerald, Peter Gorm Larsen, and Marcel Verhoef, editors. *Collaborative Design for Embedded Systems*. Springer Berlin Heidelberg, Berlin, Heidelberg, 2014.

[Fuj01] Richard M. Fujimoto. *Parallel and distributed simulation systems*, volume 1. 1st edition, 2001.

[Ger10] Andreas Gerstlauer. Host-compiled simulation of multi-core platforms. In *Proceedings of the International Workshop on Rapid System Prototyping*, pages 1–6. IEEE, jun 2010.

[GGS01] Graham C. Goodwin, Stefan F. Graebe, and Mario E. Salgado. *Control System Design*, volume 27. 2001.

[GRE+01] M. R. Guthaus, J. S. Ringenberg, D. Ernst, T. M. Austin, T. Mudge, and R. B. Brown. MiBench: A free, commercially representative embedded benchmark suite. *2001 IEEE International Workshop on Workload Characterization, WWC 2001*, pages 3–14, 2001.

[GTF+19] Gustavo Figueiredo Gontijo, Thiago Cardoso Tricarico, Bruno Wanderley Franca, Leonardo Francisco da Silva, Emanuel Leonardus van Emmerik, and Mauricio Aredes. Robust Model Predictive Rotor Current Control of a DFIG Connected to a Distorted and Unbalanced Grid Driven by a Direct Matrix Converter. *IEEE Transactions on Sustainable Energy*, 10(3):1380–1392, jul 2019.

[HHY+12] Ding Yong Hong, Chun Chen Hsu, Pen Chung Yew, Jan Jan Wu, Wei Chung Hsu, Pangfeng Liu, Chien Min Wang, and Yeh Ching Chung. HQEMU: A multi-threaded and retargetable dynamic binary translator on multicores. In *Proceedings - International Symposium on Code Generation and Optimization, CGO 2012*, pages 104–113, New York, New York, USA, 2012. ACM Press.

[HLGD18] Vladimir Herdt, Hoang M. Le, Daniel Grose, and Rolf Drechsler. Towards fully automated TLM-to-RTL property refinement. In *Proceedings of the 2018 Design, Automation and Test in Europe Conference and Exhibition, DATE 2018*, volume 2018-Janua, pages 1508–1511. IEEE, mar 2018.

[HMUH19] Muhammad Haseeb, Asad Waqar Malik, Anis Ur Rahman, and Mian Muhammad Hamayun. Toward Distributed Heterogeneous Simulation Using Internet of Things. *IEEE Internet of Things Journal*, 6(6):10472–10482, dec 2019.

[HWWR20] Long He, Fengxiang Wang, Junxiao Wang, and Jose Rodriguez. Zynq Implemented Luenberger Disturbance Observer Based Predictive Control Scheme for PMSM Drives. *IEEE Transactions on Power Electronics*, 35(2):1770–1778, feb 2020.

[IEC10] International, Electrotechnical, and Commission. IEC 61508 Edition 2.0 - Functional safety of electrical/electronic/programmable electronic safety-related systems, 2010.

[IEC19] IECON. 45th Annual Conference of the IEEE Industrial Electronics Society. Lisbon, 2019. IEEE.

[IEE11] Design Automation Standards IEEE Committee. *Standard IEEE Standard for Reference SystemC® Language Manual*. 2011.

[IEE20] IEEE. Taxonomy, 2020.

[ILG10] Syed Muhammad Zeeshan Iqbal, Yuchen Liang, and Hakan Grahn. ParMiBench - An Open-Source Benchmark for Embedded Multiprocessor Systems. *IEEE Computer Architecture Letters*, 9(2):45–48, feb 2010.

[Ima14] M. Imani. *Hardware-in-the-Loop simulation of servo drivers on an embedded system. Master Thesis.* PhD thesis, Royal Institute of Technology, Stockholm, Sweden, 2014.

[Imp08] Imperas. OVPsim, 2008.

[Ins21] National Instruments. CompactRIO System on Module, 2021.

[JHT+20] Pouya Jamborsalamati, M. J. Hossain, Seyedfoad Taghizadeh, Georgios Konstantinou, Moein Manbachi, and Payman Dehghanian. Enhancing Power Grid Resilience through an IEC61850-Based EV-Assisted Load Restoration. *IEEE Transactions on Industrial Informatics*, 16(3):1799–1810, mar 2020.

[JLU+14] Óscar Jiménez, Óscar Lucia, Isidro Urriza, Luis A. Barragan, and Denis Navarro. Analysis and implementation of FPGA-based online parametric identification algorithms for resonant power converters. *IEEE Transactions on Industrial Informatics*, 10(2):1144–1153, may 2014.

[JWLA19] Lukas Jünger, Jan Henrik Weinstock, Rainer Leupers, and Gerd Ascheid. Fast SystemC Processor Models with Unicorn. In *Proceedings of the Rapid Simulation and Performance Evaluation: Methods and Tools on - RAPIDO '19*, pages 1–6, New York, New York, USA, 2019. ACM Press.

[KC08] P.J. King and D.G. Copp. Hardware in the loop for automotive vehicle control systems development. In *UKACC Control 2004 Mini Symposia*, volume 2004, pages 75–78. IEE, 2008.

[KDB+18] Ozgur Kilic, Spoorti Doddamani, Aprameya Bhat, Hardik Bagdi, and Kartik Gopalan. Overcoming Virtualization Overheads for Large-vCPU Virtual Machines. In *2018 IEEE 26th International Symposium on Modeling, Analysis, and Simulation of Computer and Telecommunication Systems (MASCOTS)*, pages 369–380. IEEE, sep 2018.

[KKH19] Minseong Kim, Seon Wook Kim, and Youngsun Han. EPSim-C: A Parallel Epoch-Based Cycle-Accurate Microarchitecture Simulator Using Cloud Computing. *Electronics*, 8(6):716, jun 2019.

[KLG11] Sudipta Kundu, Sorin Lerner, and Rajesh K. Gupta. *High-Level Verification*. Springer New York, New York, NY, 2011.

[KYH16] Shin-haeng Kang, Donghoon Yoo, and Soonhoi Ha. TQSIM: A fast cycle-approximate processor simulator based on QEMU. *Journal of Systems Architecture*, 66-67:33–47, may 2016.

[LHL+16] Kilho Lee, Wookhyun Han, Jaewoo Lee, Hoon Sung Chwa, and Insik Shin. Fast and accurate cycle estimation through hybrid instruction set simulation for embedded systems. In *Proceedings - Real-Time Systems Symposium*, volume 0, page 370. IEEE, nov 2016.

[LP15] Alessandro Lonardi and Graziano Pravadelli. On the co-simulation of systemC with QEMU and OVP virtual platforms. In *IFIP Advances in Information and Communication Technology*, volume 464, pages 110–128. 2015.

[Mat06] Raúl Mateos. *HW / SW simulation techniques for the design and verification of SoC systems*. PhD thesis, PHD Thesis, 2006.

[Mat20] MathWorks. MathWorks - Matlab Simulink, 2020.

[Mat21] MathWorks. SimScape Limitations - Simulink Embedded Coder, 2021.

[MB16] Fuentes Morales and Jose Luis Bismarck. *Evaluating Gem5 and QEMU Virtual Platforms for ARM Multicore Architectures*. PhD thesis, Resultados de la búsqueda Resultado web con enlaces al sitio web KTH Royal Institute of Technology in Stockholm, 2016.

[MCE+02] P.S. Magnusson, M. Christensson, J. Eskilson, D. Forsgren, G. Hallberg, J. Hogberg, F. Larsson, A. Moestedt, and B. Werner. Simics: A full system simulation platform. *Computer*, 35(2):50–58, 2002.

[McG02] Ian McGregor. The relationship between simulation and emulation. In *Winter Simulation Conference Proceedings*, volume 2, pages 1683–1688. IEEE, 2002.

[MCJW17] Christian Menard, Jeronimo Castrillon, Matthias Jung, and Norbert Wehn. System simulation with gem5 and SystemC: The keystone for full interoperability. In *2017 International Conference on Embedded Computer Systems: Architectures, Modeling, and Simulation (SAMOS)*, pages 62–69. IEEE, jul 2017.

[Meh18] Ashok B. Mehta. *ASIC/SoC Functional Design Verification*. Springer International Publishing, Cham, 2018.

[MFR20] Roberto Millón, Emmanuel Frati, and Enzo Rucci. A Comparative Study between HLS and HDL on SoC for Image Processing Applications. *Elektron*, 4(2):100–106, 2020.

[MIN11] Eric Monmasson, Lahoucine Idkhajine, and Mohamed Wissem Naouar. FPGA-based controllers. *IEEE Industrial Electronics Magazine*, 5(1):14–26, mar 2011.

[MMB+13] Pedro Martin Sanchez, Osmell Machado, Emilio J. Bueno Pena, Francisco J. Rodriguez, and Francisco Javier Meca. FPGA-based implementation of a predictive current controller for power converters. *IEEE Transactions on Industrial Informatics*, 9(3):1312–1321, aug 2013.

[MOBD18] Federico Montano, Tarek Ould-Bachir, and Jean Pierre David. An Evaluation of a High-Level Synthesis Approach to the FPGA-Based Submicrosecond Real-Time Simulation of Power Converters. *IEEE Transactions on Industrial Electronics*, 65(1):636–644, jan 2018.

[Moy13] Matthieu Moy. Parallel programming with SystemC for loosely timed models: A non-intrusive approach. In *Proceedings -Design, Automation and Test in Europe, DATE*, pages 9–14, New Jersey, 2013. IEEE Conference Publications.

[MPNB12] F. Mendoza, J. Pascal, P. Nenninger, and J. Becker. Framework for dynamic verification of multi-domain virtual platforms in industrial automation. In *IEEE 10th International Conference on Industrial Informatics*, pages 935–940. IEEE, jul 2012.

[MUR09] Ned Mohan, Tore M. Undeland, and William P. Robbins. *Power Electronics. Converters, Applications, and Design.* Third edition, 2009.

[NBTN17] Van Nguyen, Yvon Besanger, Quoc Tran, and Tung Nguyen. On Conceptual Structuration and Coupling Methods of Co-Simulation Frameworks in Cyber-Physical Energy System Validation. *Energies*, 10(12):1977, nov 2017.

[ND15] Anh Quynh Nguyen and Hoang Vu Dang. Unicorn: Next Generation CPU Emulator Framework. In *Black Hat USA*, 2015.

[NI14] Walid A. Najjar and Paolo Ienne. Reconfigurable computing. *IEEE Micro*, 34(1):4–6, jan 2014.

[NLB+13] Denis Navarro, Oscar Lucia, Luis A. Barragan, Isidro Urriza, and Oscar Jimenez. High-Level Synthesis for Accelerating the FPGA Implementation of Computationally Demanding Control Algorithms for Power Converters. *IEEE Transactions on Industrial Informatics*, 9(3):1371–1379, aug 2013.

[NTB+18] Van Hoa Nguyen, Quoc Tuan Tran, Yvon Besanger, Tung Lam Nguyen, Tran The Hoang, Cedric Boudinnet, Antoine Labonne, Thierry Braconnier, and Herve Buttin. Cross-infrastructure holistic experiment design for cyber-physical energy system validation. In *International Conference on Innovative Smart Grid Technologies, ISGT Asia 2018*, pages 68–73. IEEE, may 2018.

[Nur07] Jari Nurmi. *Processor design: System-on-chip computing for ASICs and FPGAs.* Springer Netherlands, Dordrecht, 2007.

[OPA20] OPALRT. Real-Time simulation Real-Time Solutions OPAL-RT, 2020.

[PC14] Brennand Pierce and Gordon Cheng. Versatile modular electronics for rapid design and development of humanoid robotic subsystems. In *IEEE/ASME International Conference on Advanced Intelligent Mechatronics, AIM*, pages 735–741. IEEE, jul 2014.

[Plu06] A. R. Plummer. Model-in-the-Loop Testing. *Proceedings of the Institution of Mechanical Engineers, Part I: Journal of Systems and Control Engineering*, 220(3):183–199, may 2006.

[PMIG19] Ali Parizad, Sobhan Mohamadian, Mohamad Esmaeil Iranian, and Josep M. Guerrero. Power System Real-Time Emulation: A Practical Virtual Instrumentation to Complete Electric Power System Modeling. *IEEE Transactions on Industrial Informatics*, 15(2):889–900, feb 2019.

[PSI20] PSIM. Powersim: PSIM Electronic Simulation Software, 2020.

[PVL+17] Peter Palensky, Arjen A. Van Der Meer, Claudio David Lopez, Arun Joseph, and Kaikai Pan. Cosimulation of Intelligent Power Systems: Fundamentals, Software Architecture, Numerics, and Coupling. *IEEE Industrial Electronics Magazine*, 11(1):34–50, mar 2017.

[QEM] QEMU. QEMU official web page.

[RAVPM15] Juan J. Rodriguez-Andina, Maria D. Valdes-Pena, and Maria J. Moure. Advanced Features and Industrial Applications of FPGAS-A Review. *IEEE Transactions on Industrial Informatics*, 11(4):853–864, aug 2015.

[RKK+19] Kasim Rehman, Orthodoxos Kipouridis, Stamatis Karnouskos, Oliver Frendo, Helge Dickel, Jonas Lipps, and Nemrude Verzano. A Cloud-based Development Environment using HLA and Kubernetes for the Co-simulation of a Corporate Electric Vehicle Fleet. In *Proceedings of the 2019 IEEE/SICE International Symposium on System Integration, SII 2019*, pages 47–54. IEEE, jan 2019.

[RMdK19] Luiz Henrique Leite Rosa, Carlos Frederico Meschini Almeida, Danilo de Souza Pereira, and Nelson Kagan. A Systemic Approach for Assessment of Advanced Distribution Automation Functionalities. *IEEE Transactions on Power Delivery*, 34(5):2008–2017, oct 2019.

[SG] L. Semeria and A. Ghosh. Methodology for hardware/software co-verification in C/C++. In *Proceedings 2000. Design Automation Conference. (IEEE Cat. No.00CH37106)*, pages 405–408. IEEE.

[Sie] Siemens. QuestaSim.

[SLPH10] Christoph Schumacher, Rainer Leupers, Dietmar Petras, and Andreas Hoffmann. parSC: Synchronous parallel SystemC simulation on multi-core host architectures. In *Embedded Systems Week 2010 - Proceedings of the 8th IEEE/ACM/IFIP International Conference on Compilers, Architecture and Synthesis for Embedded Systems, CODES+ISSS'2010*, pages 241–246, New York, New York, USA, 2010. ACM Press.

[SMB+13] F. M. Sanchez, R. Mateos, E. J. Bueno, J. Mingo, and I. Sanz. Comparative of HLS and HDL implementations of a grid synchronization algorithm. In *IECON Proceedings (Industrial Electronics Conference)*, pages 2232–2237. IEEE, nov 2013.

[Soz17] Krzysztof Sozański. *Digital Signal Processing in Power Electronics Control Circuits*. Second edition, 2017.

[Syn21] Synopsys. ZeBu Server 4, 2021.

[Tes19] Tesla. Tesla. Financials and Accounting, 2019.

[TIM+19] Daniel Tormo, Lahoucine Idkhajine, Eric Monmasson, Ricardo Vidal-Albalate, and Ramon Blasco-Gimenez. Embedded real-time simulators for electromechanical and power electronic systems using system-on-chip devices. *Mathematics and Computers in Simulation*, 158:326–343, apr 2019.

[TPD16] Daniel Törtei Tertei, Jonathan Piat, and Michel Devy. FPGA design of EKF block accelerator for 3D visual SLAM. *Computers and Electrical Engineering*, 55:1339–1351, oct 2016.

[Trz10] Andrzej M. Trzynadlowski. *Introduction to Modern Power Electronics*, volume 3. 2010.

[TYH+19] Ha Thi Nguyen, Guangya Yang, Arne Hejde Nielsen, Peter Hojgaard Jensen, and Carlos F.M. Coimbra. Control parameterisation for POD via software-in-the-loop simulation. *The Journal of Engineering*, 2019(18):4864–4868, jul 2019.

[VPSH14] N. Ventroux, J. Peeters, T. Sassolas, and James C. Hoe. Highly-parallel special-purpose multicore architecture for SystemC/TLM simulations. In *Proceedings - International Conference on Embedded Computer Systems: Architectures, Modeling and Simulation, SAMOS 2014*, pages 250–257. IEEE, jul 2014.

[VRN+19] Steffen Vogel, Vetrivel Subramaniam Rajkumar, Ha Thi Nguyen, Marija Stevic, Rishabh Bhandia, Kai Heussen, Peter Palensky, and Antonello Monti. Improvements to the Co-simulation Interface for Geographically Distributed Real-time Simulation. In *IECON 2019 - 45th Annual Conference of the IEEE Industrial Electronics Society*, pages 6655–6662. IEEE, oct 2019.

[VS16] Nicolas Ventroux and Tanguy Sassolas. A new parallel SystemC kernel leveraging manycore architectures. In *Proceedings of the 2016 Design, Automation and Test in Europe Conference and Exhibition, DATE 2016*, pages 487–492, 2016.

[VT64] S. Vajda and K. D. Tocher. *The Art of Simulation.*, volume 127. 1964.

[WFMH20] You Wu, Lijun Fu, Fan Ma, and Xiaoliang Hao. Cyber-Physical Co-Simulation of Shipboard Integrated Power System Based on Optimized Event-Driven Synchronization. *Electronics*, 9(3):540, mar 2020.

[WLC+11] Zhaoguo Wang, Ran Liu, Yufei Chen, Xi Wu, Haibo Chen, Weihua Zhang, and Binyu Zang. COREMU. *ACM SIGPLAN Notices*, 46(8):213, sep 2011.

[WMLA16] Jan Henrik Weinstock, Luis Gabriel Murillo, Rainer Leupers, and Gerd Ascheid. Parallel SystemC Simulation for ESL Design. *ACM Transactions on Embedded Computing Systems*, 16(1):1–25, oct 2016.

[Xil18] Xilinx. Xilinx QEMU Github, 2018.

[Xil19] Xilinx. PetaLinux Tools Documentation Reference Guide, 2019.

[Xil20] Xilinx. Versal: The First Adaptive Compute Acceleration Platform (ACAP), 2020.

[Xil21a] Xilinx. Kria K26 SOM: The Ideal Platform for Vision AI at the Edge, 2021.

[Xil21b] Xilinx. Zynq-7000 SoC Technical Reference Manual - UG565, 2021.

[YMK14] Hassan Youness, Mohamed Moness, and Mahmoud Khaled. MPSoCs and multicore microcontrollers for embedded PID control: A detailed study. *IEEE Transactions on Industrial Informatics*, 10(4):2122–2134, nov 2014.

[ZBW21] Shuai Zhao, Frede Blaabjerg, and Huai Wang. An Overview of Artificial Intelligence Applications for Power Electronics. *IEEE Transactions on Power Electronics*, 36(4):4633–4658, apr 2021.

[ZYW+20] Yonglei Zhang, Xibo Yuan, Xiaojie Wu, Yalei Yuan, and Juan Zhou. Parallel Implementation of Model Predictive Control for Multilevel Cascaded H-Bridge STATCOM With Linear Complexity. *IEEE Transactions on Industrial Electronics*, 67(2):832–841, feb 2020.