Adaptive Quantum for Parallel Full System Simulation

Hugo José Duarte Ribeiro

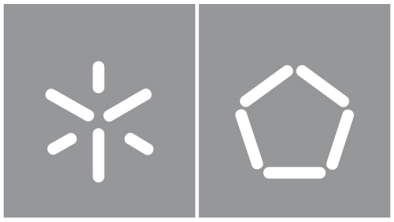
UMinho | 2024

Universidade do Minho
Escola de Engenharia

Hugo José Duarte Ribeiro

Adaptive Quantum for Parallel Full System Simulation

janeiro de 2024



Universidade do Minho
Escola de Engenharia

Hugo José Duarte Ribeiro

Adaptive Quantum for Parallel Full System Simulation

Dissertação de Mestrado
Mestrado em Engenharia Eletrónica Industrial e Computadores

Trabalho efetuado sob a orientação de
Professor Doutor Jorge Miguel Nunes Santos Cabral
Professor Doutor Rainer Leupers

janeiro de 2024

Acknowledgements

First of all, I would like to thank my supervisors, Prof. Dr. Jorge Cabral and Prof. Dr. Rainer Leupers, for the support and motivation that they gave during this journey.

Then, I would like to express my gratitude to Niko Zurstraßen. Niko was the first contact that I had when I arrived in Germany. He helped me not only in the development of this work, but also in the adaptation to a new city and new people. Also, a special word for Rui Almeida and Rui Machado, who supported me all the time with their knowledge and expertise. Furthermore, and most importantly, thank you all for your friendship.

I want to acknowledge also my family and friends, especially José Carvalho, João Carneiro, and my girlfriend, Anabela. They were crucial in this part of my life, not only backing me up but keeping me motivated and relaxed as well.

Lastly, I want to thank all the professors that I had since I started studying, for teaching me their knowledge and guiding me to achieve this goal. Just a special note to Prof. Mário Roque, and Prof. Rui Barreiro, who with their teaching and empathy guided me to choose the electrical engineering field, for which I am grateful.

STATEMENT OF INTEGRITY

I hereby declare having conducted this academic work with integrity. I confirm that I have not used plagiarism or any form of undue use of information or falsification of results along the process leading to its elaboration. I further declare that I have fully acknowledged the Code of Ethical Conduct of the University of Minho.

Abstract

In recent years, MultiProcessor System on a Chips (MPSoCs) complexity has been growing exponentially. Nevertheless, the performance of simulation tools is not following this growth, mainly because of their sequential simulation type. Therefore, simulation time increases each time a new MPSoC is developed. Concerning this problem, the Institute for Communication Technologies and Embedded Systems (ICE) Rheinisch-Westfälische Technische Hochschule (RWTH) Aachen team developed a parallel version of the atomic mode of Gem5, Par-gem5. It is based on a synchronous Parallel Discrete Event Simulation (PDES), which allows each simulation thread to run independently from the rest of the system for a time $t_{\Delta q}$ - called quantum. Although this is a huge improvement, it carries a challenge in the quantum definition for each simulation experiment. Additionally, as systems become more complex, the necessity of interaction between distinct simulator domains grows because different aspects must be taken into account.

This dissertation aims to solve two problems. The first one consists of the optimal quantum finding, which can lead to the best trade-off between simulation accuracy and performance. Nowadays, Par-gem5 is bound to a static quantum, which cannot be changed during run-time. However, an adaptive version can overcome this limitation. Thus, a flexible algorithm that can operate independently of the used benchmark and system was conceptualized and developed. The second problem lies in the interaction between Gem5 and other simulators. At its current state, few available frameworks can work with this tool. Thereby, it was proposed a co-simulation environment that can integrate any simulator. Further, it was chosen a study case to validate the developed framework.

The work developed resulted in an algorithm that brings greater benefits when compared with the present solution. It was possible to achieve, on average, a performance gain of almost 10%, only sacrificing 0.5% of accuracy. Nevertheless, when *a priori* information about the test is given, the tradeoff can be improved with the static approach, thus, it should not be fully discarded. If perfect accuracy is a requirement, the sequential version must be used, since the usage of both methods implies a loss in accuracy. Moreover, the proposed framework provided a new work environment. It maintained data integrity, data exchange, and synchronization between the tools during all the simulations. With this work, a new contribution to this subject was provided.

Keywords: Parallel Discrete Event Simulation, Gem5, Full-System Simulation, Quantum, Co-Simulation

Resumo

Nos últimos anos, a complexidade dos sistemas com múltiplos processadores num *chip* (MPSoCs) tem crescido exponencialmente. Contudo, as ferramentas de simulação destes não têm seguido esse crescimento, principalmente devido ao seu tipo de simulação sequencial. Assim, os tempos de simulação aumentam sempre que um novo MPSoC é desenvolvido. Nesse sentido, a equipa ICE da universidade RWTH, em Aachen, desenvolveu uma versão que possibilita a simulação paralela no modo atómico do Gem5, o Par-gem5. Esse modo é baseado numa simulação paralela síncrona de eventos discretos (PDES) que permite cada *thread* operar independentemente do resto do sistema por um tempo $t_{\Delta q}$, denominado quantum. Embora este trabalho tenha sido um importante avanço, ainda reside o problema de definir qual o melhor quantum para cada teste. Adicionalmente, à medida que os sistemas crescem em complexidade, a interação entre simuladores distintos aumenta devido à necessidade de avaliar diferentes aspetos.

Esta dissertação tem como objetivo solucionar dois problemas. O primeiro consiste em encontrar o quantum ótimo, ou seja, o quantum que permita obter o melhor compromisso entre desempenho e a precisão de simulação. De momento, o Par-gem5 apenas permite a definição de um quantum estático, pelo que este não pode ser alterado durante o decorrer da simulação. Porém, um quantum dinâmico consegue ultrapassar essas limitações. Portanto, foi desenhado e desenvolvido um algoritmo flexível que consegue operar independentemente do teste e do sistema escolhido. O segundo problema reside na interação entre o Gem5 e outros simuladores. No seu estado atual, poucos conseguem trabalhar com esta ferramenta. Deste modo, foi proposto um ambiente de co-simulação que permite integrar qualquer simulador. Além disso, um caso de estudo para validar este ambiente foi escolhido.

O trabalho desenvolvido resultou num algoritmo que trouxe grandes benefícios quando comparado com a solução atual. Foi possível atingir, em média, um aumento de desempenho de 10%, sacrificando, apenas, 0.5% de precisão. No entanto, quando existe informação prévia, este compromisso pode ser melhorado com a versão estática, pelo que esta não deve ser totalmente descartada. Caso seja necessária uma precisão perfeita, o modo sequencial deve de ser utilizado, já que os métodos anteriores implicam uma perda. Além disso, o ambiente de co-simulação proposto permitiu a interação com outros simuladores. Este manteve a integridade dos dados, a troca de dados, e a sincronização entre as ferramentas durante todo o processo. Com este trabalho, uma nova contribuição neste ramo foi desenvolvida.

Palavras-chave: Simulação Paralela de Eventos Discretos, Gem5, Simulação de Sistemas Completos, Quantum, Co-Simulação

Contents

List of Figures

List of Tables

Glossary

ADC	Anallog-to-Digital Converter
ANC	Adaptive Noise Cancellation
ANN	Artificial Neural Network
API	Application Programming Interface
ASIC	Application Specific Integrated Circuit
BNN	Biological Neural Network
CES	Continuous Event Simulation
CPU	Central Processing Unit
CRC	Cyclic Redundancy Check
CS	Context Switch
DAC	Digital-to-Analog Converter
DES	Discrete Event Simulation
DMA	Direct Memory Access
FFT	Fast Fourier Transform
FIFO	First-In First-Out
FPC	Forecasted Program Counter
FS	Full-System
FSS	Full System Simulator
GDB	GNU Debugger
GEMS	General Execution-driven Multiprocessor Simulator
GPIO	General Purpose Input Output
GUI	Graphical User Interface
HDL	Hardware Description Language
I/O	Input/Output
ICE	Institute for Communication Technologies and Embedded Systems
IFP	Instruction Flow Prediction
IP	Intellectual Property
IPC	Inter-Process Communication

ISA	Instruction Set Architecture
KPI	Key Performance Indicator
KVM	Kernel Virtual Mode
LED	Light-Emitting Diode
LMS	Least Mean Square
LoC	Lines of Code
MCU	MicroController Unit
MIPS	Millions-of-Instructions-Per-Second
ML	Machine Learning
MMIO	Memory Mapped Input Output
MPSoC	MultiProcessor System on a Chip
NIC	Network Interface Controller
NN	Neural Network
NPB	NAS Parallel Benchmarks
O3	Out-Of-Order
OS	Operating System
PC	Program Counter
PDES	Parallel Discrete Event Simulation
POSIX	Portable Operating System Interface
PTE	Page Table Entry
RAM	Random Access Memory
RTL	Register-Transfer Level
RWTH	Rheinisch-Westfälische Technische Hochschule
SDES	Sequential Discrete Event Simulation
SE	System-call Emulation
SoC	System-on-a-Chip
SP	Stack Pointer
SWaP-C	Size, Weight, Power, and Cost
TD	Temporal Decoupling
TDL	Tapped Delay Line
TLM	Transaction-Level Modeling
UART	Universal Asynchronous Receiver-Transmitter
VP	Virtual Platform

1 Introduction

Designing an embedded system should adopt a development model that matches the project type. For example, in the automotive industry, the most commonly adopted development model is the V-model [1]. This model allows step-backs in the development process due to potential problems or upgrades [2].

One example of this non-linear characteristic is in the development of an Application Specific Integrated Circuit (ASIC). As the name suggests, an ASIC is used for specific applications where dedicated hardware is required, for example, a critical system of a car. After the design, the prototype is developed, on which tests will be performed, or benchmarks, in order to understand if the developed ASIC satisfies all the requirements. Only in this stage, it is possible to obtain some indicators such as power consumption or compute performance to evaluate if Size, Weight, Power, and Cost (SWaP-C) requirements match. With Virtual Platforms (VPs) or Full System Simulators (FSSs), on the other hand, most of the time is possible to have those without the physical prototype. Therefore, the time to market can be accelerated because problems or upgrades can be spotted much sooner.

Although these simulation tools are useful for the design of modern massively parallel and complex multi-core systems, many cannot execute a parallel simulation, in other words, they cannot parallelize simulation workload. As the complexity of the new systems increases, due to the integration of more and more applications on a single chip [3], simulation time increases. For example, in the case of the SPEC2017 integer benchmark, where it may take up to two years to complete the simulation [4].

Another challenge in simulation is the communication between different tools. The system's complexity typically implies a simulation in different domains, since the integration of physical, software and network aspects is required [5]. Co-simulation is a technique used to model complex systems, allowing different stimuli from other simulators. The problem is that different tools can have different levels of abstraction, different communication protocols, etc. making it difficult to execute this technique directly.

1.1 Motivation

Gem5 is an example of an FSS that cannot parallelize target workloads during simulation. To solve this, the ICE RWTH Aachen team developed Par-gem5 [4], a parallel version of the atomic mode of Gem5, that exploits the multithreading capabilities of modern host systems to parallelize the simulation itself. It

is based on a synchronous Parallel Discrete Event Simulation (PDES) where synchronizations are done periodically, according to a fixed-size time window called quantum or quanta.

High quantum allows for high simulation speeds but negatively impacts the simulation's accuracy, or, in the worst case, can even break the system's functionality. If the quantum is too small, the accuracy is perfect, although the simulation performance will be unsatisfactory. Thus, there is a tradeoff between accuracy and performance, and finding an optimal quantum is one of the main challenges when running synchronous PDES.

In the current state of Par-gem5 (and as in other simulators), the quantum is set once and then kept for the rest of the simulation. This brings several different problems. First of all, to know which is the best quantum, it is necessary to do the simulation in order to obtain the simulation results, and further evaluate if it was the best choice or not. Moreover, the quantum varies on different simulations, therefore there is not a single optimal quantum value that fits all simulations. The trial and error process, although feasible, consumes a lot of time, allowing for a study regarding the optimization of this optimal quantum-finding process.

Another issue present in Gem5 is the lack of frameworks that allow co-simulation. Due to its open-source characteristics, it is possible to create or integrate any co-simulation environment. The problem is the time spent on this operation, in a way that is desirable to use other tools. Gathering this to the few available options at the present date, the Gem5 does not become an option.

1.2 Goals and Contributions

In the context of Par-gem5, a dynamic quantum could address the previously mentioned issue by adjusting the quantum value for each simulation, leading to improved results. This approach is particularly advantageous because it compromises multiple phases with distinct computing and synchronization characteristics. Specifically, for the computational phase, increasing the quantum is beneficial, while for the synchronization phase, reducing the quantum is recommended.

With the dynamic quantum, it will be possible to automatically tune to the best value without any user inputs or feedback. Therefore, the quantum adaptation must be "on-the-fly" and be independent of the simulated system or benchmark. Alongside, upon finishing simulation runs, the simulator, at the end of the benchmark execution, should give feedback to the user, by the creation of a document including information related to the adaptive quantum.

In the context of co-simulation, a framework will be developed to facilitate the integration of different tools. Its design must be flexible, so it can be used with different simulators, and respect three characteristics in order to have a well-functioning environment: data integrity, data exchange, and tool synchronization. Additionally, it should be a lightweight framework, ensuring that simulation overhead is not overly significant.

In the end, the development of this dissertation will not only aim to provide an algorithm that solves the problem of finding the best compromise between performance and accuracy, making it possible to simulate massively parallel and complex systems faster without a break in the accuracy, but also a co-simulation framework that will enable the use of Gem5 in projects with more application domains.

From an industry point of view, this work can help companies to improve the development of new products, and optimize time-to-market.

1.3 Dissertation Outline

This document provides the development of a simulation extension that allows Par-gem5 to automatically address the best quantum for the desired benchmark, together with a co-simulation framework proposal. Its content was divided into six chapters, which are going to be briefly described.

The second chapter introduces the concepts and methodologies used. The reader will receive notions about simulation in a general view, which will be specified in the embedded systems context. Topics such as simulation modes, simulation methodologies, and simulation tools will be covered, with special attention to Gem5, the simulator focused on this dissertation. Machine Learning (ML) will be exploited, having in consideration Neural Networks (NNs) and the simulation context. Finally, co-simulation practices are discussed, emphasizing their significance in the simulation coding.

The third chapter presents the developed work done regarding the dynamic quantum. It starts by giving a short explanation of the used benchmarks to test the algorithms and required changes in Par-gem5. Then, the four developed algorithms will be covered, providing an insight into the need to develop such algorithms and how they actually work. It ends with the final version, where a direct comparison with the static mode is made. The simulation results will be provided throughout the chapter in order to help the reader to have a better idea of the algorithm's evolution.

The fourth chapter contains a proposal for a co-simulation environment, providing an interface where the two tools can cooperate. A case study was chosen to validate the developed work, giving a practical example of how this technique might be useful. It was based on a Cyclic Redundancy Check (CRC) peripheral, which is used for communication error detection and memory integrity checks. Since in a real-world scenario there are other devices to manage, the framework was tested alongside other tasks. In the end, the integration of the Par-gem5 with the dynamic quantum was done, in order to evaluate its advantages and disadvantages in this situation.

The final chapter discusses the conclusions regarding the developed work. It also includes future work, where a set of aspects can be found on which this work can have continuity and further improve Par-gem5.

2 State of the Art

This chapter establishes the needed concepts to accomplish and understand this work. Initially, the chapter will start with the topic of simulation, exploring its definition and clarifying its significance. Embedded systems are briefly contextualized, highlighting the connection they share with the previous topic. Then, simulation types and modes are introduced, with a special focus on the ones that will be used in this dissertation. Gem5 will be presented in detail since it is the simulator where all the development will be done. Machine learning and co-simulation will end the chapter, providing a short view of these topics in the project's context.

2.1 Simulation

Simulation is a very recent topic in the literature. The graph presented below attests to the fact that this subject began to be studied solely in the early 1950s. Prior to the advent of computers, it was unfeasible to replicate anything without the actual object or prototype, thereby rendering this subject of relatively low significance in the industry. However, with the introduction of computers, a new realm of opportunities emerged as a "virtual world" was conceived. As computers became smaller, more robust, and more cost-effective, a panoply of novel technologies and topics began to flourish, including simulation.

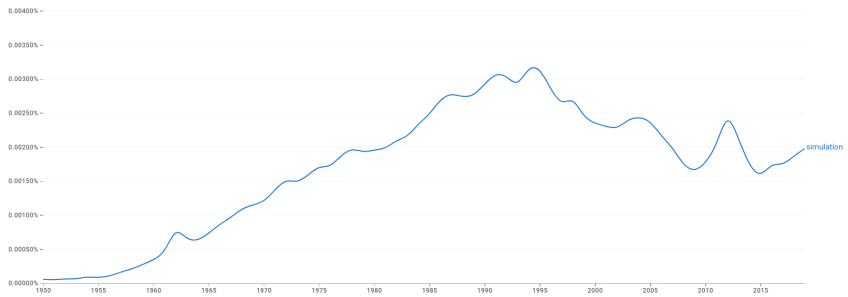


Figure 2.1: Evolution of the simulation topic in the literature by Google Books ngram viewer

Analyzing the market research report by ZION, it is possible to conclude that the global virtual training and simulation market size was valued at $332.6 Billion in 2022 and it is projected to reach $973.4 Billion by 2030.

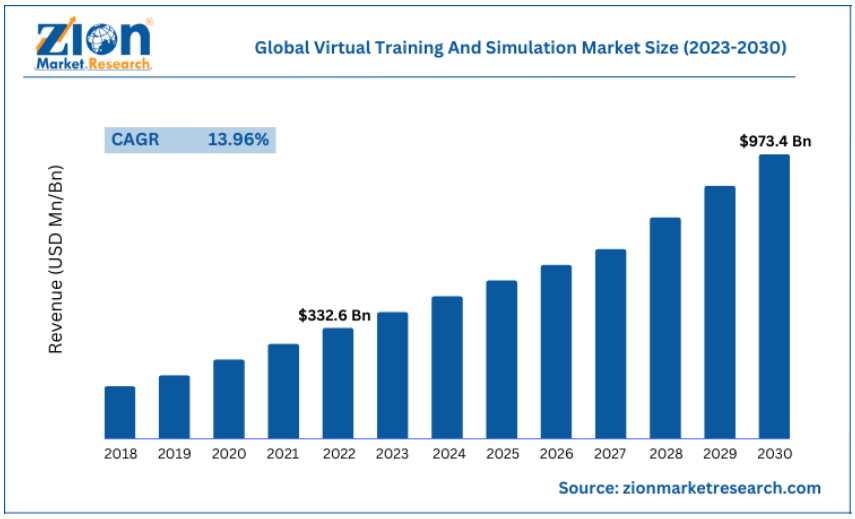


Figure 2.2: Market research report by ZION about simulation [6]

This growth and demand in the industry turn this topic into an important subject. For this reason, it is crucial to understand what is simulation and why is it necessary.

2.1.1 Definition

According to the Cambridge dictionary, simulation can be defined as *"a situation or event that seems real but is not real, used especially in order to help people deal with such situations or events"*. In a more general way, it can be defined as *"an imitation of a system"* [7]. Therefore, simulation is not exclusively confined to computers but also encompasses a diverse range of applications, both physical and virtual domains. For instance, in the automotive industry, each new model that is designed must undergo a security test. One of the numerous examinations performed involves simulating a car crash into a wall and assessing the driver's resulting damages. As a result, conclusive determinations can be made regarding whether the car has successfully met the test's requirements or not.

Hence, simulation is a controlled verification technique that helps to understand how a system would behave in a real situation. As previously mentioned, simulation can be physical or virtual. Nowadays, the last is the preferable one, since it brings huge advantages when compared to the other. First of all, the cost. It is clear that physical simulations, like the one previously described, have a lot of associated costs: the car that is going to be destroyed, the workers to make sure everything goes as planned and to clear all the wreckage, and even the infrastructure, that requires a designated area which involves costs and space. The second reason is time. In a competitive industry, the first to have the product in the market can be the one that will have more success. Physical simulation can take a lot of time. It may be necessary, beyond the simulation itself, preparation, post-cleaning, license acquisition, weather conditions, etc. which delay the process. The last factor is the computer's evolution. As noted earlier, its evolution was crucial in the way that more complex simulation environments can be tested and more accurate results can be obtained.

Looking in detail at the concept of virtual simulation, it can be categorized into three distinct types. It is illustrated in the Figure 2.3, which outlines the taxonomy of virtual simulations.

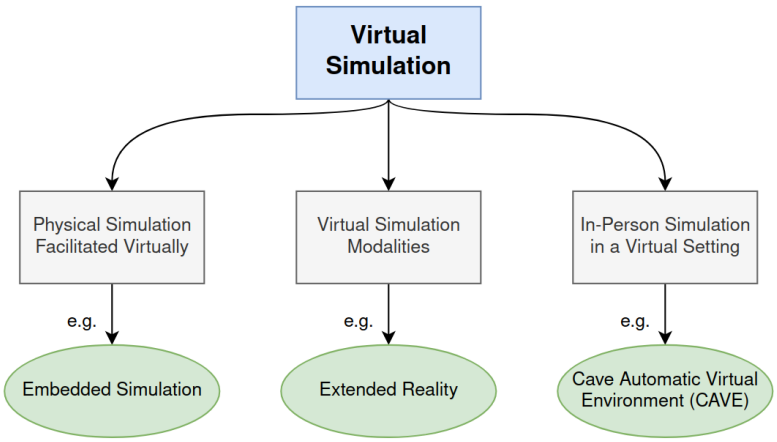


Figure 2.3: Taxonomy of virtual simulations (adapt from [8])

As shown, it can be divided into physical simulation facilitated virtually, virtual simulation modalities, and in-person simulation in a virtual setting.

Physical simulation facilitated virtually is the process of utilizing virtual simulation to replicate a physical phenomenon or system. This is done through the creation of a digital model that imitates the physical object, and through various algorithms and computational methods, simulates its behavior in a virtual environment. In this way, it is possible to predict how the physical system will behave in various scenarios, without the need for physical testing, saving time and resources. It is used, for example, in the development of an embedded system, where it would be mandatory to have the physical prototype to evaluate if the system matches all the requirements.

This dissertation will focus on computer architecture simulation. Therefore, the concept of simulation can be redefined as *"the imitation of the operation of a real-world process or system over time"* [9]. The computer architecture itself becomes the system under study, and simulation enables the user to explore its capabilities and limitations in a simulated environment.

2.1.2 Importance of Simulation

To have a full scope of why simulation is important, three topics should be considered: how the nature of the system affects the simulation, and the advantages and disadvantages of this technique compared to other validation methods [7].

The first topic exhibits three characteristics: variability, interconnections, and complexity. Consider the production of a Central Processing Unit (CPU) chip as an example. The production process may have interconnections with other factors such as the chip manufacturers or the chip designers. These

interconnections can make it difficult to predict the overall performance of the system, especially when variability is present. Furthermore, systems can be complex, and these complexities make it challenging to predict the performance of a system when changes are made or actions are taken. With simulation models, these problems can be solved in a way that they can explicitly represent the variability, interconnectedness, and complexity of a wanted system.

Keeping the example of CPU chip development, the process goes through various stages and, in the end, it is crucial to test the product before mass production. Thus, a prototype is built so that tests can be made. However, building a prototype takes time and money. In the majority of cases, the prototype can be improved, so that developers need to go through the process again. A new cycle begins, which means spending more time and money. Simulation can solve this problem in the way that development and testing can be done in parallel. In other words, while the developers are creating the chip, it can be tested in simulation, even without the physical prototype.

Simulation is flexible, which means that small changes can be made easily; it is controlled, that is, it can control the experimental conditions in a way that direct comparisons can be done; and it is transparent, ensuring that there are no subjective results.

In terms of drawbacks, the development team may consist only of individuals specialized in CPU chip design. Therefore, the team would need to hire a new employee with expertise in simulation to fulfill the requirements. Moreover, it requires powerful computers and a huge sufficient storage capacity, as simulation results can be in the order of dozens of gigabytes. The software to run the simulation and the creation of the model to test it requires investments. All these aspects represent an enormous investment for the company, where it will not get a direct profit. On top of that, simulations are not 100% accurate, meaning that they can produce results that are not the correct ones.

Considering all the aspects, simulation brings a lot of benefits when compared to other techniques, such as experimentation in real life. Still, the downsides must be taken into account and concluded whether this solution fits the needs or not. Banks in [10] mentions examples of applications where it can be used, like computer system performance, food processing, transportation systems, and embedded systems.

2.2 Embedded Systems

Embedded systems have a ubiquitous presence in modern life, from cars, computers, and televisions to smartphones and even toothbrushes. It is so common and present everywhere that the world without it wouldn't be the same. While this topic is not new to academia, with books like [9] mentioning it as early as the 19th century, it has gained more attention and deeper analysis since the 1980s until nowadays.

The following topics explain more about how embedded systems can be defined, and which are the development processes used in the industry. In the end, a specific topic about simulation in embedded systems will be covered.

2.2.1 Definition

The definition of embedded systems is not straightforward, that is, there is no agreement in the literature mostly because this term is being under development every day. One possible definition can be a combination of computer hardware and software, and perhaps additional mechanical or other parts, designed to perform a dedicated function [11]. Other authors prefer to highlight their relevant properties, like Raul Camposano in [12], where he focuses on correctness, real-time, and dedicated functions, among others, as the important aspects to retain when an embedded system is mentioned.

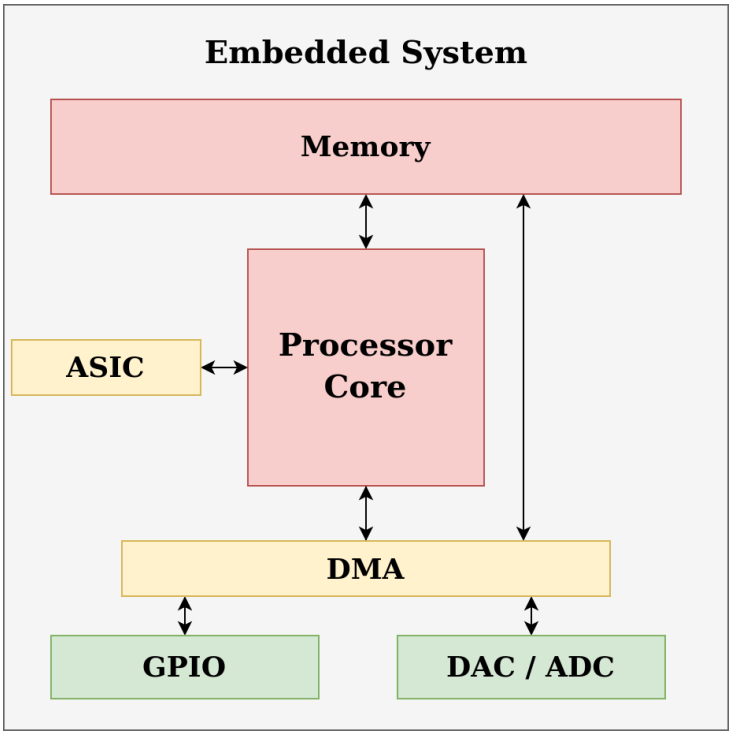


Figure 2.4: Typical embedded system (adapted from [12])

In the same work, he defined how a typical embedded system is, which is shown in the Figure 2.4. The colors depicted in the figure indicate the significance of each component in the embedded system. Red denotes an indispensable part that must be present. Yellow means that it is not mandatory, although it is recommended. Green stands for optional, as it can be used or not depending on the application. The arrows are used to demonstrate if the communications between the different parts are one-directional or bidirectional.

The processor core and memory are the two crucial components in an embedded system, as they are interdependent and cannot function without each other. Specifically, the processor requires the memory to access the data that needs to be processed, while the memory itself cannot perform any processing on its own. Therefore, the existence of both components is essential for the development of a functional embedded system. The Direct Memory Access (DMA) is a very important part of an embedded system. It is a hardware mechanism that allows peripheral components to transfer their Input/Output (I/O) data

directly to and from main memory without the need to involve the system processor. Thus, the significance of this lies in its ability to avoid a substantial processor overhead [13]. An ASIC, as the name suggests, is used for specific applications where dedicated hardware is required. It can reduce significantly the execution time of certain applications [14] [15] [16]. For this reason, many systems require dedicated hardware, however, it will become a smaller fraction as time passes [12]. General Purpose Input Output (GPIO), Anallog-to-Digital Converter (ADC), and Digital-to-Analog Converter (DAC) are examples of parts that establish an interface between the embedded system and the external world, and, therefore, they may be necessary or not, depending on the requirements.

The range of embedded systems is very wide. It starts from low power, like receiving data from ADC and sending it to a database, to multi-core applications, for example, laptops, smartphones, or even tablets. Hence, when designing a system, it is important to comply with the SWaP-C constraints, which means that the system can only use the essential resources, commonly referred to as "resource-constrained devices" [17].

Furthermore, embedded systems should be dependable, that is, they should perform as and when required [18]. The following characteristics must be fulfilled for dependability, nevertheless, different applications can attend better to different topics. In the case of cars, trains, or planes, the focus should be more on safety, whereas databases or banks should prioritize security.

1. **Reliability:** Continuity of service delivery while in use, e.g., the probability of the system working properly since it worked after startup;
2. **Maintainability:** Capability to be retained in, or restored to a state to perform as required, under given conditions of use and maintenance;
3. **Safety:** Ability to not cause catastrophic effects on the environment as a consequence of a failure;
4. **Availability:** Capacity to be in a state to perform as required;
5. **Security:** Capability to provide communication confidentiality and authentication.

2.2.2 Operating Systems

An embedded system can be complex. Take a laptop as an example. There are lots of different resources to manage, each one with its characteristics. Screen, mouse, keyboard, I/O devices, and network interfaces are some examples. Creating code to control all of them efficiently is almost impossible since the programmer needs to understand in detail every component. Operating Systems (OSs) came to solve this problem by providing a software layer that not only abstracts all the details for the user but can also optimally manage all the resources. The next picture presents how an embedded system with a OS can be divided. It is important to point out that every embedded system does not need to have an OS, like a device that only reads a ADC port and sends the information.

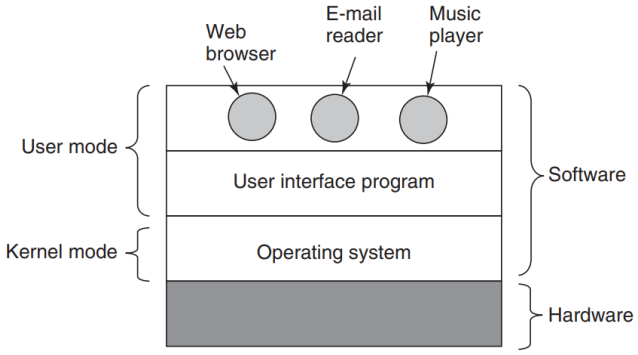


Figure 2.5: Layers of an OS [19]

Going into detail in the OSs, one important concept to keep in mind is the **process**, which is an instance of an executing program. Processes are crucial in a way that they allow the system to have multiple jobs at the same time. Following the previous example, a user with a laptop can play music and surf the internet at the same time. Even with one CPU, OSs have the capability of keeping track of multiple processes, by switching from process to process speedily, creating the illusion of parallelism. This concept is called **concurrency** and it is illustrated in the Figure 2.6

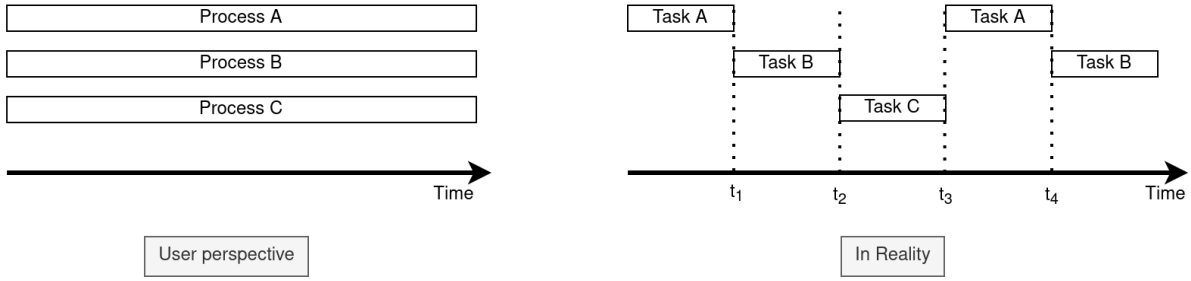


Figure 2.6: Concurrency between processes

Note that, in this example, it is considered a system with one CPU core. It is clear that if the system has more cores, which is common nowadays, the OS can split the work among them and have real parallelism. If the system has four CPU cores, each core can execute a process at a time, increasing the performance even more. To do this concurrency, the **scheduler** manages the processes by attributing states to them. There are three states: running (the process is in execution at that time), ready (the process is not running at the moment but it is ready), and blocked (the process is unable to run until some external event happens).

When thinking about programs, it is clear that one can have multiple works to do at the same time, for instance, a web browser. It needs to display the content, receive the keyboard and mouse information, receive web packages, and so on. Purely receiving keyboard information without handling the remaining tasks would render the process inefficient. Therefore, processes can have multiple **tasks** that run within the process, which are called **threads**. Threads are essential for the following reasons [19].

1. They have the ability for the parallel entities to share address space and all of its data among themselves;
2. They are lightweight and easier/faster to create and destroy than processes (Around 10-100 times faster);
3. They can increase performance when there is substantial computing and I/O because it allows these activities to overlap;
4. They are useful on systems with multiple CPUs, where real parallelism is possible.

Processes are assigned independent memory regions, and the threads within the process share the process memory between them. A process can have multiple threads, but a thread can have only one process associated. Without threads, a process would contain one single stack and a set of registers. With threads, each of them has a stack and a set of registers. **Execution context** is represented with a combination of registers and stack, representing the CPU state for each task. Figure 2.7 illustrates the previous notion.

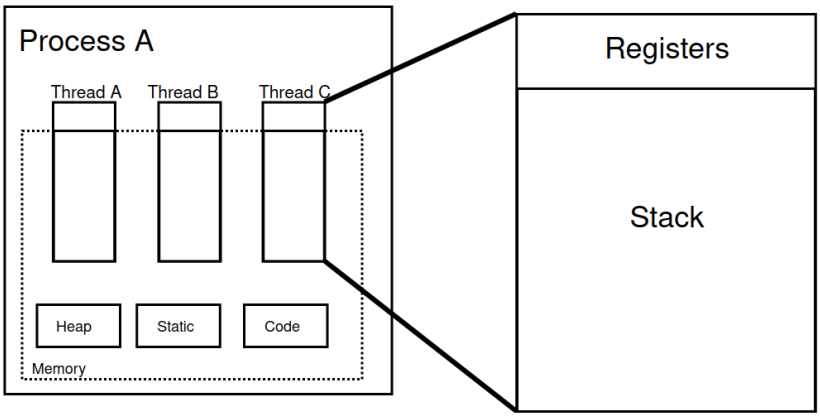


Figure 2.7: Process and threads block diagram

with threads, concurrency can also be applied, in order to have parallelism and better computational performance. Similar to processes, the scheduler defines which task will be executed. There are four states: run (the task is in execution at that time), ready (the task is not running at the moment but it is ready), wait (the task is waiting for data), and null (the task is created or killed). Figure 2.6 can also be applied to this context.

Nevertheless, in reality, when a task is running and another replaces it, there are some steps to pursue. First of all, it is necessary to save the CPU state of the running task. After that, the scheduler needs to select another to get running. Then, it restores the context of the selected one, so it can transfer the execution control. All this process is called **Context Switch (CS)**. The time consumption of the CSs is a problem, as present in the figure below. However, the benefits of having concurrency, as explained earlier in this section, are massive, and, thus, the trade-off is positive.

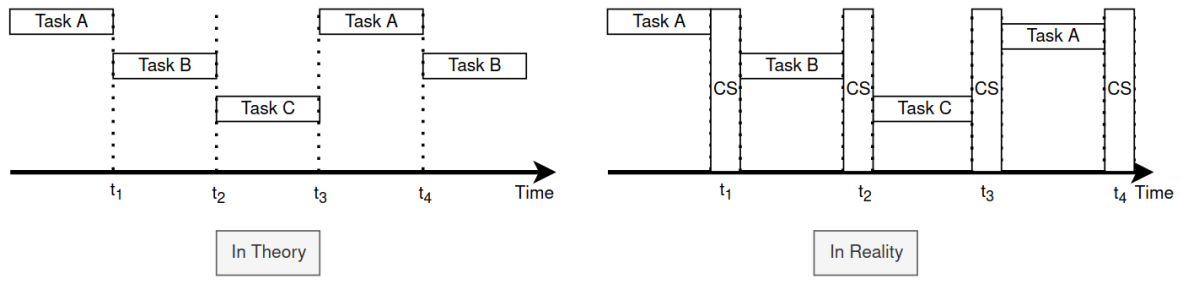


Figure 2.8: Context switch

Another issue that can take plan is when two different processes are using the same memory. By definition, they are assigned independent memory regions, but there are situations where this is not true. The most common is when two or more different processes interact with each other, where one, for example, writes in the memory and the others read from it. To control this **Inter-Process Communication (IPC)**, two mechanisms were created: data transfer and shared data.

Data transfer involves the concepts of writing and reading. The most typical IPCs are pipes and message queues. The main difference between both is that pipes are unidirectional communication channels, while message queues are bidirectional.

Shared data is a memory region that is shared by multiple processes. Thus, if a process wants to share data, it only needs to make it available in the shared memory region. For this reason, it is the fastest IPC available because the data transfer occurs at the speed of memory access, but it is dangerous because it has the same meaning as global variables, so there is no control to access them. The next figure shows a possible problem that can happen with shared memory.

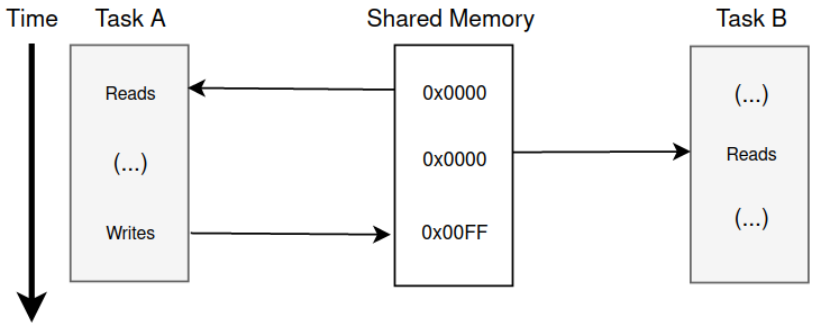


Figure 2.9: Two different tasks using the same resource

In this example, task A reads the value from the shared memory region, executes an equation, and writes the result in the same variable. For instance, this equation can be as simple as summing 0xFF (Y = X + 0xFF). Task B reads that value and prints it on the terminal. As demonstrated in the figure, task B will read 0x0000 even though task A was using the variable. In the case of multiple tasks and the result

of one matters to another, this issue can be a serious problem. This problem is called **race condition**, and it is more critical with the increasing parallelism due to increasing numbers of cores.

To avoid this, **task synchronization objects** were created. The most important methods are semaphores and mutexes. The first one can be divided into binary semaphores and counter semaphores, and they are useful methods to synchronize interrupts with a given task. The second one is a shared variable that can be in one of two states: owned or free. Binary semaphores and mutexes share many characteristics. Both can be utilized for mutual exclusion, ensuring that only one thread accesses a shared resource at a time, but only the first can be used for synchronization [20]. Regarding the previous example, Figure 2.10 illustrates the same situation but using a mutex.

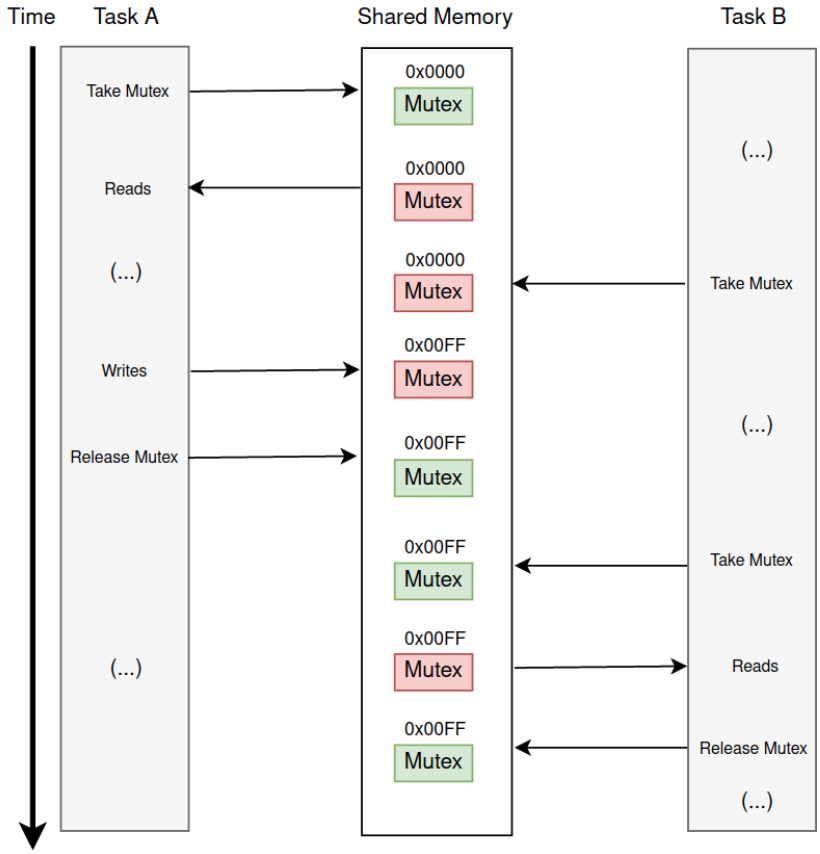


Figure 2.10: Two different tasks using the same resource with mutex

Now, both tasks, before using the shared variable, need to take the mutex. If the mutex was already taken (first trial of task B), the task will receive a blocked resource response, meaning that it will be unavailable until the owner releases the mutex.

2.2.3 Development Models

When building an embedded system, several aspects should be taken into account, like its requirements, constraints, and levels of complexity. Therefore, one development model that works well for one

project may not work well for another with different requirements or constraints. This section will present some examples of models typically used in the industry and research teams. There are a lot more models with different characteristics, thus, before the beginning of the project, all possibilities must be considered.

Waterfall model

The traditional waterfall model is linear, that is, there are well-defined stages and once the project moves to the next stage, it can not go back [21]. Figure 2.11 represents the different stages of the model.

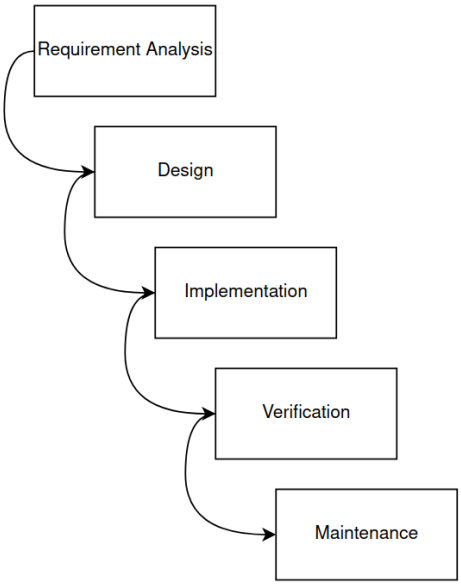


Figure 2.11: The waterfall model

Because of this linearity, no previous stages can be reviewed or improved, which makes the traditional waterfall model a good option for projects that have well-defined requirements and a short development cycle. Adetokunbo in [21] goes further and states that it is also viable to use it when altering the software after coding is strongly prohibited.

Some variations allow for an iterative relationship between phases and even add more phases. Royce in [22] presents and explains some of them in detail.

V-Model

The verification and validation model, more known as V-model, is a modified version of the Waterfall method [23]. It is non-linear, which means that it allows step-backs in the development process. An important aspect of this model is that testing activities like planning, and test designing happen well before coding, preventing bugs or bad-functional systems [24]. The following picture shows a typical representation of this model.

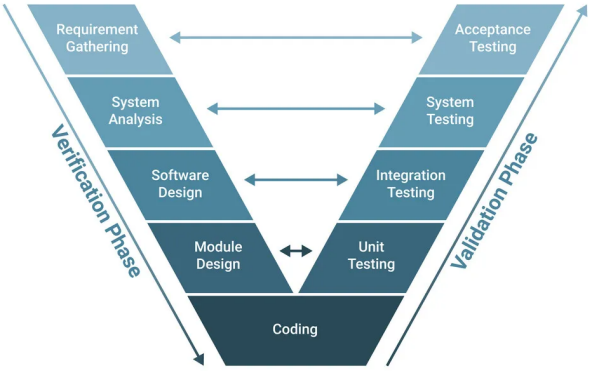


Figure 2.12: The V-Model

The main advantages of this model are the proactive tracking of defects in the various phases, cost reduction in the correction of defects since these are spotted much sooner, and it is simple to understand and apply. On the other hand, it lacks flexibility as any changes require updating the majority of documents, demanding significant time and resources, which can make it challenging for companies to adopt [23] [24]. Nevertheless, it is arguably the most traditional model utilized for software test management [25].

Other variations try to improve these downsides. Some examples are the shark tooth and W-model [24].

Agile Model

Although there are lots of agile techniques, they share common characteristics, including iterative development and a focus on interaction, communication, and the reduction of resource-intensive intermediate artifacts [26]. Therefore, this model states that the project should be divided into mini-projects to remove unnecessary activities that waste time and effort. These mini-projects have requirements analysis, design, implementation, and test [26], and should not exceed 30 days [27]. In the end, all of them are combined to obtain the final project.

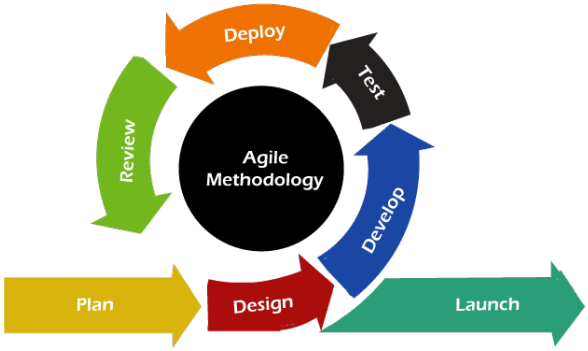


Figure 2.13: The agile model

Figure 2.13 shows the agile model. In the review stage, an important aspect happens, which is customer interaction. The customer adaptively specifies his requirements for the next release based on the observation of the current release, rather than speculating at the start of the project [28].

2.2.4 Embedded Simulation

All the models presented in the subsection 2.2.3 have the verification stage, which is crucial to validate if the developed system meets the desired specifications. This validation normally requires the construction of a prototype in a way that engineers can evaluate its performance, test its functionality, and identify any potential issues or shortcomings. For example, in the automotive industry, where the most commonly adopted model is the V-model [1], the construction of a mock-up is necessary to identify flaws, delaying the overall development process.

Therefore, simulation in the context of embedded systems is essential since the system can be tested without having the physical prototype, and some requirements, such as cache hits/misses and computer performance, can be evaluated [4]. A simulator of this type is commonly referred to as a FSS or a VP, and they can be described as a computer architecture simulator that simulates software in an electronic system, being this independent of the nature of the host computer. Usually, this term is mixed with emulation because although it can also run software tests inside flexible software-defined environments, an emulator goes beyond by simulating both software and hardware configurations, being useful to test how software interacts with underlying hardware or a combination of both.

There are two types of FSSs, full system hardware simulator and full system software simulator. The hardware version offers a cycle-accurate simulation. As the name suggests, this technique is employed to perform a comprehensive analysis of the simulated embedded system at the clock level. Therefore, it is possible to accurately simulate hardware state transitions, obtaining specific information as the state of all logic gates or registers. Nevertheless, it can have very poor simulation performance because hardware-level simulators are not suited for the simulation of such complex systems as the embedded ones [29]. Figure 2.14 shows the simulation of a system using a hardware simulator. Some examples of this kind of simulator are GHDL [30] and Icarus Verilog [31].

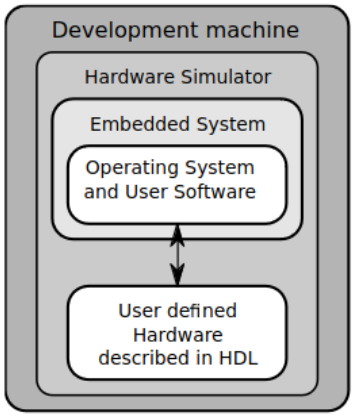


Figure 2.14: Full system hardware simulator [29]

The software version typically employs instruction-accurate simulation, where computations are performed according to the instruction set. However, this type of simulation does not provide information about the execution time of each instruction, resulting in a less detailed simulation that runs faster. Because of that, the hardware is not described in Hardware Description Language (HDL), unlike the previous version, as presented in the Figure 2.15.

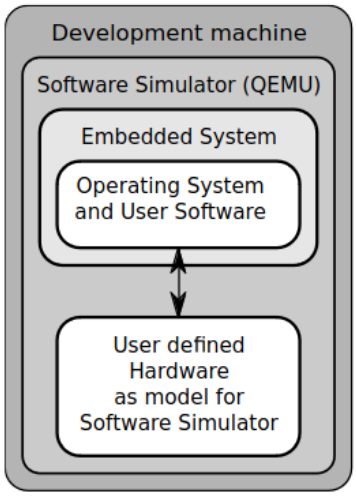


Figure 2.15: Full system software simulator [29]

The hardware model must be defined for the software simulation, thus, these may or may not be available for the target machine. It requires the selection of a simulator that aligns with the application scenario or offers the flexibility to expand the range of supported hardware modules, such as QEMU [32] and Gem5 [33].

As modern embedded and integrated systems become increasingly complex, an escalating number of design companies are embracing virtual prototyping methods [34]. VPs should follow this trend, that is, as embedded systems are becoming more complex, VPs should be faster yet accurate, so that they can be a reliable option.

The problem is that VPs are not following this evolution, which results in low performance, and, thus, high simulation times [4] [34] [35]. Although several works have tried to solve the problem, none of them have been able to provide an effective solution.

2.3 Discrete Event Simulation

Simulations, while valuable, cannot guarantee 100% reliability due to certain scenarios that may be challenging to evaluate. For instance, certain physical phenomenon modeling may not be fully accurate when compared with real-world behavior. However, to enhance reliability, a simulator should strive to accurately represent real-world conditions. Discrete Event Simulation (DES) aligns with these requirements by simulating system dynamics event by event and providing comprehensive performance reports.

A DES can be defined as a simulation technique where state changes (events) happen at discrete instances in time. Events take zero time to happen, and it is assumed that nothing happens between two consecutive events, that is, no state change takes place in the system between the events. A group of events organized by execution order is called an event queue or process [36] [7]. From this point forward, whenever the term "process" is mentioned, it specifically refers to the event queue.

Consider a supermarket as an illustrative example. A supermarket system consists of one employee and two cashiers, PAY1 and PAY2. In this context, three events can be specified, and they are related to each other as shown in the Figure 2.16.

- Customer arrives at the supermarket (CA)
- Customer goes to the cashier (PAY1 / PAY2)
- Customer leaves the supermarket (CL)

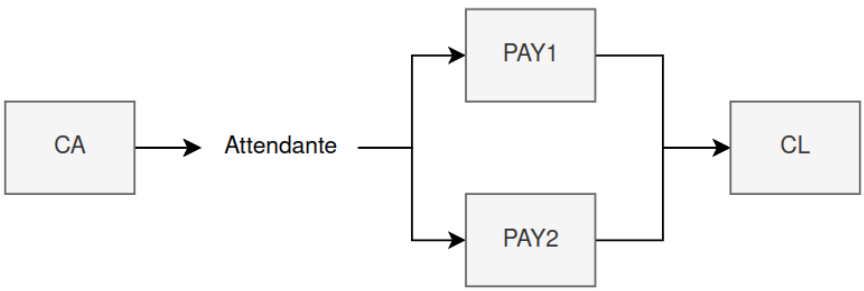


Figure 2.16: Supermarket flow schematic

Clients arrive randomly at the facility, and an attendant directs them to a cashier, according to the following rules: Whether both cashiers are available, the customer goes to PAY1; If only one of them is free, the client goes there; If both are occupied, the buyer waits until one is ready; If another buyer wants

to pay, a queue is created following a First-In First-Out (FIFO) style. The next table presents a situation assuming five clients and a payment time of five units of time.

Table 2.1: Example of a sequence of events in a supermarket

Event number	Time	Event description
1	1	CA1
2	1	CA1 goes to PAY1
3	3	CA2
4	3	CA2 goes to PAY2
5	4	CA3
6	5	CA4
7	6	CL1
8	6	CA3 goes to PAY1
9	7	CA5
10	8	CL2
11	8	CA4 goes to PAY2
12	11	CL3
13	11	CA5 goes to PAY1
14	13	CL4
15	16	CL5

When the time between arrival and reaching the attendant is considered to be zero, it indicates that there is no significant activity occurring. Otherwise, an additional event would have been included. In addition, when an event occurs, it is referred to as an event timestamp, typically measured in the same unit as the time within the model, known as the simulation time. Wall clock time or CPU time refers to how long the simulation program has been running and how much time it has consumed on the target system. On the other hand, the amount of time spent, known as host or real-time, represents the actual duration taken to perform the simulation.

In this example, some events depend on others, for instance, event number eight is executed only when event seven has been executed as well. However, others are independent, like events one and two. So, even in this simple case, there are relationships between events that can complicate the concurrent execution of events, not allowing for a reduction in wall clock time.

This method has applications not only in the context of embedded systems but also in a wide area of topics. It is traditionally used for industrial applications, for example, in the design and evaluation of new manufacturing processes, and in the establishment of optimum operational policies [37].

2.3.1 Continuous Event Simulation

Besides DES, there is another simulation technique, which is the Continuous Event Simulation (CES). The perfect one does not exist, since they should be chosen to take into account the application. Therefore, a brief explanation will be made to understand when one is better than another.

Continuous simulation is used for modeling systems with continuous or analog behaviors. The state changes are continuous since they are modeled by differential equations. [38] and [39] are examples of applications where the authors decided to use this technique.

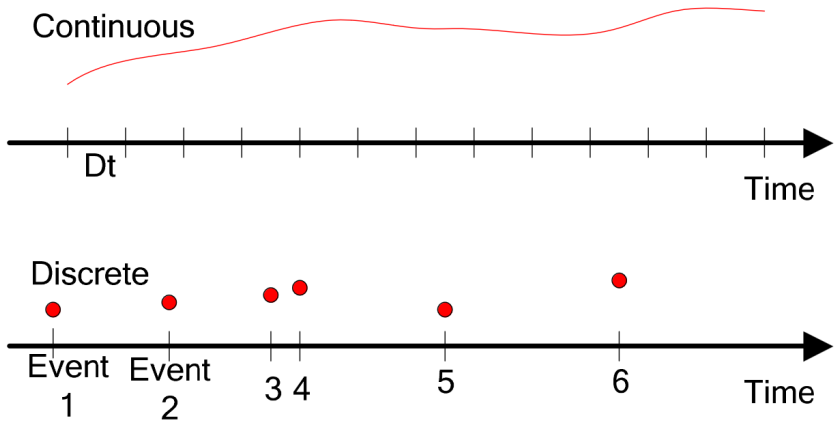


Figure 2.17: Updating state over simulated time in continuous and discrete simulation [40]

2.4 Simulation Modes

Different FSSs can use different modes to perform the simulation. These modes have an impact on the performance and accuracy of the simulation. For example, while Gem5 is limited to sequential simulation [33], QEMU offers the flexibility to be used in both sequential and parallel simulation [41]. Depending on the application and the requirements of the project, one option can be better than another.

According to [42], there are two dominant approaches to advance time within a simulation: asynchronous and synchronous methods. In the first one, each thread communicates with the others in a way that each one can determine when it is secure to execute an event. The number of synchronizations can be reduced by far, although knowledge about the communication behavior of models and deadlock mechanisms, such as roll-back, is required. The second one uses global synchronization times to synchronize all threads. During these intervals, all events scheduled for execution can be parallelized. This approach is simpler but it may incur high simulation overhead.

This section will present the two available modes, the sequential and the parallel. Temporal Decoupling (TD) technique will be explained as well since both can use it in order to obtain more performance. Moreover, all further context will be related to the DES technique and the synchronous approach.

2.4.1 Sequential Simulation

Sequential simulation, or Sequential Discrete Event Simulation (SDES), is the simplest and most accurate simulation mode. It executes the workload sequentially, that is, each event executes at its simulation time, resulting in a perfect simulation without any error.

Going into detail, the simulator runs the process corresponding or sensitive to an event that should be executed at a specific timestamp. This process will run until a certain time when another event must be executed, and it is not related to the process in execution. In this exchange, there is a CS, where synchronization occurs with the rest of the system. All variables are updated to ensure that when a process reads or writes a variable, it accesses the state of the variable as it would be at the current simulation time. The coming image shows a sequential simulation with two different processes.

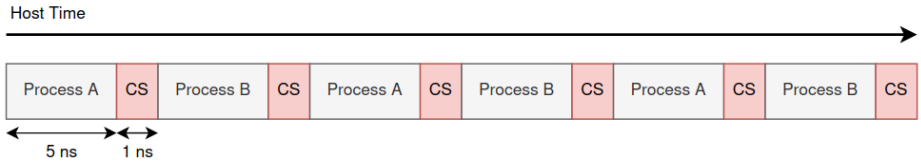


Figure 2.18: Example of a sequential simulation

Imagine a scenario with two processes. One process writes values in the memory and the other reads. Each operation takes five nanoseconds, and each CS takes one nanosecond. If the simulation lasts for one minute, ten seconds would be just for CSs. It is visible that a lot of simulation time is spent here, and it tends to get worse if more processes are needed. The overhead of the event scheduling and process context switching become the dominant factor in simulation speed, leading to a huge host time.

2.4.2 Temporal Decoupling

TD [43] is a technique used by SystemC, which is a standard C++ class library for system and hardware design, to improve performance. It is a method where individual processes are permitted to run ahead in a local time, without actually advancing simulation time, until they reach the point when they need to synchronize with the rest of the system. This time slice, from the beginning of the execution until the synchronization, is called quantum or quanta. The usage of TD may result in a faster simulation in some cases since it increases the data and code locality while reducing the scheduling overhead of the simulator. Figure 2.19 shows the application of TD to the previous example.

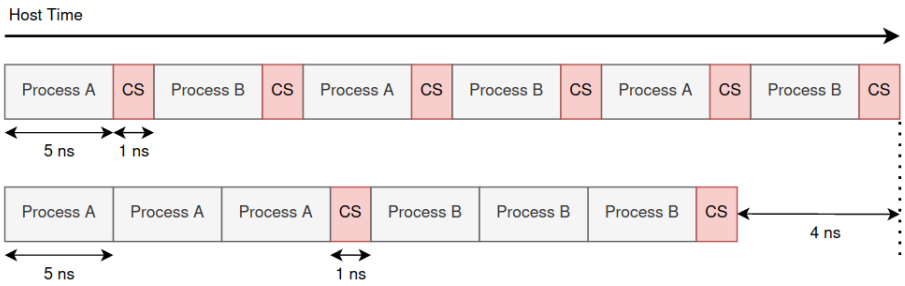


Figure 2.19: The principal of temporal decoupling

Now, for the same process execution time, the host time was smaller ensuring the existence of less CSs. In this example, the reduction was four nanoseconds (four CSs), about 11% of host time, compared to the approach without TD. It can be even higher if more processes and CSs were being simulated.

As mentioned earlier, the process is permitted to advance beyond the current simulation time until it requires interaction with another process. This interaction could involve reading or updating a variable that belongs to another. At that moment, two things can happen: either accessing the current value and proceeding, sacrificing accuracy, or requesting synchronization and resuming the process when the simulation time aligns with its local time. Proceeding with the current value entails making assumptions about communication and timing within the modeled system. It assumes there will be no adverse consequences when sampling or updating the value either too early or too late. Typically, this assumption holds in the context of a virtual platform simulation, where the software stack is designed to be independent of the underlying hardware timing intricacies.

Each process is responsible for evaluating whether it can progress beyond the current simulation time without compromising the functionality of the model. This is a SystemC characteristic because it guarantees functional congruency with the standard semantics of the simulator. Thus, it is not a mandatory feature to implement in other situations.

Nevertheless, a problem can occur if the process does not respect the previous rule and runs with no restrictions. Other processes will not be able the run, compromising the system's functionality. A solution is to define a global quantum that forces the synchronizations. This global quantum, in the original version of SystemC, is static, which means it is defined by the user before the simulation and never changes. Forcing the synchronization results in another problem which is the trade-off between speed and accuracy. A small global quantum guarantees that the processes are using always the updated values and that the simulator does not crash, therefore, the accuracy will be high. However, the simulation speed will be sacrificed as more CSs are occurring. On the opposite side, if the time slice is big, it means that the system might introduce timing inconsistencies, which can lead, in the worst case, to a crash in the simulator. Hence, this value must be chosen carefully in order to have a fast yet accurate simulation.

The SystemC reference manual [43] enforces that some processes cannot be temporally decoupled due to their characteristics, as they might become a bottleneck in simulation speed.

From here onwards, whenever the word "quantum" or "quanta" is mentioned, it refers to global quantum.

2.4.3 Parallel Simulation

In the parallel mode, or in a PDES, as the name suggests, the simulation uses more than one simulation thread in order to have real parallelism. In consequence, the host time will be smaller compared to the sequential mode, since multiple processes can be running at the same time. The next figure shows the speed difference between the two modes.

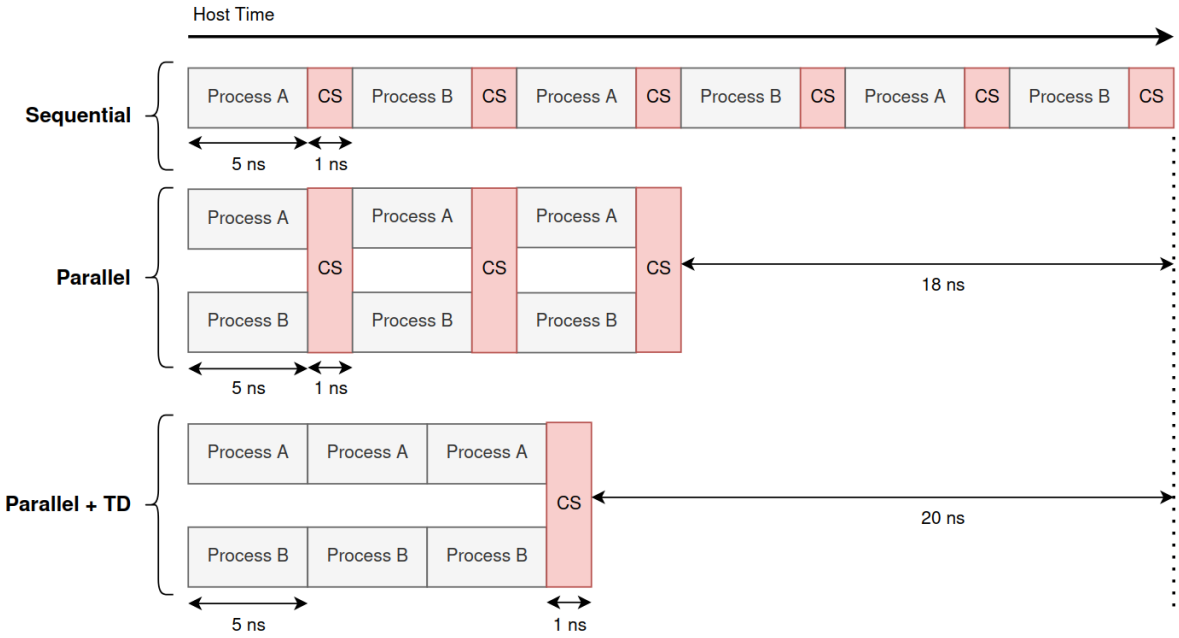


Figure 2.20: Sequential VS parallel simulation with and without TD

Continuing the previous scenario, with the parallel mode, the simulation can be fully optimized in a way that each process can have its execution thread. For this reason, the performance of the simulation increases a lot. In this case, for the same number of executed processes, the parallelized version was able to reduce eighteen nanoseconds of host time, which means a reduction of 50%. With TD, the gain was pushed further when compared to the approach without it. However, it does not only offer advantages due to the several challenges that make it difficult to carry out a PDES. Typical problems are load imbalance, limited parallel work, and causality errors [44] [45].

When working with tasks there is always the problem of balancing the workload among them to make an efficient use of today's multicore computers. An inefficient distribution may create performance bottlenecks. Take as an example a system that is being programmed to read two different GPIO ports, do a logical XOR between both, and write in another GPIO port the result (to turn on a Light-Emitting Diode (LED)) and send it to the terminal. This workload results in four events, which will be assigned to two

processes. If process A is attached with only one event, e.g. sending the information to the terminal, process B will be overloaded, and vice-versa. This issue is easy to solve in this simple example, but in programs with thousands Lines of Code (LoC), it is a challenging job. [46] and [47] are examples of works that help to detect and measure the work imbalance.

It is rational to think that, after observing Figure 2.20, more parallelism will provide an even faster simulation, but this is not true. Normally programs can be parallelized until some point, like in the previous example. It is possible to simulate this system with four cores, however, it is clear that extra cores will not execute anything because there are no more processes to run. For this reason, the performance will not be improved, and the opposite can happen, that is, it becomes even worse [48]. Some works identify this scalability issue to show the programmer how it can be improved [49] [50].

The last problem arises when there are causality errors. Causality errors happen when one event in the future affects an event in the past. Sequential simulation ensures that event B will execute after event A if the timestamp of event A is smaller than the timestamp of event B. In this mode that cannot happen, and if event B tries to change the state variables used by event A, a causality error happens. The following image illustrates the previous scenario, with events colored in green denoting "in execution," and events in gray indicating "scheduled".

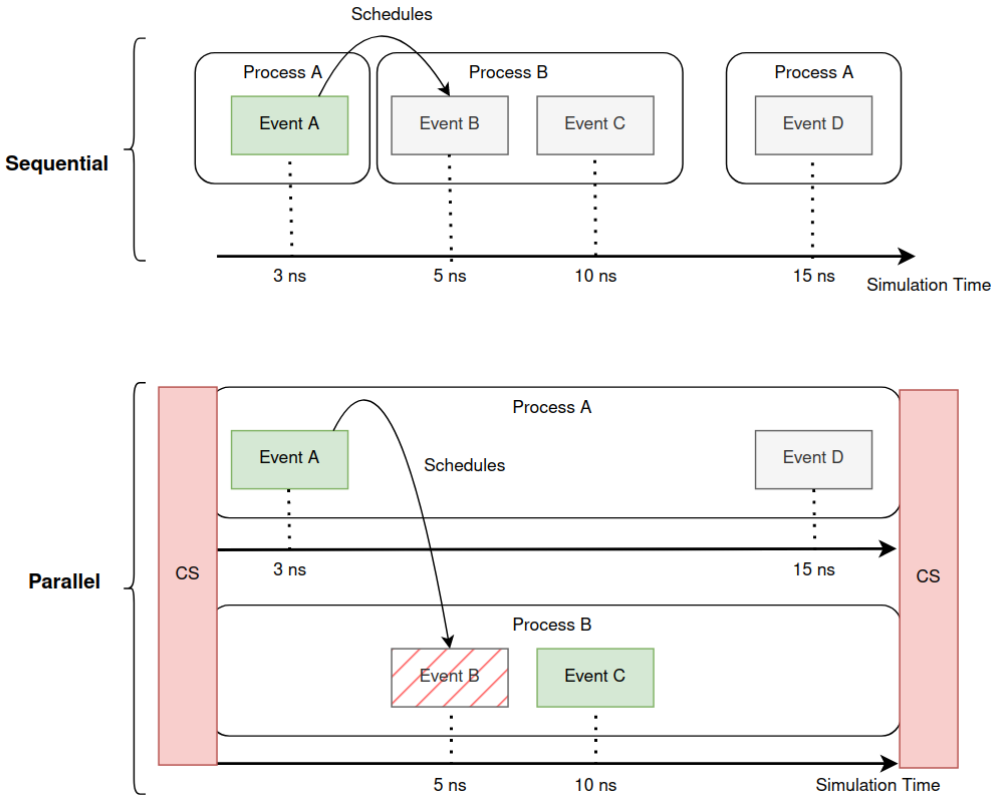


Figure 2.21: The causality problem

When the simulation starts, event A is executed. Meanwhile, it triggers event B, which has an event timestamp of five nanoseconds. In the sequential mode, since it executes one event at a time, this is not a

problem. In the parallel version, it can be, because each execution thread has its own time. It can happen that one thread is ahead of another, e.g. thread 1, which is executing process A, is at three nanoseconds, and thread 2, which is executing process B, is at ten nanoseconds. When event A schedules event B to five nanoseconds, this event will not be executed for the reason that a DES cannot execute events in the past. On top of that, as mentioned before, if event B modifies the state variables of event C, the problem arises.

The usage of TD in both modes may increase the occurrence of causality errors. These errors might become more prevalent if the selected quantum is higher than optimal. Take the coming scenario as an example. A simulator executes a benchmark that writes values into a DAC to play a song in an analog speaker, and, at the same time, solve complex mathematical equations to perform that audio. Each task is associated with processes A and B, respectively. Process B schedules events to write the values on the DAC into process A. Supposing the wanted output frequency was the gold standard audio (44.1 kHz) [51], the associated event needed to update the GPIO port every 2.27 microseconds. If the defined quantum is higher than that value, the event may not be executed at this event timestamp. Hence, the causality error was caused by the chosen quantum time. In this particular example, the consequence would be the production of audio that differs from the expected output. However, in other cases, these errors can break the system functionality and crash the simulator.

As peer with Fujimoto in [52], there are two mechanisms to solve the problem: optimistic and conservative methods. In the first one, causality errors are not prevented at all, although they use detection and recovery strategies to correct that. The correction can be done with a rolling back method, which returns the simulation to a point before the causality error and sets the tighter synchronization. A problem with this method is the performance cost, because of this rewind in the simulation. The second option, on the other hand, avoids the possibility of any occurrence of these errors. An evaluation is done on the event, thus, it can be identified if the event is safe to process, that is, it is ensured that all events that should influence the given event have been processed before its execution. Some works with optimist approaches are [53] and [54]. For [55] and [56] are used conservative methods.

2.5 Gem5

One available FSS used in academia and industry is the Gem5 simulator [33][57]. It had its roots in 2011, from the merge between M5 [58], and the multifacet General Execution-driven Multiprocessor Simulator (GEMS) toolset [59]. With this combination, it is possible to use the best of both tools. The effective support of multiple Instruction Set Architectures (ISAs) and the cache memory and cache coherence models. Also, it is the result of the work between AMD, ARM, HP, MIPS, Princeton, MIT, the Universities of Michigan, Texas, Wisconsin, and many other institutions.

In addition, Gem5 came to overcome a demand in the market at the time. The main highlights are the flexible tool for researchers to evaluate their design in several different ways; Licensing terms that allow

researchers and academia to work together without the pressure of revealing the proprietary information, in the industry case, or not getting credits for their contributions; Defined code style that ensures the code quality remains good so that new collaborators understand faster and better [33].

This section will explain in detail what are its capabilities and how it can be used. Then, the work done by the ICE team of the RWTH Aachen University will be presented, which will be the reference work for this dissertation. In conclusion, other simulators will be explored to gain a comprehensive perspective of the options available in the market and to understand the scenarios where one may outperform another.

2.5.1 Overview

Gem5 is a DES platform that has as its main goal to be a community tool focused on architectural modeling. It has a set of CPU models and memory systems, and two different system modes, System-call Emulation (SE) and Full-System (FS). Figure 2.22 represents the different simulation configurations regarding the trade-off between speed and accuracy.

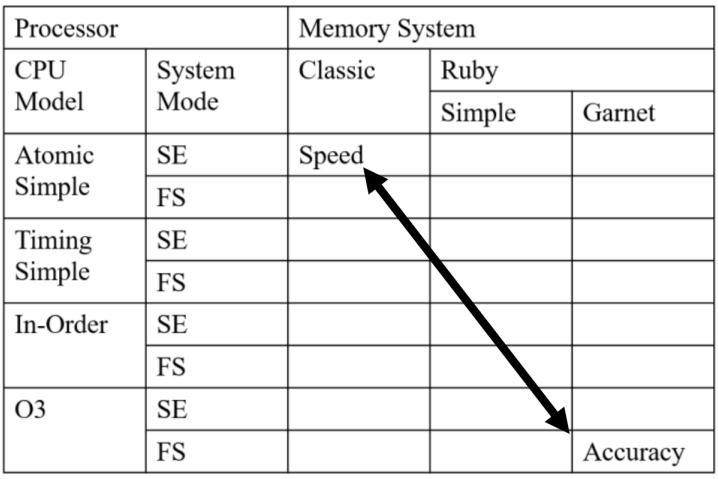


Figure 2.22: Speed vs. accuracy spectrum [33]

As mentioned in the introduction of this section, one characteristic of Gem5 is flexibility. Several simulation configurations allow an adaptation to fit the requirements of the project or the level of wanted detail. For example, a project emphasizing scalability may not require a detailed CPU model.

In this simulator, components can be rearranged, parameterized, extended, or replaced easily to suit the project's needs. This is possible thanks to its pervasive object-oriented design, with the mix of Python and C++. All major components, like the memory, are SimObjects and share common behaviors for configuration, initialization, statistics, and serialization. Moreover, a user can create a customized SimObject, increasing even more the flexibility. To do that, it is required to define the SimObject's parameters, such as instantiation, naming, etc., and define its behavior.

Another remarkable characteristic is the support of different ISAs, counting with Alpha, ARM, SPARC, MIPS, POWER, x86, and, recently, RISC-V. Currently, not all possible combinations of ISAs and other components are known to be functional, as demonstrated in the Table 2.2

Table 2.2: Overview of the supported architectures in Gem5 [60]

ISA	Level of ISA support	Full-system OS support
Alpha	High	Linux
ARM	High	Linux, BSD, Android
MIPS	Low	None
Power	Low	None
RISC-V	Medium	None
SPARC	Low	None
x86	Medium	Linux, BSD

Furthermore, it is also possible to simulate more than one computer system in various ways. It is done by instantiating another system and connecting it via a network interface, creating a client/server pair that communicates with the TPC/IP protocol. The results of the simulation remain deterministic, in other words, a particular input will produce always the same output.

Gem5 uses other tools to perform specific jobs. The most important ones are the pybind11 [61], a lightweight header-only library that exposes C++ types in Python and vice versa, and SCons [62], an open-source software construction tool that orchestrates the construction of software in a way that solves several problems compared to the other build tools. With the SCons scripts present in Gem5 it is possible to build different binaries for different purposes.

- **Debug:** This mode has no optimizations, and, thus, it is the slowest one. It is mostly used when the opt version does not provide enough detail in the debug session.
- **Opt:** It has most of the available optimizations but still with same debug information. Compared to the debug version, it is much faster.
- **Fast:** The fastest binary available. All optimizations are on, and there are no debug symbols. In addition, asserts are removed, although panics and fatals are still included. This binary should only be used when it is desirable for maximum performance, and the code is very unlikely to have bugs.

2.5.2 Simulation Capabilities

As shown in the Figure 2.22, this simulator provides different features that should be used regarding the project requirements. Those are the support for different ISAs, CPU models, and execution modes.

Also, it has the capability to model multiple systems at the same time, supports multiple devices, offers two different network models, and provides different cache coherence protocols [33].

While the event queue is responsible for executing its associated events, the CPU is answerable for scheduling the events. Thereby, it is possible to keep running the simulation even if no CPUs are working. There are four types of CPU models: atomic simple, timing simple, in-order, and Out-Of-Order (O3). The atomic and timing models are the simplest ones. The main difference between them can be seen in the figures below. Meanwhile the first completes all memory accesses immediately, which makes it a proper option for tasks like fast-forwarding (a technique used to warm up micro-architectural state), the timing mode models the timing of memory accesses, providing a more accurate simulation.

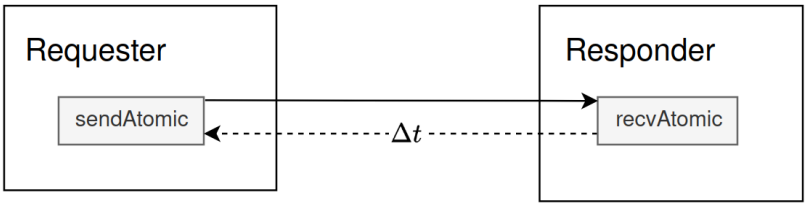


Figure 2.23: Atomic mode

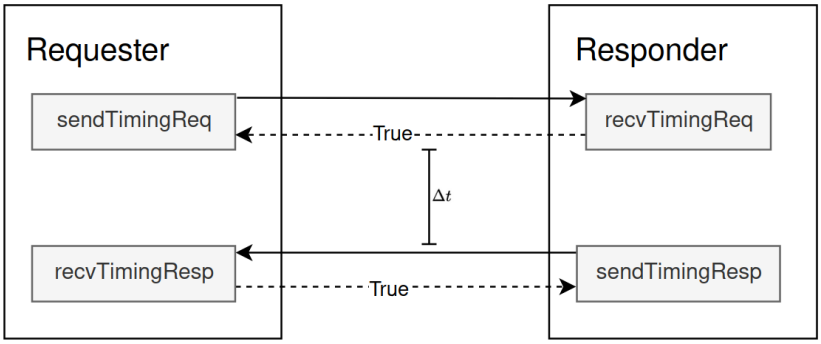


Figure 2.24: Timing mode

The in-order and O3 models are "execute-in-execute" models, which means that instructions are executed only in the execute stage once all dependencies have been resolved. For this reason, those are highly accurate.

Gem5 can operate either in FS and SE mode. The FS is more complex and complete, in a way that it supports interrupts, exceptions, I/O devices, and so on. As the name suggests, the SE simulates system calls, like write(). This characteristic can be a bottleneck to multi-thread applications, but results in a faster simulation. If the workload has lots of I/O iterations or OS services, the FS is the proper choice, otherwise the SE may be considered, even because some ISAs do not support the FS mode yet.

Another important capability is the support for the classic and Ruby memory systems. On one hand, the classic system is fast and easy to configure. On the other hand, the ruby system is more flexible

and accurate. Gem5 also offers an extension for the simple version named Garnet. At the time, Gem5 supports HeteroGarnet or Garnet 3.0 which brings improvements compared to the previous version, for example, it can enable accurate simulation of emerging interconnect systems.

Even though Gem5 is a very flexible simulator, according to [33], there are other capabilities that the developers want to include. A first-class power model, full cross-product ISA/ CPU/memory system support, and parallelization are some examples. It is evident that there is still much work to be done, but compared to the first review, there are lots of advances and improvements, like the support for the RISC-V architecture and the Kernel Virtual Mode (KVM) [57].

2.5.3 Usage

To start using Gem5 it is necessary two things. First of all, the simulator's binary for the wanted ISA. As mentioned before, there are three different types of binaries, therefore, they should be chosen mindfully. Then, it is mandatory the definition of the system. It describes the components comprising the System-on-a-Chip (SoC) and details their interconnections. Gem5 provides in its tutorials the simplest system that a developer can use, and it is presented in the following picture.

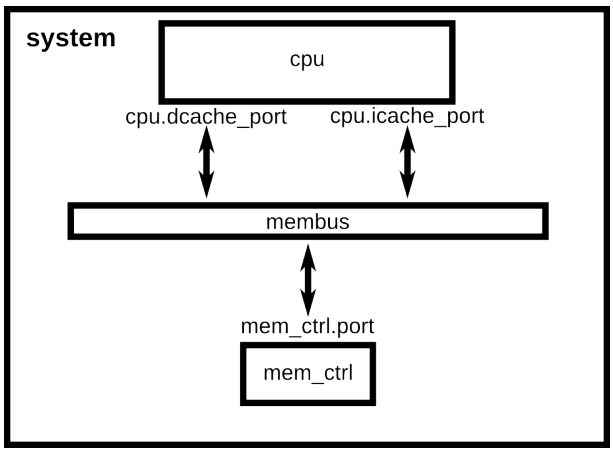


Figure 2.25: Simple system configuration diagram

All configuration is performed in Python scripts, while the functionality of the simulator is implemented in C++. Making changes to the latter requires creating a new binary, whereas in the former case, it does not. The next code demonstrates the instantiation required in the Python script to execute the simulation.

```
#create the system to simulate
system = MySystem ( opts )

#set up the root SimObject and start the simulation
root = Root(full_system = False, system = system)
m5.instantiate()

#instantiate all objects above
print("Beginning simulation!")
exit_event = m5.simulate()

#After execution
print('Exiting @ tick {} because {}'
  .format(m5.curTick(), exit_event.getCause()))
```

Code 2.1: Script to instantiate and execute the simulation

Finally, to run Gem5, the command should first have the directory of the binary, and, then, the directory of the script. Furthermore, it can have more arguments to specify the system or the simulation modes, e.g. the CPU model.

2.5.4 Par-gem5

A huge problem of Gem5 is its speed. At the moment, its official version only offers a single-threaded simulation. Even with the best CPU or the fastest memory on the host machine, the simulation can not get close to one Millions-of-Instructions-Per-Second (MIPS), which results in long execution times. For instance, while the SPEC2017 integer benchmark can take ten minutes in a host computer, with Gem5 the same workload can take more than two years [4]. As referred in the subsection 2.5.2, parallelize Gem5 is one desired feature to have, however, more than ten years passed and there are no developments on this topic, only the support for KVM.

Par-gem5 [4] arises to solve this problem. It was developed by the ICE team of the RWTH Aachen University, with the collaboration of Huawei. It utilizes the multi-threading capabilities of the host system through a modified conservative, synchronous PDES approach, which allows the execution to be dispatched to multiple simulation threads that run independently from the rest of the system for a time $t_{\Delta q}$ called quantum or quanta. At the time, only dist-gem5 [55], a work that focuses on simulating distributed systems connected via a Network Interface Controller (NIC), has implemented a parallel extension. Nevertheless, its application is very strict, as it can only be used in this type of system. Thereby, Par-gem5 comes to overcome this problem and be a generalized parallel version.

In the initialization phase, each CPU is assigned to a dedicated event queue, and all other objects are assigned to the default event queue (q0). Thereby, the total number of threads will be the number of

cores plus one. This version cannot be classified either as conservative or optimistic for the reason that it allows causality errors to occur.

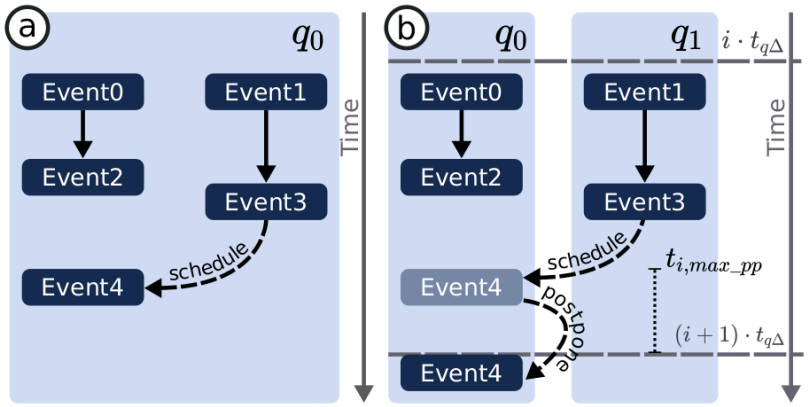


Figure 2.26: Example of scheduling events in Gem5 [4]

Observing the previous figure, scenario A represents the typical sequential simulation, and scenario B the Par-gem5 scheduling method. When an event is scheduled in another event queue, it can either be executed or postponed to the following synchronization. The event is postponed when its timestamp has passed. Consequently, it can lead to a chronologically incorrect execution order of events, affecting the simulation's accuracy, or, in the worst case, flaws in the functionality of the simulated system. Still, in [4] is proven that the inaccuracy can be kept within a single-digit percentage.

The results of this work were very positive. For example, a speedup of 24.7x was achievable without losing accuracy significantly. Moreover, it has been demonstrated that there exists a saturation point in the definition of the quantum. Based on their tests, this point falls between ten and a thousand microseconds, indicating that increasing the quantum beyond this range does not result in performance gains. Another important conclusion was the relationship between cores and inaccuracy. If the quantum does not change, when the number of cores increases, the inaccuracy also increases.

One way to solve the problem is to set a smaller quantum, but getting the perfect quantum is hard. One can be perfect for one benchmark and bad for another simultaneously. Zurstraßen et al. [63] go further and perform a study where this trade-off is evaluated. In the end, it was concluded that the speedup of the simulation as a function of the quantum behaves similarly to a sigmoidal or an "S"-shaped curve, while the simulation inaccuracy, on the other hand, grows linearly. Moreover, for the cases studied, 95% of the maximum attainable speedup was obtained at a quantum between 1000 and 10000 instructions, as different values suit better for different workloads.

Finding a quantum that allows high accuracy with high speedup is one of the main challenges when running a PDES. There are some works that try to explore this problem and come up with solutions, but none of them, at the moment, as shown in the table below, is generalized, real-time, and for PDES simultaneously.

Table 2.3: Overview of the different works regarding the quantum definition

Work	Real-Time	Supports PDES	Generallized
Jung et al. [54] (2019)	X		X
Glaser et al. [64] (2015)	X		X
Jünger et al. [35] (2021)		X	X
dist-gem5 [55]	X	X	

The first work uses a technique that tries to achieve 100% accuracy by rectifying causal errors when they happen, forcing the system to simulate again the faulty part with a smaller quantum. This approach is called a rollback mechanism. One of the best-known methods of this type is the “Time Warp algorithm” [65]. Despite this concept is not new, it is still very present nowadays [53]. Beyond the non- PDES support, a criticism of this method is the performance cost. Because it ”rolls back” the simulation every time a causality error happens, the execution time will be much higher.

The second work uses a Wiener filter to update the quantum within the simulation. This work was able to obtain great performance since it does not require lots of computational resources. Here, there are two simulators, SystemC and Verilog, that communicate between them by a ADC. Hence, the quantum that leads to high accuracy should be equal to the smaller latency. To get that, it assumes a stationary process and model order is known, which sometimes is not possible. Moreover, this work is done in the context of a single-threaded simulation, so there are no IPCs being evaluated.

In the context of PDES, Jünger developed a method whose principle is to classify events as relevant or irrelevant. An event is considered irrelevant to others if its synchronization is not related to them, for example, a timer interrupt. With this information, each local quantum is adapted accordingly to the following event. In consequence of that, it is possible to obtain perfect accuracy, without performance costs. To do this classification, firstly the simulation must be run, and then, with the results, the events can be classified. This makes the technique not so desirable since the user needs to execute twice the same workload.

2.5.5 Other Simulators

Like Gem5, other simulators are used for the same purpose. This subsection will discuss some of them, their advantages and disadvantages compared to the simulator under study, to have a better perspective of what is offered about this subject at the moment.

Starting with QEMU (Quick EMUlator) [32], as the name suggests, it is a versatile full-system emulator that can emulate various architectures, including x86, ARM, PowerPC, and more. As explained earlier in the subsection 2.2.4, there is a difference between an emulator and a simulator, yet QEMU was considered due to other similarities, like the open source availability, the support to different architectures, and the

active development community. José Morales in [66] compared these two simulators in detail and, in the end, the conclusion of his study can be summarized in the next figure.

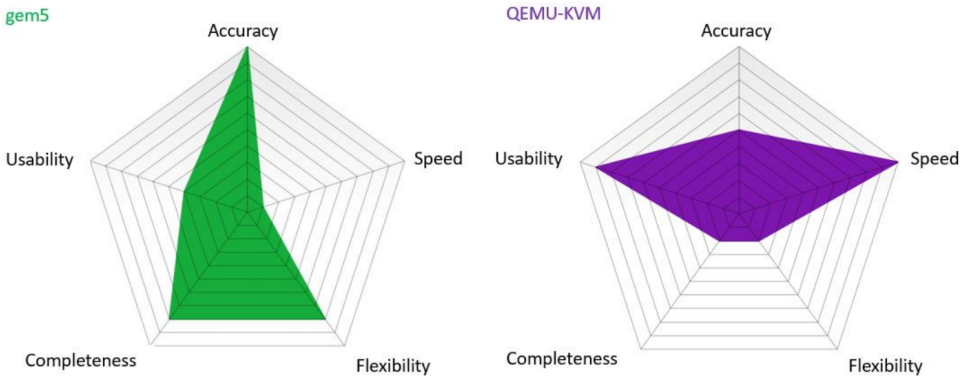


Figure 2.27: Final assessment charts [66]

One characteristic of QEMU is its loosely timed coding style, which provides high simulation speed environments. Cycle-accurate coding style, as Register-Transfer Level (RTL) simulators, on the other hand, covers fine-level Intellectual Property (IP) tuning, which is useful to validate hardware design. Gem5 can be positioned within the spectrum as an intermediate solution, offering a balance between performance and power exploration for early system-level solutions.

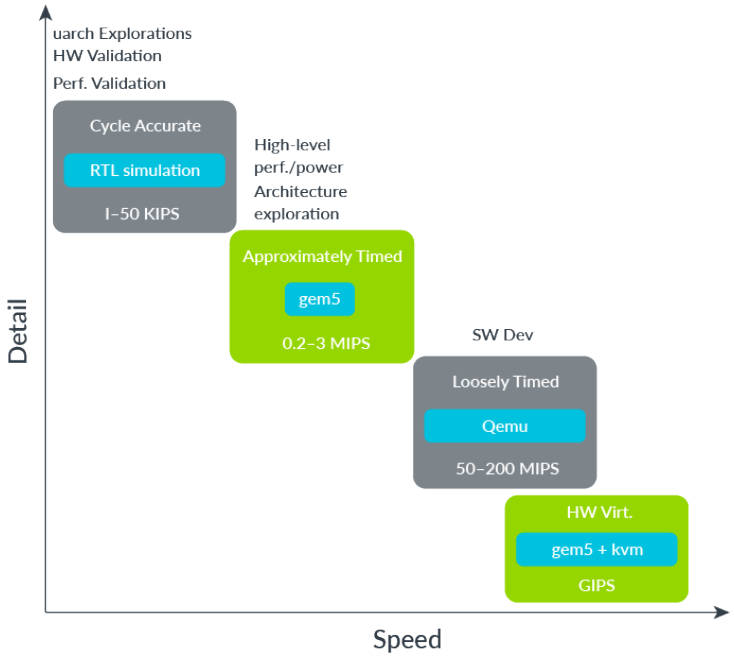


Figure 2.28: Modelling solutions spectrum [67]

Another platform already mentioned is SystemC [43]. It is a set of C++ classes and macros which provide an event-driven simulation interface for system and hardware design. Its main purpose is to provide a C++-based standard for designers and architects who need to address complex systems that

are a hybrid between hardware and software. Furthermore, it can be used also as an HDL, still, it may require an evaluation due to its syntactical overhead compared to other options, like Verilog or VHDL. It also supports Transaction-Level Modeling (TLM)-2.0, which allows us to model the communication as function calls. Events are applied to an entire transaction payload, rather than to individual bus signals, and at the protocol phase boundaries, rather than at clock edges [68]. Thus, the simulations have much better performance, 100-10000 times faster, when compared to the cycle-accurate version. The main advantages are its standardization in the industry, lots of documentation and work, and the capability to quickly simulate hardware and software systems on different levels of abstraction. Contrarily, it is not so flexible, since it requires external modules to complement itself. For instance, it can not perform cycle-accurate simulations, and most of the modules/ IPs are not free of charge, which implies additional costs.

Ayaz Akram and Lina Sawalha developed a work where a comparison of x86 computer architecture simulators is done [69]. In this study, four simulators are evaluated: Gem5 [33], Multi2Sim [70], Sniper [71], PTLsim [72], and ZSim [73]. The authors selected these simulators based on their diverse design approaches regarding the level of detail and abstraction. All of them are modern simulators that are actively being developed, except for PTLsim, which is currently not undergoing active development but is still widely utilized. The following image presents a comparison between the previous simulators.

Feature	Gem5	Sniper	PTLsim	Multi2Sim	ZSim
Platform support	P++	P	P	P+	P
Target support	T++	T	T	T+	T
Full system	✓	X	✓	X	X
Fast forwarding & cache warmup	✓	✓	X	✓	✓
Checkpointing	✓	X	X	✓	X
Trace generation	✓	✓	✓	✓	✓
Details of generated performance stats.	D++	D	D+	D+	D+
Pipeline depth configuration	✓	X	✓	X	✓
Energy and power modeling	E++	E	E	E-	E
In-order pipeline support	✓	✓	X	X	✓
HMP support	M,G,S	S	X	M,G	S
GPU-Modelling	✓	X	X	✓	X
Multi-threaded app. support	✓	✓	✓	✓	✓
Community support	C++	C++	C-	C	C+
Note: [feature's 1st letter]++ is better than [Feature's 1st letter]+ which is better than [feature's 1st letter] which is better than [Feature's 1st letter]- , S=Single-ISA, M=Multi-ISA, G=GPU					

Figure 2.29: Feature comparison between simulators [69]

In their work, these platforms were tested with three different benchmarks. In the end, it was possible to conclude that Gem5 is a good option when very detailed results and multiple ISAs support are intended. Sniper is specifically designed for multi-core simulations and it is considered the most accurate among the simulators examined. However, it lacks the capability to generate detailed performance statistics for the simulated system and it is comparatively less flexible than the other simulators. Then, if the focus is a CPU-GPU architecture simulation, Multi2Sim proves to be a great preference. Finally, PTLsim did not get

a best-case scenario to be used, nevertheless, it is the base of other x86 diverse simulators, such as the MARSSx86 simulator [74].

2.6 Machine Learning

Machine Learning (ML) is at present-day an extremely important subject, encompassing applications in both smaller devices, like watches, and larger ones, such as cars. Autonomous driving is one example of what can be done with machine learning [75]. Although this theme emerged recently with the evolution of computers, it was first defined by Arthur Samuel in 1959, who states that *"it is a field of computer science that gives computers the ability to learn without being explicitly programmed"* [76].

There are various types of ML algorithms, such as classification, error cancellation, and prediction. Depending on the project requirements, these algorithms can be utilized individually or combined in a manner that complements each other to address complex problems, like [75]. The following picture shows some of the commonly used ML algorithms.

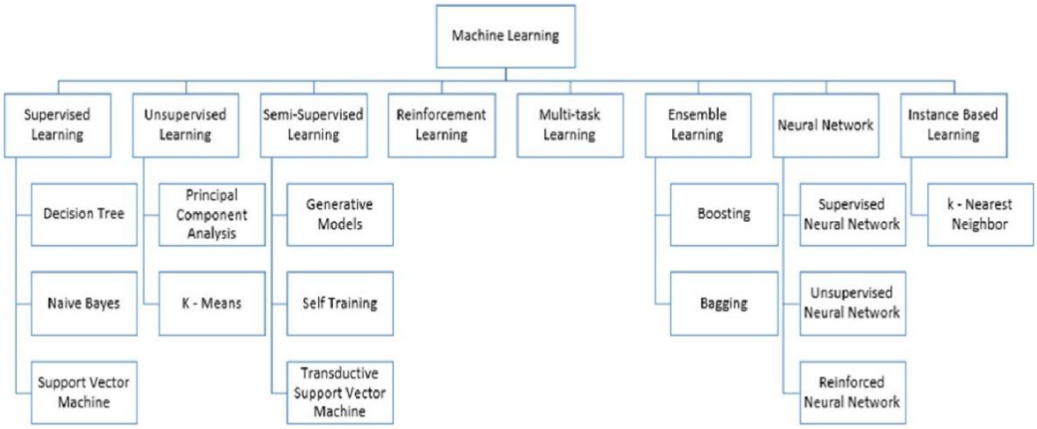


Figure 2.30: Examples of ML algorithms [77]

Within this dissertation, only neural networks will be considered. They can be divided into Artificial Neural Networks (ANNs) and Biological Neural Networks (BNNs). A BNN is a network of neurons that are connected by axons and dendrites, and the connections between neurons are made by synapses. ANNs will be explained in detail in the next subsection. Also, it will be presented how ML can be applied to simulation and the improvements it brought.

2.6.1 Artificial Neural Networks

An ANN is a type of ML that is inspired by the operation of the human brain. They try to model it to implement a particular task or function of interest. To do that, the biological neuron was studied in detail to understand how it works: how the information flows from one to another, how it adjusts the

Table 2.4: Comparison table between a biological and an artificial neuron

BNN	ANN
Dendrites	Inputs
Soma / Body cell	Summing junction and activation function
Synapses	Weights or interconnections
Axon	Output

importance of several inputs, and so on. Figure 2.31 and Table 2.4 represent how a biological neuron can be transformed into an artificial one.

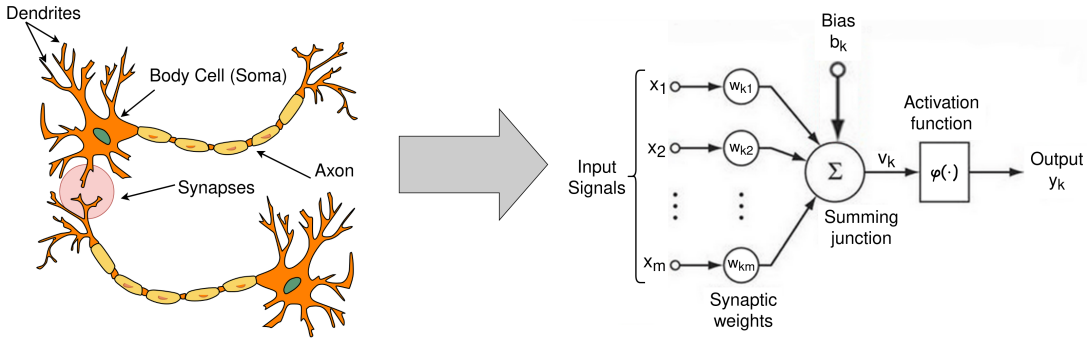


Figure 2.31: Comparison diagram between a biological and an artificial neuron

Like the human brain, ANNs also need to learn. There are three ways to train an ANN, which are also inspired by human actions. Supervised learning is a method that teaches the ANN by providing the outputs for certain inputs. The predicted outputs are compared to the real ones, and, then, weights are adjusted according to the error. In the real world, it can be compared to heating food. The human knows how warm wants his food, but he cannot calculate how long it should be warming in the microwave. Thus, he makes a prediction, and based on the result, time may be redefined.

In the opposite way, unsupervised learning is a technique where the ANN only knows the inputs. The training is done only with the input information, which is a good solution for clustering or density problems, where the data is categorized based on similarities. A great example of this is when a person is tasked to sort apples, where the color can be used to distinguish the apples and perform the sort operation.

The last method is the reinforced NN. There is an agent that controls the outputs of the system. The NN may receive a reward or a penalty for each action it performs. With this information, it adapts its behavior to receive the fewest penalties possible. A comparison can be made to a relationship between a mother and her son, where the mother supervises the actions of her son, and rewards or alerts him when he does something.

Also, ANNs can be classified into two different classes. The Feed-Forward NNs and the Recurrent NNs. In the first one, the information pursuers a linear path, where the output of one neuron will be the

input of another. The other way around, the output of a neuron is used as input of the same neuron, which is appropriate for non-linear applications.

2.6.2 Learning Rules

As illustrated in the Figure 2.31, the inputs, before being added together, suffer an adjustment by the synaptic weights. To define these weights, learning rules are used, that is, self-adaptive equations that update the weights and bias levels of neurons, enabling a neural network to learn from its input data and to enhance its performance. There are diverse types of learning [78]. Some examples are:

- Hebbian learning rule
- Perceptron learning rule
- Delta learning rule (Widrow-Hoff rule)
- Competitive/correlation learning rule (Winner-takes-all)
- Outstar learning rule (Grossberg learning)

For this thesis, only the delta learning rule or Least Mean Square (LMS) algorithm will be considered. It was introduced by Widrow and Hoff in 1960 in [79], and its objective is to minimize the sum of squares of the linear errors over the training set. Additionally, a typical neuron used in NNs is the "ADAptive LInear NEuron" or ADALINE, and its schematic can be seen in the figure below. It consists of an adaptive linear combiner cascaded with a hard-limiting quantizer, which generates a binary output (-1 or 1) used, for example, for classification problems. If it is removed, the output will be analog, which may be useful, for instance, to noise-canceling applications [80]. Further, it is possible to have multiple layers of neurons to address more complex systems (Multiple ADAptive LInear NEuron, or MADALINE).

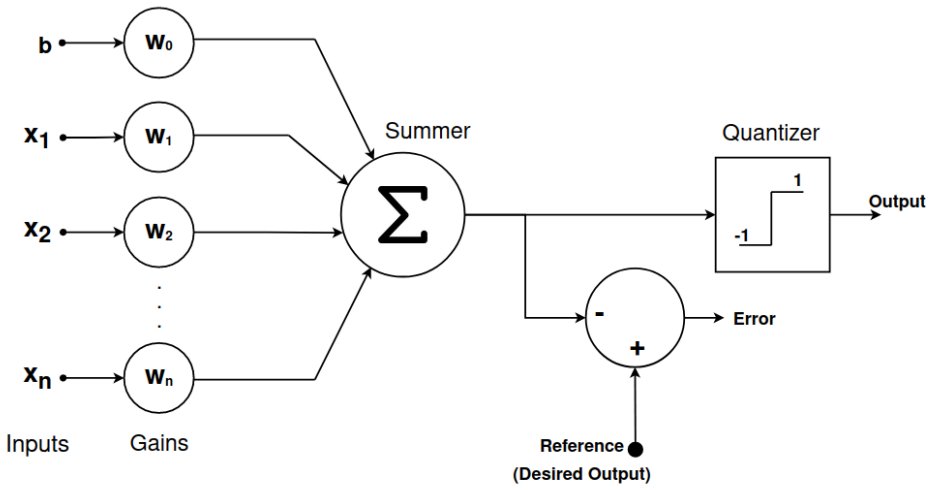


Figure 2.32: Schematic of ADALINE [79]

The linear error is defined as the difference between the desired response and the linear output. After that, this error signal is used to train the NN, regarding the Equation 2.1 [81].

$$Wi_{k+1} = Wi_k + 2\mu\varepsilon_k Xi_k \tag{2.1}$$

Where Wi is the weight associated with the input i, k defines the time, μ is a parameter that controls stability, called learning rate, ε_k is the linear error, and Xi_k is the input i.

2.6.3 Machine Learning in Simulation

ML and simulation are two different topics that can complement each other. For example, when evaluating risk management, where the behavior of the system is based on causal relationships, hidden dependencies, etc. [82], a combination of both worlds could be beneficial. L. von Rueden in [83] presents a work that defines three subfields when combining both, simulation-assisted ML, machine-learning-assisted simulation, and hybrid.

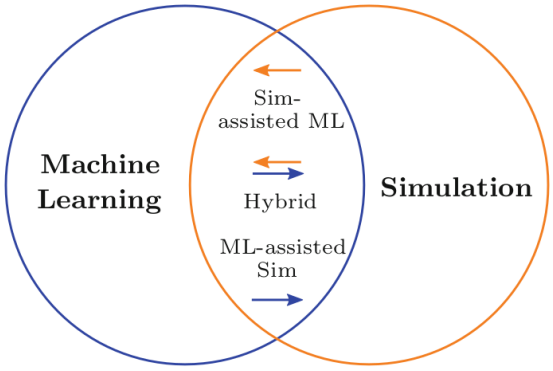


Figure 2.33: Subfields of combining ML and simulation [83]

Machine learning in simulation is often used to detect patterns in data or support the solution process. It can integrate all four components of the simulation. These components are: model reduction [84], input parameters [85], numerical method [86] and simulation results [87]. Because of this versatility, ML is an important part of the simulation world and should be used to obtain better conclusions and performance. Yet, it is not being fully exploited, that is, simulations, especially in the industry, are run with very specific analysis goals, ignoring further analysis that could be interesting for other contexts [83].

2.7 Co-Simulation

The development of truly complex engineered systems that integrate physical, software and network aspects is on the rise [5]. The simulation of these all together is not reliable, hence, the need to divide

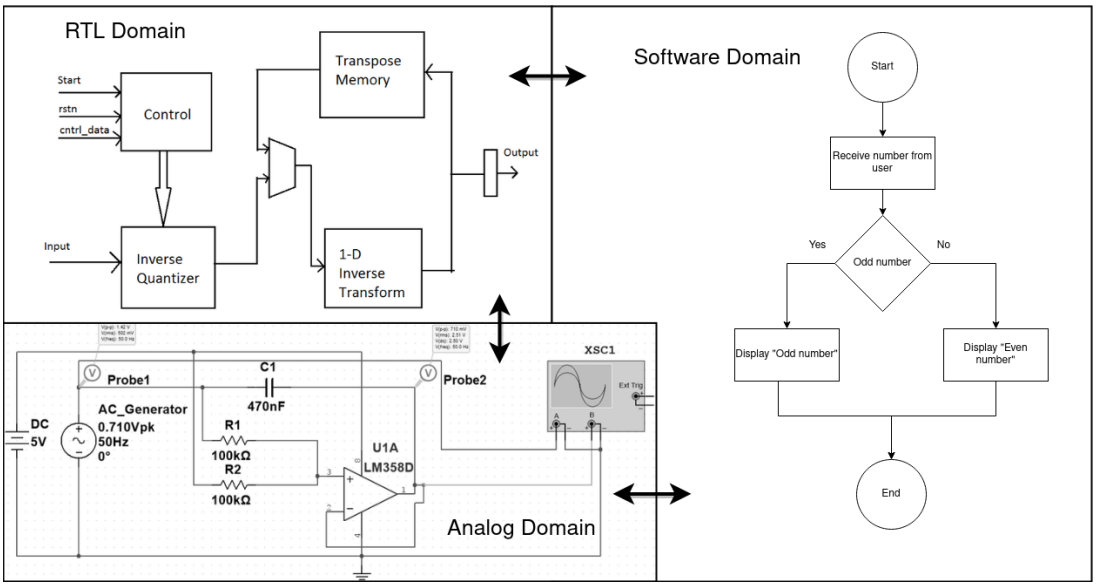


Figure 2.34: Different domains of a complex system

the system into different teams arises. Each one develops a part of the final solution, that, in the end, is integrated into the others. Different abstraction layers may demand distinct simulation tools, as some are finely tuned for specific domains, yielding results that closely mock real-world behavior. The Figure 2.34 shows an example of different domains in a complex system.

Normally, each part is tested and verified alone, that is, without iterations from other modules [5]. However, information on another part of the system may be needed, for example, a wave input to produce a sound. A possible solution can be the generation of local files containing the necessary information but interactions are not thoroughly tested. This may create validation failures, which can result in delays in the development and extra costs in the development.

Co-simulation is proposed as a solution to overcome the latter issue [5]. It consists of a combination of simulators that execute concurrently, with one providing information to the others and vice versa. This approach is used in different domains, as shown in the Figure 2.35. To have the co-simulation environment it is required an Application Programming Interface (API) that creates the interface between the simulators like LibSystemCTLM-SoC, which contains various SystemC/ TLM-2.0 modules that enable co-simulation of Xilinx QEMU, SystemC/ TLM-2.0 and RTL [88]. If no API is provided, the simulator can also be extended to allow such interactions, as long as it is open-source.

One of the challenges when running a DES co-simulation is synchronization. As previously mentioned, there are different iterations of different abstraction levels, which can result in temporal misalignment. In other words, causality errors can happen due to a distinct time granularity. This problem is similar to the one faced when executing a parallel simulation, and the solutions applied to address them are remarkably consistent.

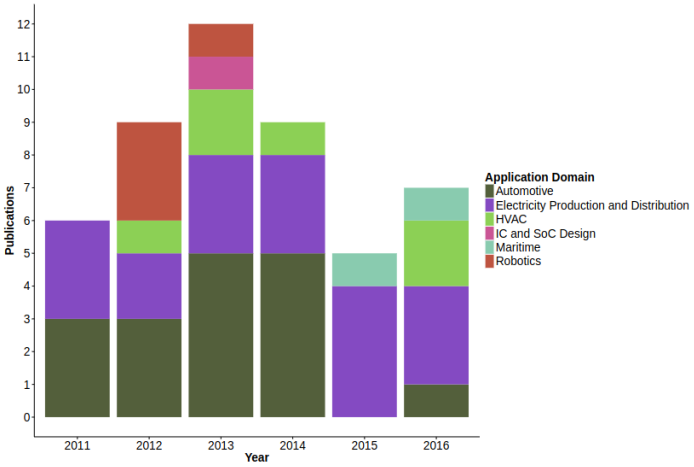


Figure 2.35: Research publications of co-simulation applications over five years[5]

2.7.1 Co-Simulation on Gem5

Until the present date, few works have used Gem5 in a co-simulation environment. The primary reason for this could be that official support is only available for SystemC [57]. However, the open-source nature of Gem5 allows for extensions and integrations of additional tools. An example of that is the work developed by Manu Komalan in [89], where Gem5 and NVMain work together to obtain a good simulation accuracy.

SystemC is being widely used in the industry and research hardware design space exploration [90]. Nonetheless, there is a notable scarcity of precise, freely available, adaptable, and lifelike SystemC models for contemporary CPUs. Christian Menard in [90] developed an API that allows full interoperability between the two simulators, which is now part of the official version of Gem5 [57].

In the paper, there are presented three possible scenarios for the co-simulation environment. The issue is that achieving co-simulation requires compiling Gem5 as a library and, then, using it in SystemC. This does not fully adhere to the previously mentioned co-simulation definition because the two simulators are not running simultaneously in separate processes. This limitation can lead to verification failures as direct inputs from other simulators to Gem5 are not possible.

3 | Dynamic Quantum Extension

The development of Par-gem5 [4] tackled the low-performance issue of Gem5 [33]. However, as exhibited in the subsection 2.5.4, it was achievable by sacrificing simulation accuracy, creating a tradeoff between both. The tradeoff is defined by the quantum value, which, in its current state, is set at the beginning of the simulation and remains unchanged throughout. Introducing a dynamic quantum allows the value to be adjusted during the simulation, making it possible to adapt it to the simulation needs. Concerning the current works on this subject, none fulfills the three characteristics presented in the Table 2.3, therefore, they can not be employed in the Par-gem5.

To overcome these limitations and find the optimal quantum automatically, this dissertation proposes some algorithms to tackle these limitations. During the chapter, these algorithms, which are named ADALINE-based, Step Ladder, Instruction Flow Prediction (IFP), and Loop Detection, will be described and explained in detail. They will operate on the synchronization process, more precisely, in the quantum definition, as presented in the Figure 3.1.

All event queues (EQs) are running independently from each other until they reach the synchronization barrier. Due to their independence, each one has its local time, thus, the processes will not hit the barrier at the same moment. To avoid parallelism issues, the synchronization process is only executed when all have reached the barrier. When starting this process, the simulator verifies if there are more events to carry out and establishes whether the simulation should continue or not. If affirmative, the postponed events are scheduled to their respective event queues and the next synchronization is programmed for the

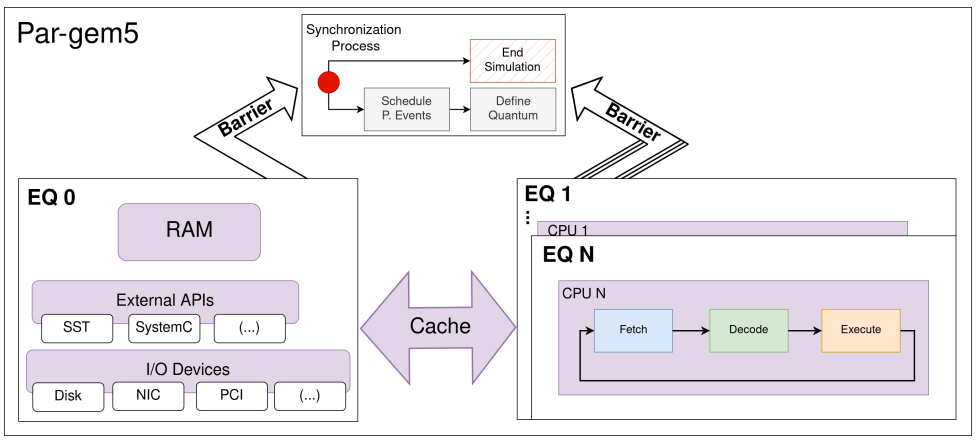


Figure 3.1: High-level diagram of Par-gem5 operation mode

current time plus the result from the developed algorithm. If not, it finishes the workload.

This chapter will start by presenting an introduction to the developed extension, demonstrating the required modifications to integrate the solution and explaining the used benchmarks. After that, the remaining sections will outline the advantages and disadvantages of each algorithm, consistently comparing their performance with the previous scenario. In the end, an evaluation will be done, resulting in the definition of the definitive algorithm. Its results will be compared with the static version already presented in Par-gem5.

3.1 Par-gem5 Changes and Benchmarks

Before going deeper into the algorithms' design, a brief introduction will be given about the changes in Par-gem5 and the used benchmarks to test the algorithms. These changes were done to integrate the usage of the dynamic quantum since the current Par-gem5 version only supports a static mode.

3.1.1 Par-gem5 Changes

In order to modify the simulator to integrate the dynamic quantum, two changes were required. The first one was the addition of a new function mode, and the other consisted of the inclusion of more simulation feedback regarding the dynamic algorithm.

Par-gem5 has two possible simulation modes: sequential and parallel with a static quantum. With the introduction of the dynamic version, users must specify whether the quantum value will be static or dynamic. In the case of a dynamic quantum, no additional actions are needed. If a static quantum is chosen, users can still define the quantum value, otherwise, a default value is selected (1 microsecond).

Concerning this, the *QuantumMode* was created to adhere to the new situation. This variable can have two states, static or dynamic, representing the two available versions. To integrate it into the simulator, a couple of steps were done. First of all, the *QuantumMode* was defined and added to the list of the simulator's available types, as shown in the Code 3.1. After that, a new file was created, Code 3.2, to implement the name translation, due to the fact that the names used on the user side are different from the simulator side. At last, the remaining simulator was adapted to be able to select between the two versions, including the quantum global synchronization event, which is reasonable for the synchronization process.

```
                                # (...) #
class ByteOrder(ScopedEnum):
"""Enum representing component's byte order (endianness)"""

vals = [
    'big',
    'little',
]

class QuantumMode(ScopedEnum):
    """Enum representing quantum available modes"""

    vals = [
        '_static',
        '_dynamic'
    ]
    wrapper_name = 'QuantumMode'

# How big does a rounding error need to be before we warn about it?
frequency_tolerance = 0.001  # 0.1%
                                # (...) #
```

Code 3.1: Snippet of Params.py

```
import sys
from m5.util import warn
from m5.params import QuantumMode

def mode(quantum_mode):

    if(quantum_mode == "static"):
        return '_static'
    elif (quantum_mode == "dynamic"):
        return '_dynamic'
    else:
        raise TypeError("Can't convert '%s' to a quantum_mode" % type(
    quantum_mode))

__all__ = ['mode']
```

Code 3.2: Quantum.py file

At the end of each simulation, Par-gem5 provides a set of statistics that reflect the behavior of the performed simulation. Examples of this are the number of quanta with, at least, one postponed event (*errQuanta*), the absolute error of Par-gem5 in seconds (*errSimSeconds*), and the number of events postponed (*eventsPostponed*). However, to have a better view of how the dynamic algorithm was executed, the *quantumMean* and *DynReset* statistics were included in the final simulation results.

The first, as the name suggests, refers to the mean of applied quantum. During the simulation, the dynamic algorithm may vary the quantum, hence, its average value might be useful to have an idea of how the simulation was conducted. For instance, if the workload was focused on performance and the *quantumMean* gives a small value, it might be a sign that something went badly. The development of a dynamic algorithm can be another example, since this data can reveal if, in some tests, the quantum was or was not the most adequate.

The *DynReset* gives information about how many times the dynamic algorithm had to reset. This might happen when the dynamic system becomes unstable, providing illogical quantum values. Later, in this chapter, this concept will be better explained.

3.1.2 Benchmarks

After a project development, it is common to verify if the requirements were accomplished. In this manner, the engineer needs to execute tests or benchmarks to perform the evaluation. They are designed to measure performance, capabilities, efficiency, and other system components. There are different types of benchmarks, each one emphasizing an area of interest. Some examples are: CPU, network, and storage.

In the context of this dissertation, it will be used two CPU benchmarks: the bare-metal bubble sort and the NAS Parallel Benchmarks (NPB). These will be executed on a host and target system with the configurations exhibited in the Table 3.1a and Table 3.1b, respectively.

Table 3.1: System Configurations

(a) Host

CPU	AMD Ryzen 3990x (64 cores, 128 threads)
RAM	128GB of 3200MHz DDR4-DRAM
OS	Ubuntu Linux 20.04

(b) Target

CPU	ARM64, AtomicSimpleCPU @ 2GHz
Cores	2, 4, 8, 16, 32, 64
Caches	64kiB L1-D, 32kiB L1-I, 2MiB L2 shared
Main Memory	2GB of DDR3 RAM @ 1600MHz
Periph. Sub-system	Real View Virtual Express V1

Bare-metal Bubble Sort

The main objective of this benchmark, as implied by its name, is to rearrange an array in such a way that elements with higher values are positioned at the top. The algorithm employed for this task is quite straightforward. It involves a loop that iterates through the input list, examining each element in turn. In

each iteration, the algorithm compares the current element with the subsequent one, and if the latter is smaller, the values are swapped. This process continues until the entire array is sorted according to the desired criterion. It was designed to attain a near-best-case simulation throughput, meaning that thread synchronizations and accesses to shared memory are reduced to a minimum.

NPB

NPB [91] is a group of a small set of programs designed to help the performance evaluation of parallel supercomputers. In total, there are eight benchmark specifications: five kernels, and three pseudo-applications. The kernel benchmarks are:

- **Integer Sort (IS):** It performs a sort operation where both integer computation speed and communication performance are tested.
- **Fast Fourier Transform (FT):** It solves a three-dimensional partial differential equation using Fast Fourier Transforms (FFTs). Long-distance communication performance is the main evaluation point.
- **MultiGrid (MG):** It is a simplified multigrid kernel that requires highly structured long-distance communication. Thus, short and long-distance data communications are evaluated.
- **Conjugate Gradient (CG):** It computes an approximation to the smallest eigenvalue of a large, sparse, symmetric positive definite matrix. The main goal is to test irregular long-distance communication.
- **Embarrassingly Parallel (EP):** It provides an estimate of the upper achievable limits for floating point performance, with no significant inter-process communications.

The pseudo-applications are the **Block Tridiagonal (BT)**, **Scalar Pentadiagonal (SP)**, and **Lower-Upper symmetric Gauss-Seidel (LU)** benchmarks. Each of these benchmarks is designed to solve a synthetic system of nonlinear partial differential equations using a distinct algorithm. The names of the benchmarks correspond to the specific algorithms employed in solving these mathematical problems.

Moreover, NPB has implemented benchmark classes that define the problem size of each workload. There are eight problem sizes (S, W, A, B, C, D, E, and F), where S is the smaller one and F is the bigger one. In the context of this dissertation, only the W size will be considered.

3.2 ADALINE-Based Algorithm

Adaptive filters are widely used due to their capacity to autonomously adjust parameters and operate with minimal or no prior knowledge of the signal [92]. One application of these filters can be in the

development of an Adaptive Noise Cancellation (ANC) algorithm [93]. A generic scheme can be found in the image below.

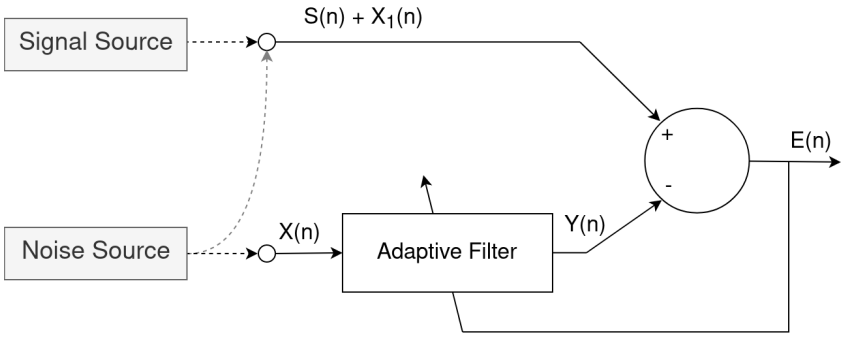


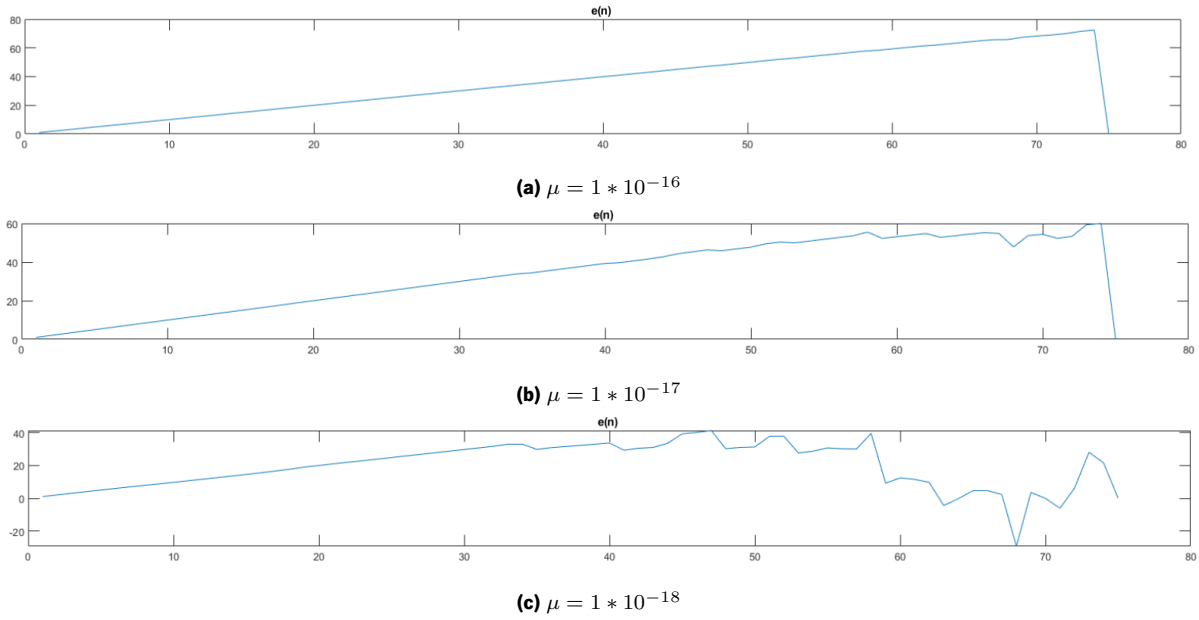
Figure 3.2: ANC scheme

While $S(n)$ is the input that has the signal mixed with the noise, $X(n)$ receives only the noise, hence, it is a noise reference. If the noise in $S(n)$ was the same as the one presented in $X(n)$, it would only be needed to subtract the $X(n)$ to $S(n)$. Nevertheless, this noise is different because it is attenuated, delayed, and filtered by the noise path. For these reasons, an adaptive filter should be utilized, allowing $Y(n)$ to closely resemble $X_1(n)$. $E(n)$ will be the output without the error. It is also used by the filter in order to adapt their weights.

A comparison can be made to the Par-gem5 world. The signal is the quantum, the noise is the simulation error, and the output is the quantum without the error. Also, the $X_1(n)$ and $X(n)$ are different due to the impossibility of calculate exactly $X_1(n)$ in runtime. When a benchmark is running, the simulation error obtained is referred to as the worst-case scenario. In other words, the simulator analyses when there is a cross-schedule event, and the time difference between when that happens and the next synchronization is considered the error, since it is taken into account that the event must be postponed. However, most of the time, it is not true, and the event is not postponed, causing a smaller error. To accurately analyse what was the impact of inaccuracy ($X_1(n)$), it would be needed to run the sequential simulation first, killing all the benefits of Par-gem5. The next equation describes how the worst-case error estimation is calculated.

$$e_{rel,t} = \frac{t_{sim,meas}}{t_{sim,meas} - \sum_{i=0}^{Q} t_{i,max\_pp}} - 1 = \frac{t_{sim,meas}}{t_{sim,est}} - 1 \tag{3.1}$$

In each quantum, the postpone-mechanism records which event experienced the most significant time shift caused by the postponement, t_{i,max_pp}. Q is the number of simulated quanta and $t_{sim,meas}$ is the measured simulation time.



(a) $\mu = 1 * 10^{-16}$

(b) $\mu = 1 * 10^{-17}$

(c) $\mu = 1 * 10^{-18}$

Figure 3.3: Control action with different learning rate values

3.2.1 Adaptive Filter

Following the Figure 2.32, the adaptive filter will be responsible for adapting the weights of the ADALINE NN. It will use the Equation 2.1 for the training, which means that it is essential to carefully choose an appropriate learning rate.

The learning rate is a constant that is chosen at the beginning of the simulation and it is not modified. This parameter compromises a trade-off between control speed and stability. With lower values, the learning process is fast, nevertheless, it is likely to become an unstable control system. With higher values, stability is granted, but the learning process is very slow, taking a lot of time to be operational. Finding the best value is not linear, therefore, an iterative approach was used.

A small simulation was done, and in each quantum synchronization, the quantum values and the error were recorded. Furthermore, in this test, the quantum was incremented 1 microsecond every time a record was done. Then, the ADALINE-based algorithm was implemented in Matlab, obtaining the results illustrated on Figure 3.3.

The aforementioned mention trade-off can be observed, where for $\mu > 1 * 10^{-16}$, the results were similar to Figure 3.3a, and for $\mu < 1 * 10^{-18}$, the system become even more unstable. To choose the learning rate, the logarithmic scale is used, as small variations in the μ do not result in a significant change. From the results, it was concluded that the best value for μ is $1 * 10^{-17}$.

Although the test with μ equal to $1 * 10^{-17}$ did not result in an unstable control system, there is a possibility that it could happen, for example, when there are huge error variations. Typically, these variations occur when the synchronization time is much longer than the ideal case, which requires resetting the algorithm. Regarding the results on [4], where it was observed that the inaccuracy grows with increasing quantum and number of cores, it was also implemented a *safeQuanta*, a value that is smaller than the

actual quantum. It depends on the previously mentioned characteristic, that is, when more cores are being simulated, this value becomes smaller. The next flowchart illustrates how the reset system works.

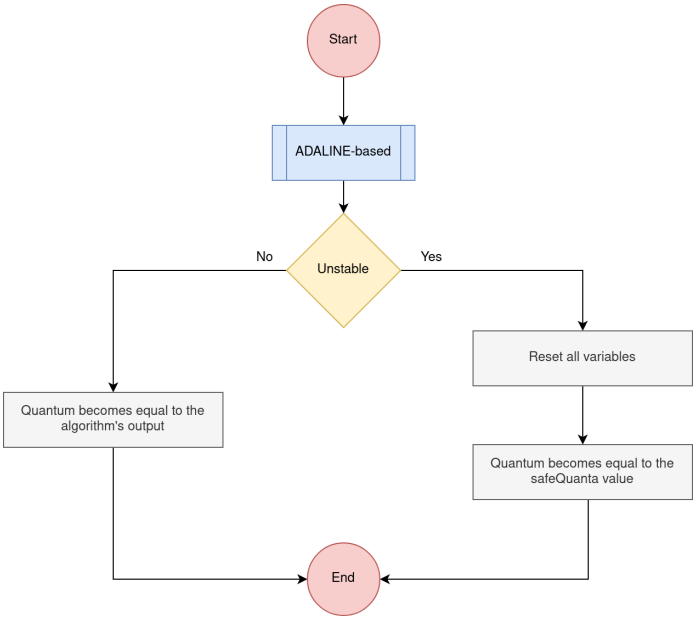


Figure 3.4: Reset system in ADALINE-based algorithm

If the ADALINE output is negative, it indicates an unstable system, requiring a reset. In the reset, all the variables used in the calculation are set into their initial conditions. Although it will sacrifice simulation performance, this action is lightweight, which means a negligible variation. Yet, it is desirable to have the least resets possible.

3.2.2 TDL

As the simulation progresses, there is more data to analyse, taking away performance from the NN. Stella et al. [93] presented one example of an application where this problem is also present. The solution to make full use of the ADALINE network as an adaptive filter was to implement a Tapped Delay Line (TDL). Its operating method is described in the figure below.

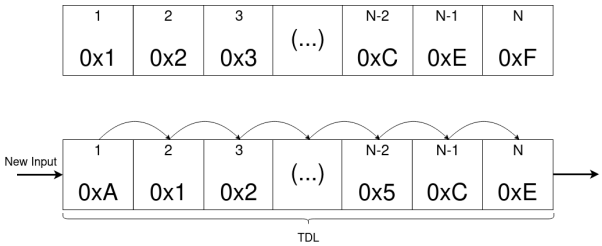


Figure 3.5: TDL working method

One input is considered N times, being N the size of the TDL. When a new input arrives, the oldest one is discarded. In the end, the TDL always has the signal at the current time, the previous input signal,

and so on. Combining the TDL with the ADALINE, efficiency and performance are no longer sacrificed. This approach was implemented in the noise source, since it receives new data in each new quantum evaluation.

TDL size varies from application to application. In this case, when the size is small, the learning process may be subpar, but it also becomes less sensitive to variations in the noise. In contrast, a larger size leads to more effective weight adjustments through improved learning, but it makes the NN more susceptible to noise variations. To choose the best size for this scenario, a small test was done where different TDL sizes were utilized. The results are presented in the following graph.

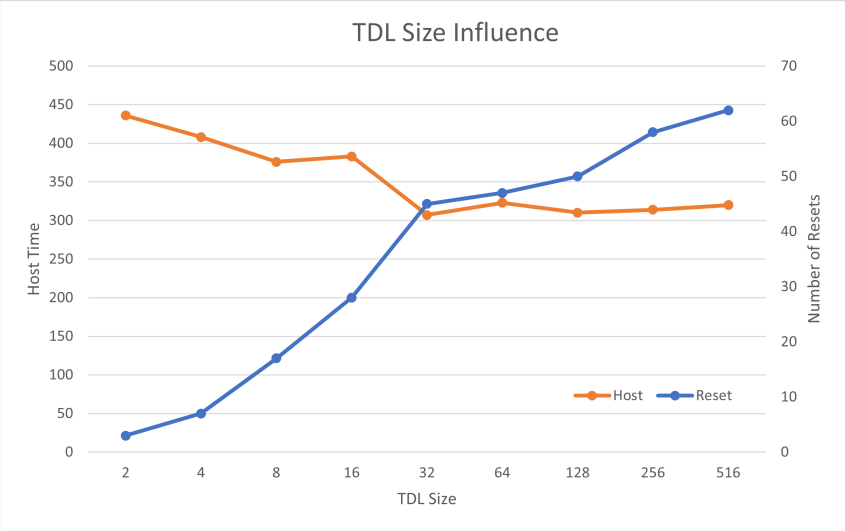


Figure 3.6: TDl size influence

To have maximum performance on the simulator, only base 2^n sizes were chosen, where $n \in N$. Observing the outcome, it can be seen that there is a point where the host time does not improve significantly, however, the number of resets continues increasing. As previously noted, each reset has a performance impact, as it requires resetting every variable to its initial conditions. Therefore, it is desirable to minimize the number of resets whenever possible. In conclusion, the best size to be used is 32 and, for this reason, it was used for the subsequent tests.

3.2.3 Quantum Increment

The ADALINE-based algorithm output, that is, the filtered quantum, will always be lower than the source. Using this new quantum implies that there will be a point in time when it becomes too small because it fails to increase its value. This situation inevitably leads to a trade-off, wherein performance is compromised. To strike the optimal balance, it becomes necessary to increment the quantum value. This adjustment should occur when no errors are detected in the system. Figure 3.7 presents the flowchart for the ADALINE-based algorithm.

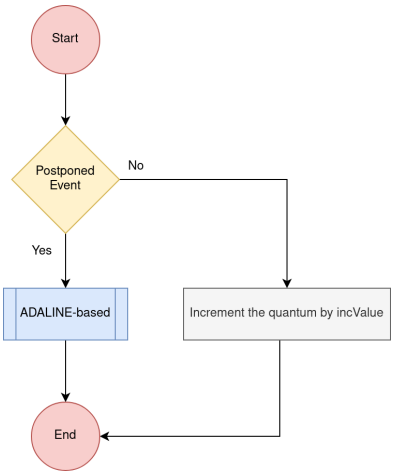


Figure 3.7: ADALINE flowchart

When an event must be postponed, it is considered that an inaccuracy problem occurred, indicating that the quantum should be reduced. In the synchronization state, the first step is to verify the preceding scenario and determine whether the quantum is incremented or the ADALINE-based algorithm takes action.

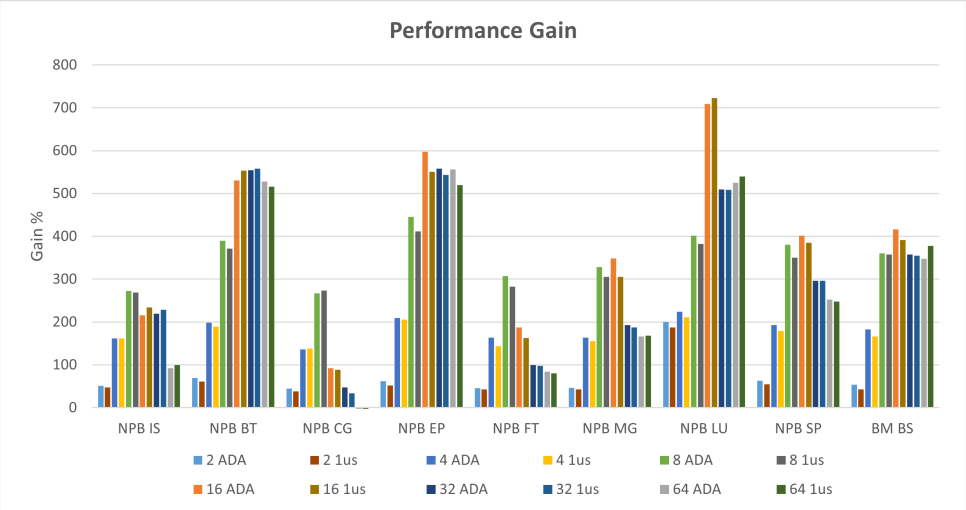
The increment value should have a balance, in the way that, with a smaller increment the performance may not be improved, and with a high value the accuracy may be ruined. According to [4], the quantum of 1 microsecond gives a good trade-off between those two, thus, it was chosen to use an increment 10 times lower. This value enables a significant balance without causing any harm.

3.2.4 Results

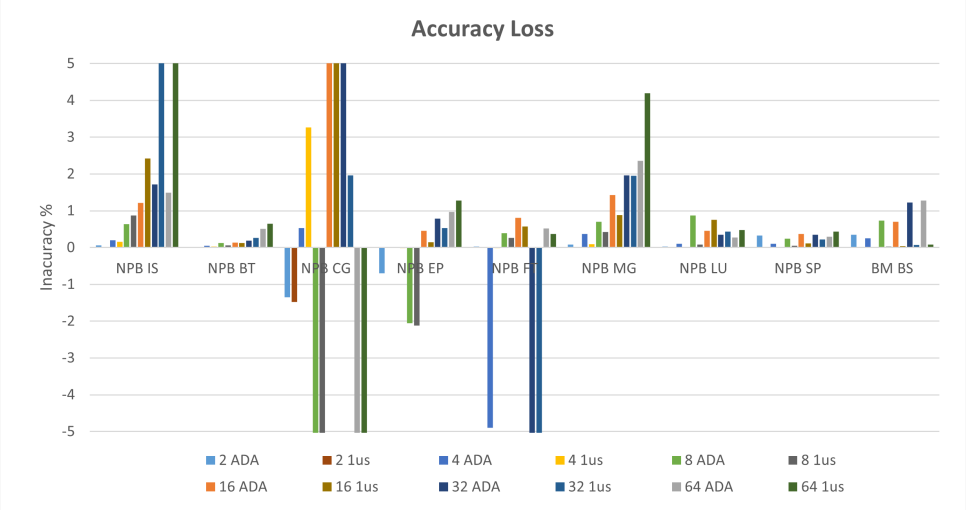
The aforementioned benchmarks executed the developed algorithm with different numbers of cores. The results obtained are depicted in the Figure 3.8.

The horizontal subtitle illustrates the executed benchmark along with the number of cores and the corresponding algorithm used, hence, 4 ADA means four cores were simulated with the ADALINE-based algorithm. To assess the ADALINE performance, the static version was also executed with a quantum set to one microsecond. In all cases, the comparison reference is the result obtained from sequential simulation. Additionally, the results for a single simulated core are not presented because, in all conducted tests, the sequential simulation exhibited similar performance. Hence, this scenario is not relevant for evaluating the algorithms.

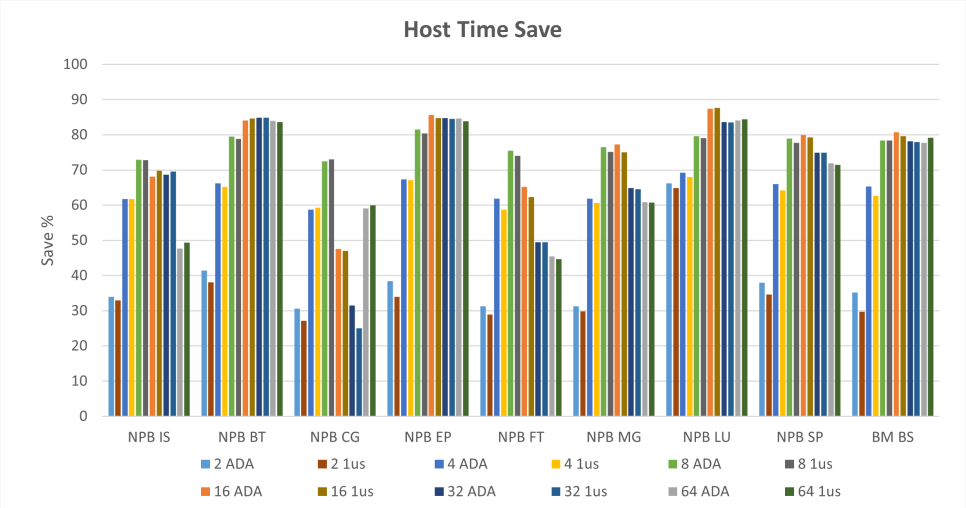
When it comes to performance, it is noticeable that the algorithm yielded better results in certain benchmarks, such as bare-metal, NPB EP, and SP. However, in other cases, like NPB IS and CG, it did not show significant improvements. Shifting the focus to accuracy, the static version generally outperformed the dynamic approach. Nevertheless, it systematically maintained an accuracy of under 2%, with the exception of some cases, like NPB CG, where both approaches had a poor performance. Lastly, when comparing the host time to performance, similar outcomes were observed, though certain benchmarks exhibited superior results to others.



(a) Performance gain



(b) Accuracy lost



(c) Host time save

Figure 3.8: ADALINE-based algorithm results

Concerning the Figure 2.26, it can be stated that the target time obtained with the parallel mode can only be higher than the one obtained with the sequential mode. Two reasons justify the previous sentence. The first one is related to TD, where its use may cause a higher target time due to the obligation of only ending the simulation in the synchronization process. The second one is connected to postponed events, since these may delay the simulation. For these reasons, inaccuracy can only be positive. Nevertheless, as shown on the graph, this is not the case. It is worth mentioning that these occurrences are not related to the algorithm itself but rather to Par-gem5, as simulation with the static version also evidences this dilemma.

The performance issue may be related to the previous quantum increment consideration. It is clear that this value for some workloads is good, but for others, it may not be the most adequate. Therefore, having a dynamic value would solve this problem, resulting in a more flexible algorithm. Moreover, special attention should be given to its design, as excessively large values can lead to an unfavorable trade-off.

3.3 Step Ladder Algorithm

As shown, it was found in some cases that the performance was equal or lower when compared with the static approach. Concerning the quantum increment, a fixed value was defined following the aforementioned philosophy. However, a dynamic approach may bring better results, as different benchmarks require different needs.

Therefore, the ADALINE-based algorithm was updated to integrate this new functionality. Observing the Figure 3.7, the calculation of the increment value (*incValue*) will be done before the *if* statement, so whether there is a postponed event or not, the *incValue* is always controlled. The new flowchart is represented in the next figure.

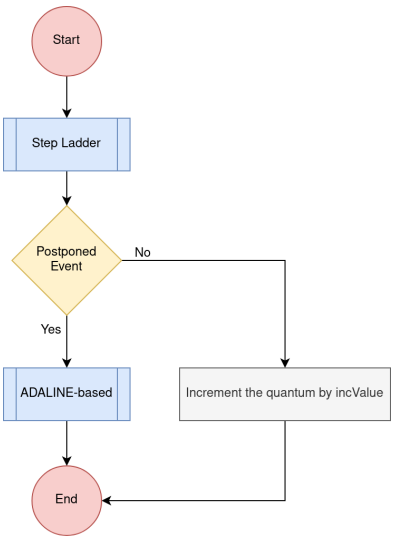


Figure 3.9: ADALINE-based and Step Ladder algorithms combined

The main idea of this algorithm is to gradually increase the *incValue* when there are no postponed events. It can be seen as an exponential, however, an approach like this would scale too fast, affecting the accuracy. A solution can be the definition of a threshold, where the increment value only increases if no "accuracy issues" have happened N times in a row. An example is illustrated in the Figure 3.10. It is possible to observe that there is a point, marked in red, where there was no variation in the *incValue*, indicating the occurrence of a postponed event.

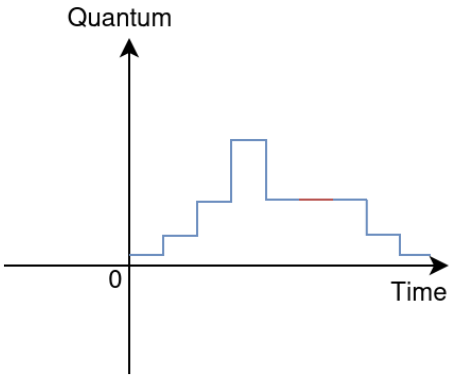


Figure 3.10: Example of the increment value evolution with a threshold

The *incValue* starts in the smaller value possible (*minValue*), which is equal to the clock period, and can grow until a maximum (*maxValue*) of a hundred microseconds. This value was defined regarding the results on [4] and [63], where the benefits of a quantum of this magnitude are few. The addition method respects the subsequent equation.

$$incValue = 2 * incValue \tag{3.2}$$

Oppositely, the decrement is done by dividing by two the actual *incValue*, until it reaches the minimum. The threshold value depends on how many cores are used by the simulation since the inaccuracy grows with the increasing number of cores. The more cores are being used, the greater the threshold, making it harder to increase the *incValue*. Moreover, the increment value only changes (increases or decreases) if it fulfills the condition for N consecutive times, as shown in the later image.

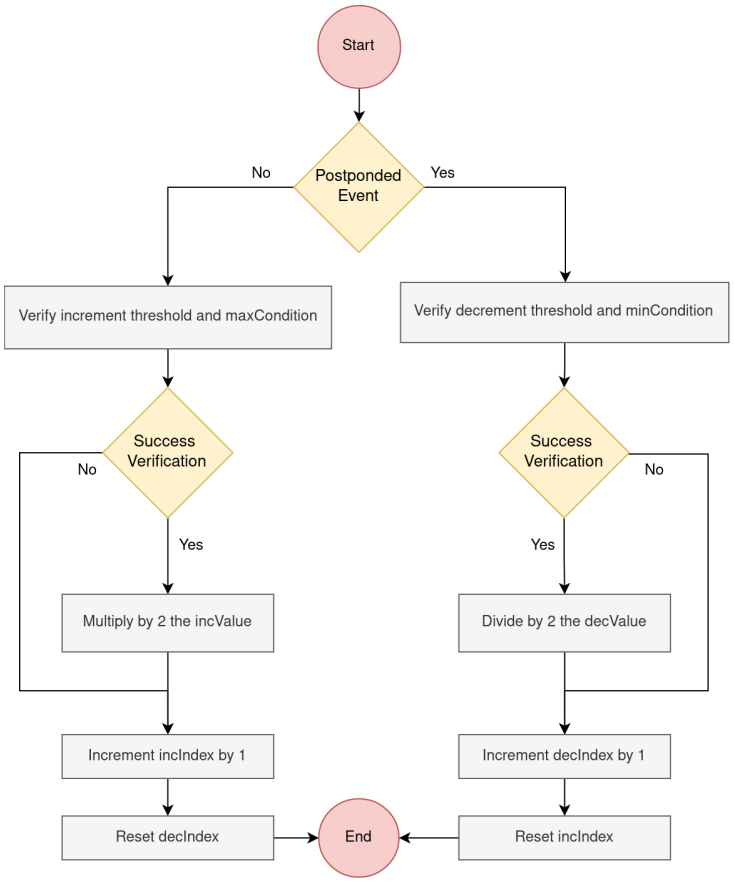
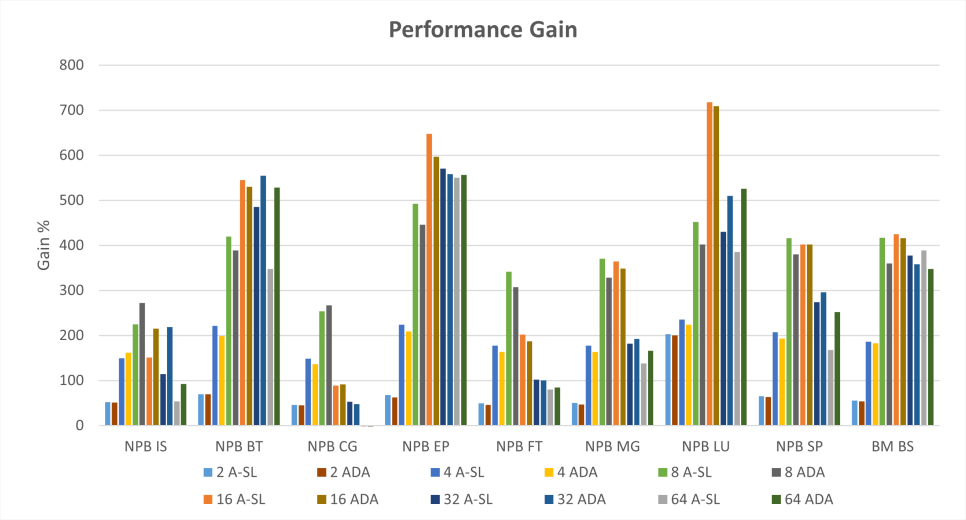


Figure 3.11: Step Ladder algorithm flowchart

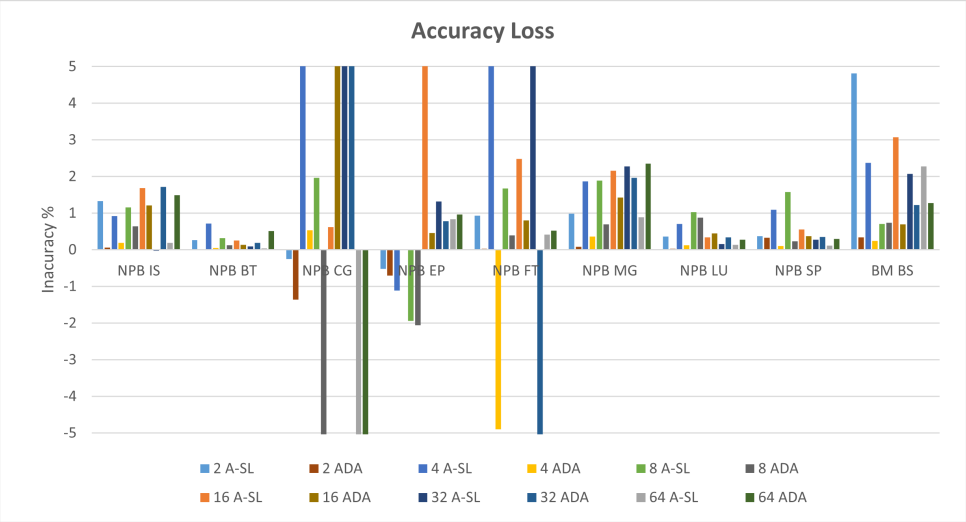
The algorithm starts checking if any event was postponed. If yes, it is considered that an "inaccuracy problem" occurred. If not, it is considered that the quantum value was the right one. Then, the *maxCondition* and *minCondition* ensure that the *incValue* does not cross the previously set boundaries. To control the increment or decrement sequence, it was created the *incIndex* and *decIndex*, respectively. These are incremented or reset, as shown in the subsequent flowchart, and they are used to check if the increment or decrement threshold has been reached.

3.3.1 Results

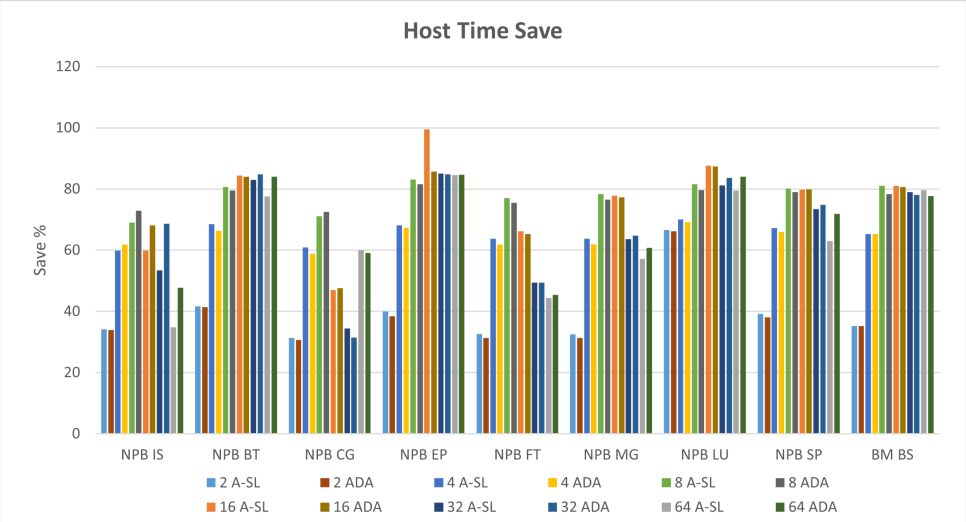
The previous benchmarks executed the new approach with different numbers of cores. The results obtained are depicted in the Figure 3.12.



(a) Performance gain



(b) Accuracy lost



(c) Host time save

Figure 3.12: Dynamic increment results

The ADA refers to the first results, while A-SL is related to the new configuration. In general, the dynamic approach with the increment value resulted in better performance. Due to the dynamic threshold, when simulating with 32 and 64 cores, there are cases where the previous approach yielded a better performance. Examples of that are NPB BT, NPB LU, and NPB SP. However, focusing on the remaining cases, on average, the performance increment was about 11%. Additionally, as a consequence of that, the overall host time was lower, reducing it by 1%.

Unfortunately, the same did not occur in terms of accuracy. Negative inaccuracy is still present and, in all cases, just with a few exceptions, there was an accuracy reduction, where, in some cases, such as the NPB FT and NPB EP with sixteen simulated cores, it fell below 95%. Regarding the first topic, the reasons are the same as presented in the previous section. About the other, a possible explanation for this could be the nature of the algorithm itself. The adaptive filter may not be designed to handle drastic interventions. As shown in Figure 3.3b, the control action appears to be smooth, even though there are moments, for example, in inter-process communications when it should be more aggressive to prevent postponed events in cross-scheduling. The problem was not evident at the beginning, however, with the dynamic increment, it became more notable.

One of the requirements for this dissertation is to maintain a maximum inaccuracy of 5%, thus, the current approach does not meet this criterion. It is crucial to explore alternative methods and optimizations to ensure that the desired level of accuracy is achieved.

3.4 Instruction Flow Prediction Algorithm

As mentioned before, one problem of the previous algorithm is the incapability to reduce the quantum significantly in a short period. A potential solution for this issue could involve predicting when this situation is likely to occur. In other words, an attempt can be made to assess when a postponed event will take place. If it becomes possible to forecast when such an event will be postponed, adjustments can be made to the quantum before it happens. This preemptive action can help prevent significant inaccuracies from occurring.

Before the simulation initiates, the commands intended for execution are loaded into memory and remain unaltered throughout the simulation. When executed, these commands can lead to a postponed event. An example of that is the Wait For Event (WFE) instruction, where the CPU enters in low-power standby state. Furthermore, in general, loops are the backbone of benchmarks, whether due to multiple iterations or the benchmark's inherent characteristics, such as testing the transfer of memory blocks between different locations. As a result, the same memory address can be encountered more than one time.

By evaluating the Program Counter (PC), it is possible to understand if the processing of one memory address can result in a postponed event (*PPaddr*), and, so, avoid its execution with a large quantum. With this idea in mind, the IFP algorithm, as the name suggests, will analyse the PC during the simulation, in

order to perform the prediction of when these will be executed. Figure 3.13 shows an example of how the algorithm works.

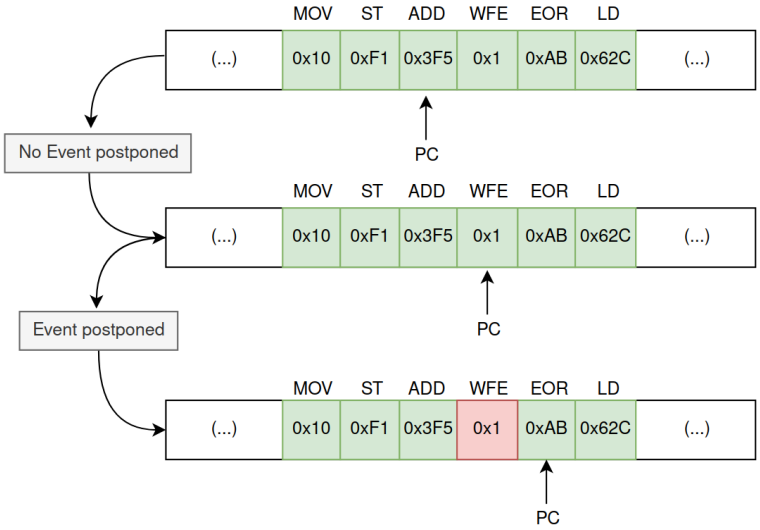


Figure 3.13: IFP Algorithm

In the beginning, every memory address is considered risk-free, but as the simulation proceeds, it can be changed. When it happens, the processed address becomes a spotted address (marked in red in the figure) until the end of the simulation. All these addresses will be taken into account in the new quantum calculation, thus, to avoid possible noise on the attribution, these will be acknowledged as genuine "postponed event generators" only when they are identified regularly.

3.4.1 Forecast

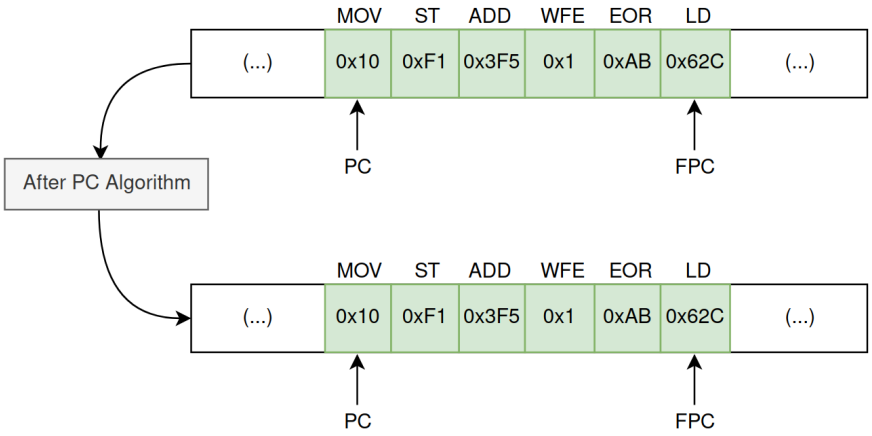
Workloads are not straightforward in a way that the PC do not follow a linear path. Instructions like "jump" (JMP) or interruptions have the potential to redirect the execution flow to different memory locations. Understanding exactly what type of instructions the simulator will execute and tracking them is computationally heavy, which implies performance costs. Since the algorithm should be lightweight, it is considered that the PC is linear, that is, the next PC will be the actual plus the instruction width.

First of all, it uses an analytic method to find where the program counter will be in the future. As a result of the previous consideration, the Equation 3.3 was developed.

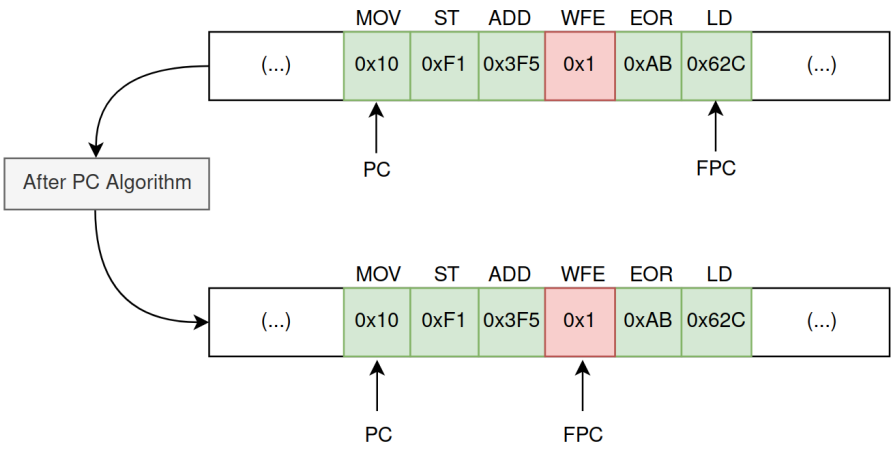
$$FPC = PC + \left(\frac{Quantum * InstWidth}{CyclePeriod} \right) \tag{3.3}$$

$$Quantum = \frac{(FPC - PC) * CyclePeriod)}{InstWidth} \tag{3.4}$$

Thereafter, one of two scenarios can unfold, as depicted in the next pictures. Nothing may transpire if no addresses were identified between the PC and Forecasted Program Counter (FPC); Alternatively, if the FPC corrects its value to the nearest identified address, a quantum recalibration is necessary using the Equation 3.4. Hence, the synchronization will occur right after the execution of the spotted address, avoiding inaccuracy as the cross-schedule event is inserted in the target event queue at the expected time.



(a) Scenario 1



(b) Scenario 2

Figure 3.14: Possible scenarios after the forecast

3.4.2 Step Ladder Update

With the development of this new algorithm, there is a new way to verify if the quantum was reduced, meaning that the old one was greater than the desirable. For this reason, the Step Ladder algorithm will also verify this situation, counting as an "inaccuracy problem", contributing to the reduction of the increment value.

To determine whether the previous scenario occurred, the quantum value before the IFP algorithm's action is stored and compared to the current. If the stored value is lower than the current quantum, no action is taken. However, if the stored value is equal or higher, it indicates that the quantum was reduced, triggering the flag.

3.4.3 Results

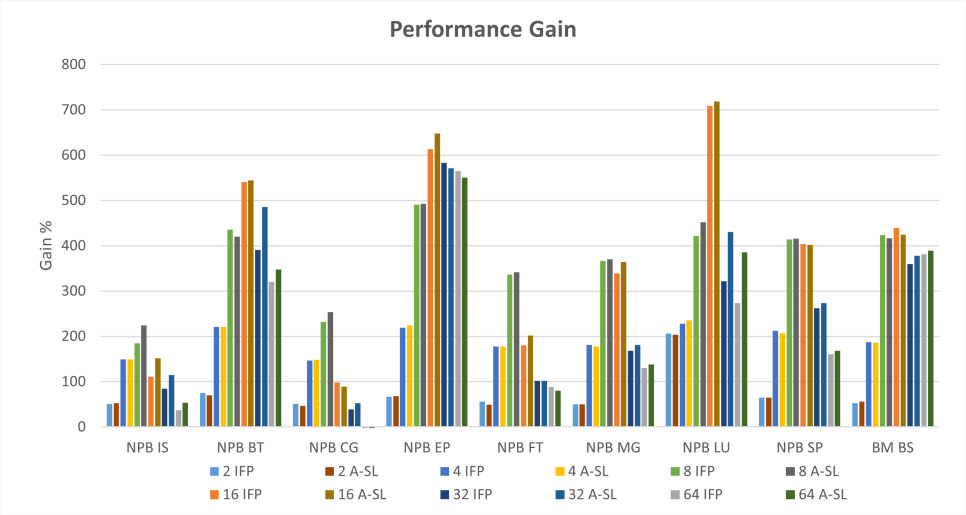
The results of the bare-metal bubble-sort and NPBs benchmarks execution are presented in the Figure 3.15. As previously, A-SL refers to the results of the combination of the ADALINE-based and Step Ladder algorithms, and IFP points to the IFP algorithm intervention results.

As expected, there was a drop in the performance gain when compared to the previous. This was about 5%, which, in practice, reflected in an increment of the host time of around 0.3%. Nevertheless, there are few cases in the algorithm's action conducted into better performance, for instance, the NPB BT over two and eight simulated cores.

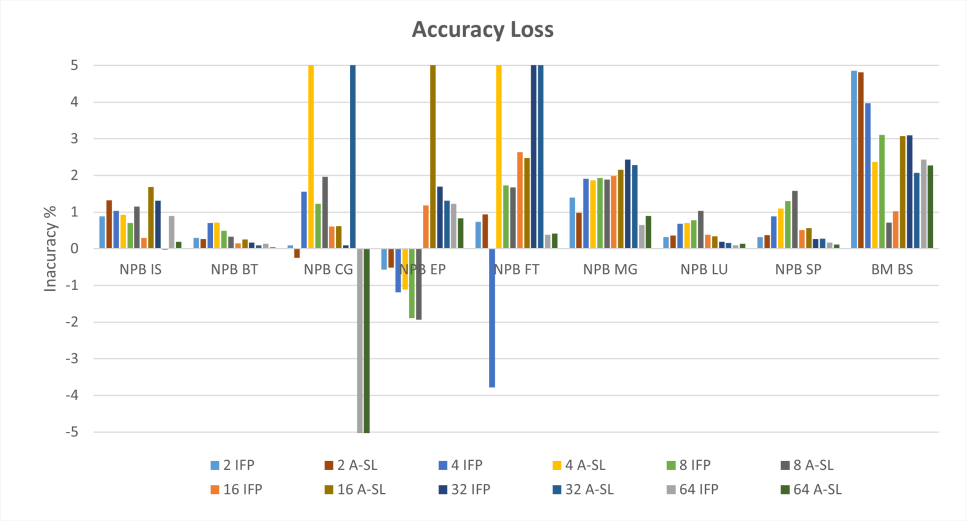
The most distinguished side is the accuracy, where the PC analysis inclusion allowed the removal of the inaccuracy peaks that existed in the A-SL version. Hence, the previously mentioned cases that passed the 5% are now within the limit. Only the NPB FT with 32 and NPB CG with 64 simulated cores are exceptions, however, these should not be considered, as explained further in this chapter. When considering up to 16 simulated cores, there was approximately an increment of about 0.5% in accuracy.

Negative inaccuracy persists in the simulations. While the occurrence of this issue has been reduced in comparison to the initial version, it is important to retain that this problem remains associated with Par-gem5, as aforementioned.

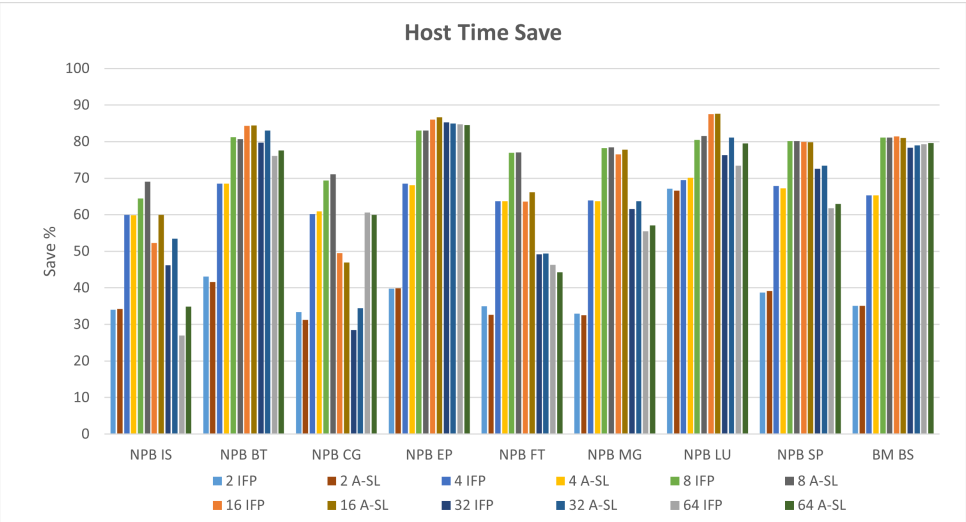
It can be concluded that the IFP algorithm presence allowed the quantum to immediately reduce its value. It also can be identified when analysing the worst-case scenario of inaccuracy, where drops can reach up to 10%. The tradeoff was even improved, since the enhancements in accuracy outweighed the increase in host time.



(a) Performance gain



(b) Accuracy lost



(c) Host time save

Figure 3.15: IFP algorithm results

3.5 Loop Detection Algorithm

As mentioned earlier in this chapter, loops are strongly present in benchmarks. Thinking from this perspective, if a record of every executed memory address is made, a loop can be identified in real-time. Combining this with the previous algorithms, the quantum could be adapted to have the highest value possible. Furthermore, the accuracy would be nearly perfect, since all cross-schedule events would be known, allowing for precise adaptation. The later figure presents how this idea can be integrated into the quantum calculation.

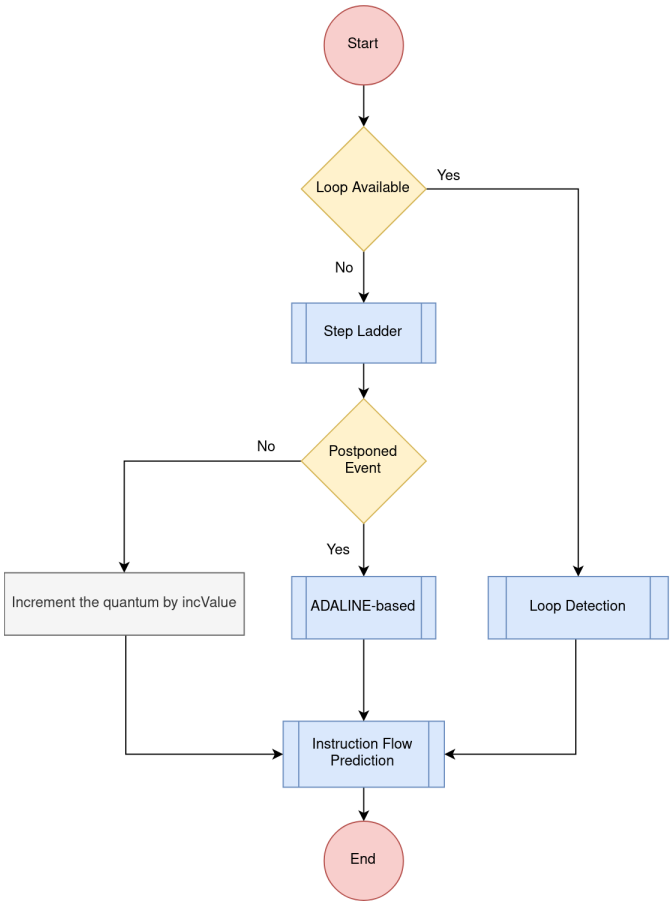


Figure 3.16: Quantum definition with Loop Detection algorithm

When the simulation starts, there is no information about loops or *PPaddrs*. To achieve this, it would be necessary to execute the simulation once and, then, obtain the results, which do not meet the requirements. One solution is to conduct the simulation in a regular manner while simultaneously recording the executed memory addresses. When a loop is detected, the algorithm begins utilizing the loop to calculate the quantum until the PC no longer follows the loop's path. At that point, the loop is reset, and the process starts anew.

3.5.1 Hare-Tortoise Algorithm

The detection method must be light, otherwise, the simulation performance is set in danger. One algorithm with this characteristic is Floyd's Tortoise and Hare. It consists of two pointers, one twice faster than the other. If both match at some point, it means that there is a loop, as shown in the figure below.

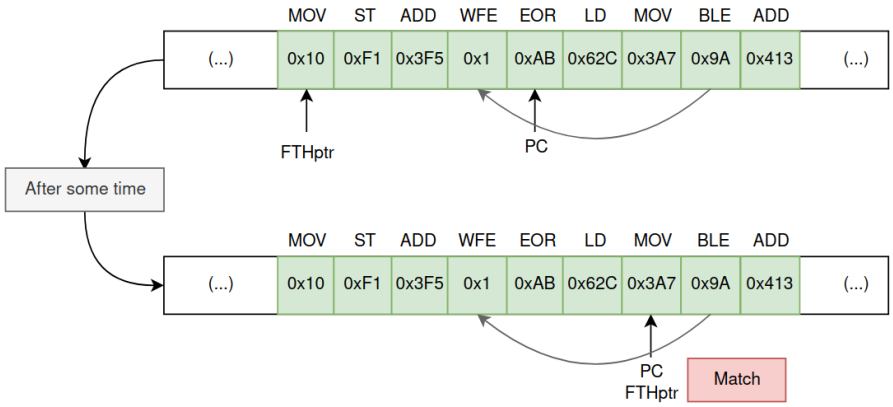


Figure 3.17: Hare-Tortoise Algorithm

The flowchart present in the Figure 3.18 describes the Loop Detection flow. In this case, the faster pointer is the PC, and the slower one is the FTHptr. A match occurs when both are pointing to the same memory at the same time. Going into the process in detail, loop delineation can be achieved through two methods. Firstly, an analysis can be conducted on the preceding memory address to precisely define the loop boundaries. Alternatively, after the initial match, the FTHptr remains in the same position while the PC continues execution. When they match once more, the loop becomes well-defined. In terms of performance, it is trivial that the second approach is more underweight, reason why it was chosen. After the detection, it is necessary to keep track of the PC, as mentioned earlier. It is done by comparing the actual PC with the expected one. If they match, no action is taken. In the opposite way, if there is a mismatch, the loop is discarded and the entire detection process starts again.

Although this technique is simple and very effective, it has some inherent problems. The first issue pertains to the inability to detect nested loops. Another challenge is the incapability to identify subsequent loops following the one that has been detected. Lastly, the most crucial problem is the difficulty in detecting conditions within loops. All of these can be solved with the PC track, nevertheless, this solution may cut the benefits of the algorithm, since these problems can be recurrent in the benchmark.

Another problem is the multi-thread environment. When more than one CPU is active, the PC is used by multiple event queues, meaning an unpredictable pointer. In this situation, it is almost impossible to identify any loop, thus, a restriction that permits the Loop Detection only when one CPU is active was implemented. If there is a loop defined and the other CPUs wake up, this is automatically discarded, and the algorithm stops, restarting again when the later condition reverts.

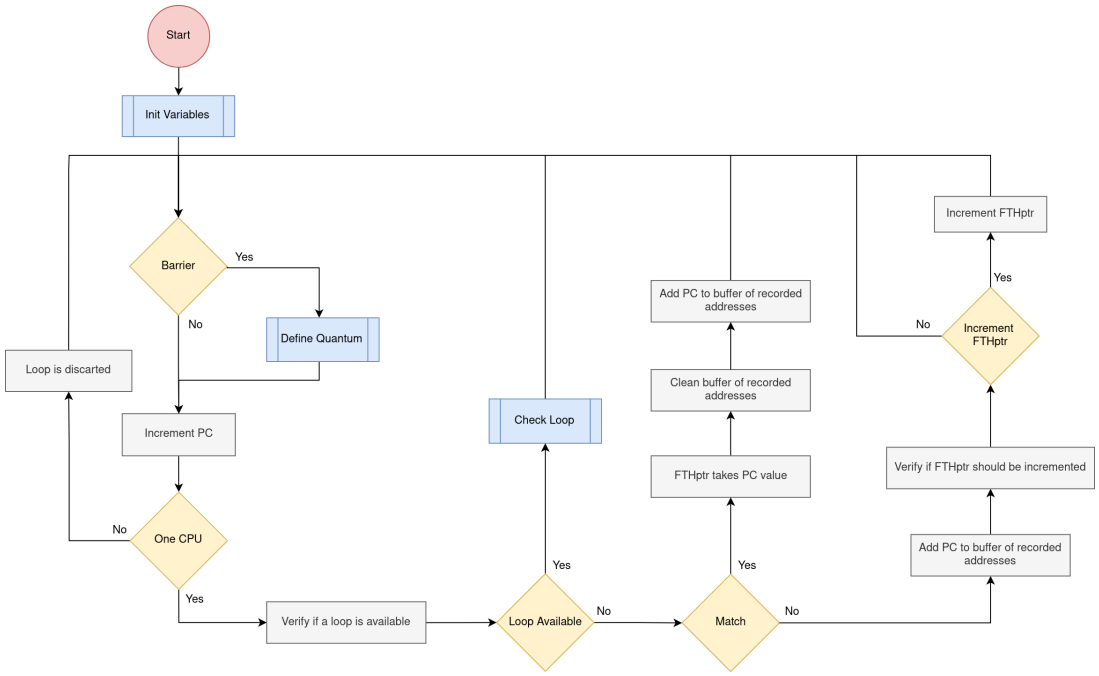


Figure 3.18: Loop Detection flowchart

3.5.2 Quantum Calculation

The quantum calculation is done at the synchronization point, as shown in detail in the Figure 3.19. It can be divided into 2 distinct tasks. The first one consists of classifying the loop addresses. The second comprises the calculation of the new quantum.

Regarding the first task, it is only executed when the loop is new, so, it is only done once per loop. The loop's size is utilized to determine whether it is the same loop or not. This approach is chosen for its simplicity and the low probability of encountering two different loops with identical sizes. After this verification, an iteration within the loop is made, comparing these with the spotted addresses. If there is a match, that value is associated with a *PPaddr*.

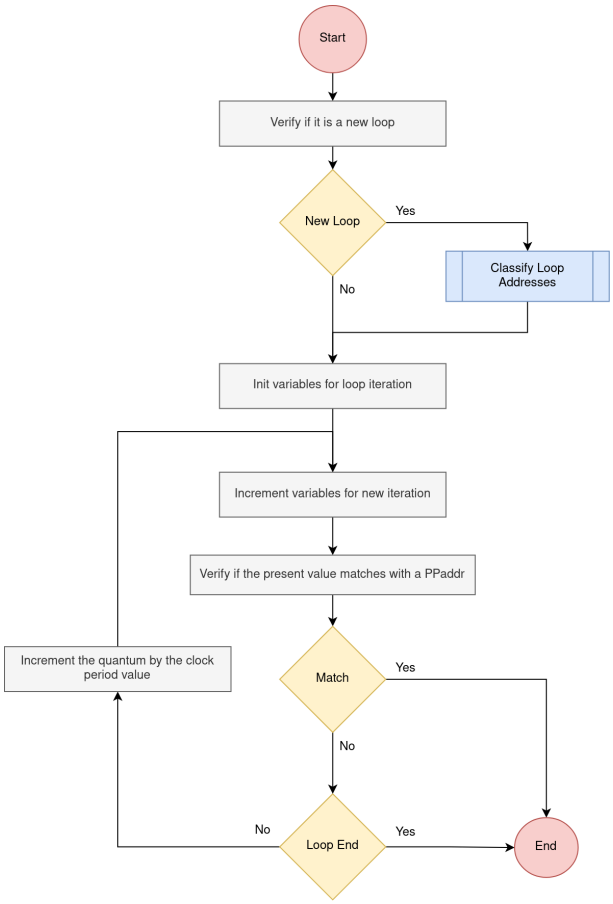


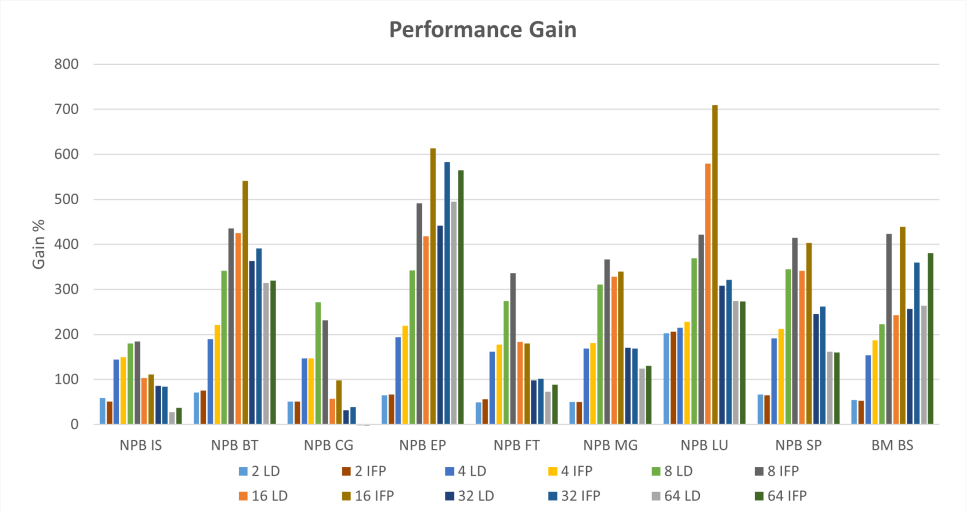
Figure 3.19: Quantum calculation flowchart

To calculate the quantum, the last task uses an iteration technique, as demonstrated in the above flowchart. The iterator starts where the PC is, and pursues the loop until it either finds a *PPaddr* or reaches the starting point. The quantum starts with the minimum value, which is the clock period, and increments that value in each iteration.

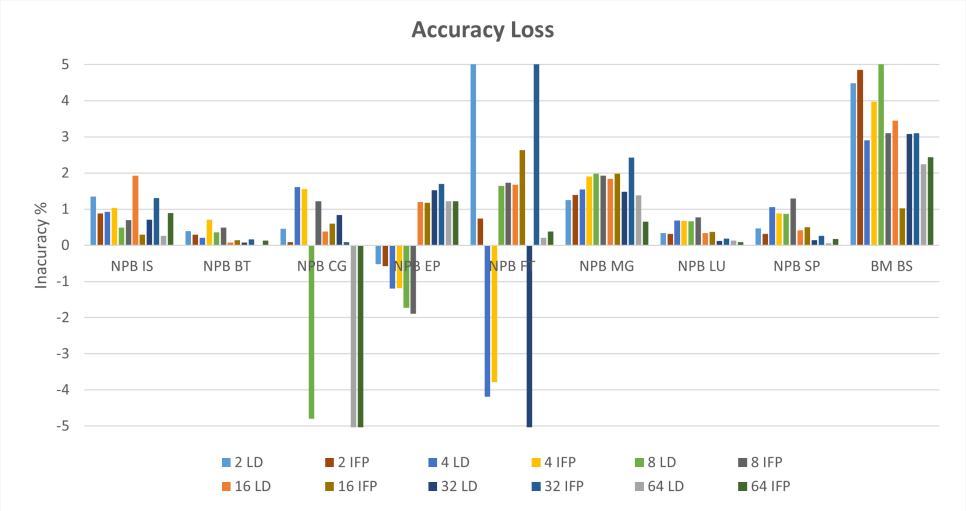
3.5.3 Results

Figure 3.20 exhibits the results of the executed benchmarks with the previous (IFP) and new (LD) configurations. When compared with the previous algorithm, the new one does not bring a better tradeoff between performance and accuracy. In reality, it is the opposite, that is, the Loop Detection action resulted in a loss of benefits.

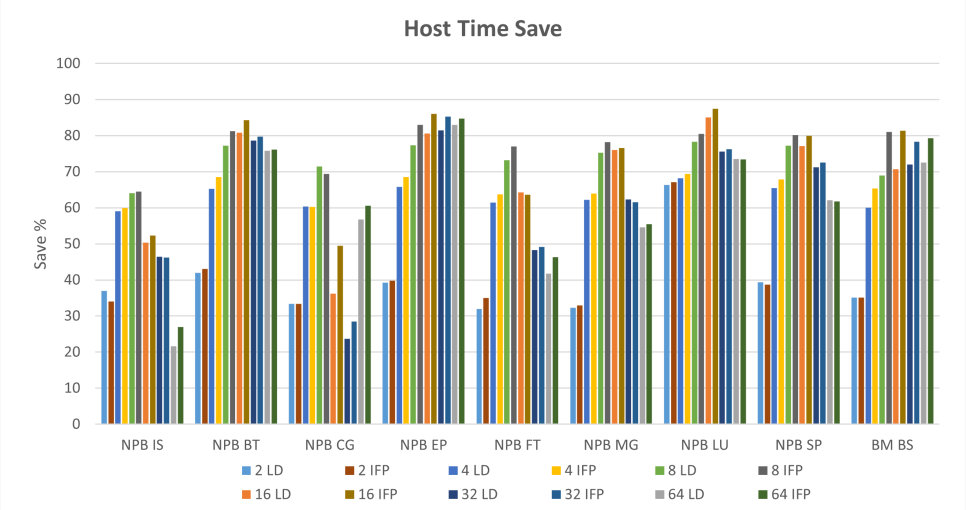
Performance, and, consequently, host time were the ones that experienced the most decline. Nearly every simulated scenario exhibited it, with particular emphasis on the NPB EP case, where the loss was around 97%. Overall, the performance drop was, on average, almost 40%, representing in host time, also on average, a reduction of 3%. Although it is a small value compared to the previous one, it can reflect a huge difference, like the case of NPB LU with 16 simulated cores. In percentage, the host time difference is almost 2.5%, however, the actual time difference is 1433 seconds, that is, almost 24 minutes.



(a) Performance gain



(b) Accuracy lost

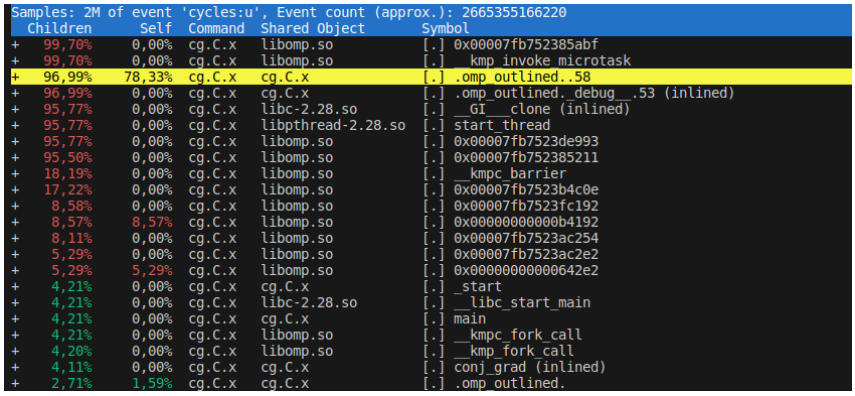


(c) Host time save

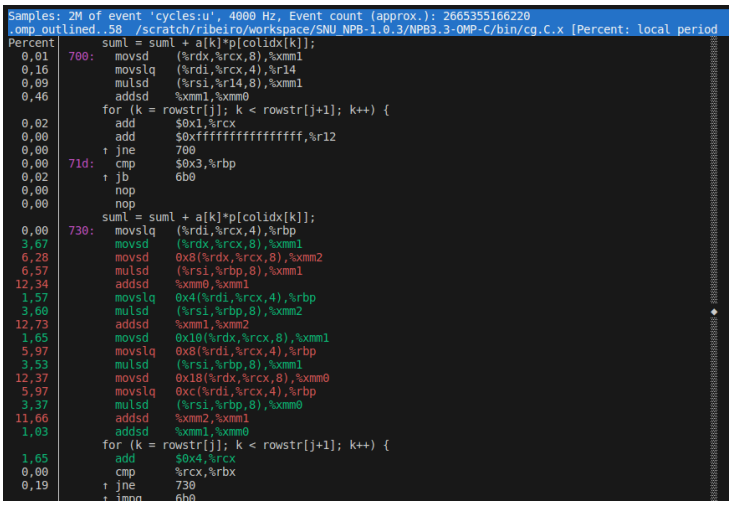
Figure 3.20: Loop Detection algorithm results

On the accuracy side, the gains were not as evident as expected. In some cases, there was a growth in the inaccuracy, having inclusively overpassed the 5% limit in the two cases. There are also others where there was a reduction, for example, the NPB FT with 16 cores and the NPB MG with 32 cores. Negative inaccuracy is still present due to the exposed reasons in the section 3.2.4.

In the final analysis, the performance sacrifice does not reflect a relevant gain in accuracy. Nevertheless, there is one case, NPB CG, where until eight simulated cores, the performance was a little bit better than the approach presented in the last section. Concerning this exception, a deeper look was done at this benchmark using perf, a performance analysis tool for Linux that provides a framework for both hardware and software-level features. After analysing the NPB CG, the result present in the Figure 3.21a was obtained.



(a) Performance analysis



(b) Region of greatest time cost

Figure 3.21: Perf analysis on NPB CG

It is evident that there is a specific part within this benchmark where the simulator consumes the major processing time. This part is exposed in the Figure 3.21b. If all percentages that are associated with an instruction with a green or red color were summed, the outcome would be nearly 94% of time spent. By examining the benchmark's source code, it is possible to identify the previous region, which is shown below.

```
//---------------------------------------------------------------------
//  q = A.p
//  The partition submatrix-vector multiply: use workspace W
//---------------------------------------------------------------------
for (cgit = 1; cgit <= cgitmax; cgit++)
{
    //...//

    for (j = 0; j < lastrow - firstrow + 1; j++)
    {
        sum = 0.0;

        for (k = rowstr[j]; k < rowstr[j+1]; k++)
        {
            sum = sum + a[k]*p[colidx[k]];
        }

        q[j] = sum;
    }

    //...//
}
```

Code 3.3: Snippet source code of NPB CG

Upon the code examination, it is evident that three nested loops perform repetitive tasks. In other words, there are no conditional statements to modify the workflow path, which does not happen on the remaining benchmarks. Thereby, it can be stated that this type of workload takes advantage of this approach, obtaining better results. For this reason, this algorithm should not be discarded, as potential future optimizations for specific tasks may require this approach.

3.6 Improved Baseline Algorithm

After the development and testing of all aforementioned algorithms, it was determined that the better approach is the combination of the ADALINE-based, the dynamic increment, and the PC analyses. The results show that they complement each other, providing a great trade-off between performance and accuracy. The loop detection algorithm was not integrated into the final solution due to its weak performance and accuracy gain.

The synchronization process can now be redefined to integrate the dynamic approach, as illustrated in the following flowchart. The static version was not removed because, with a-priory information about the benchmark, the best quantum can be calculated before simulating. One example of this consists of

a network application, where the communications delays are well-defined. If the quantum is set with the smaller delay, a perfect accuracy can be achieved [55].

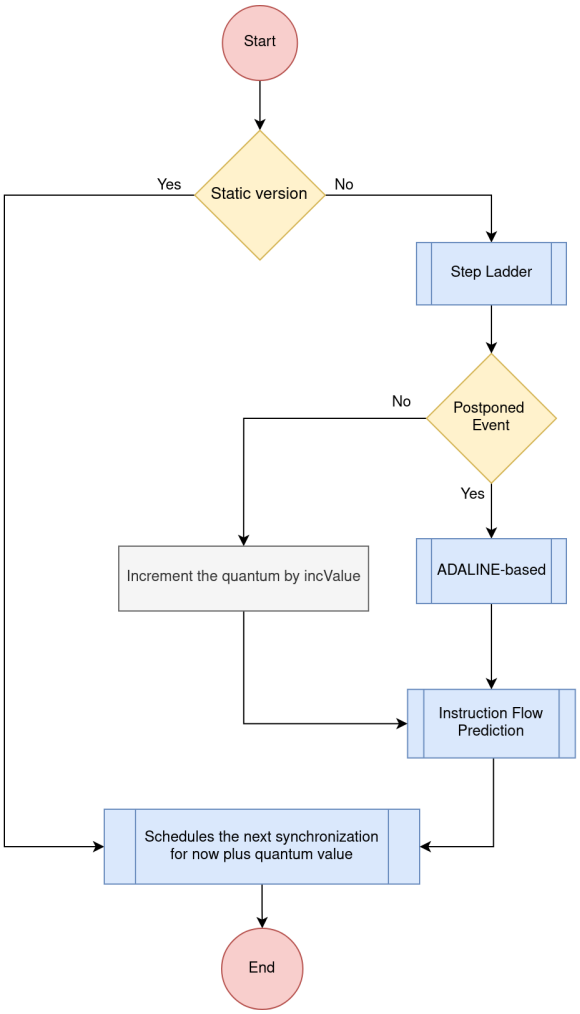
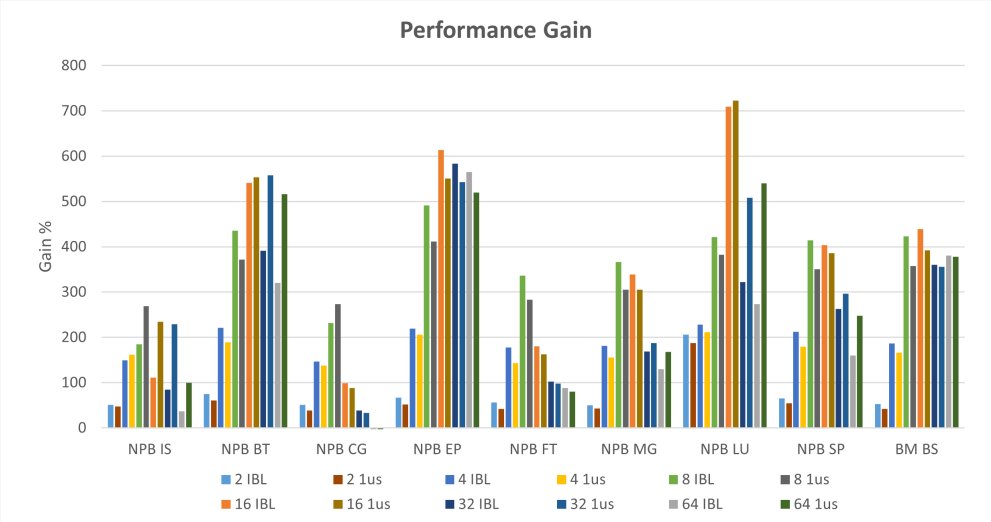


Figure 3.22: New quantum definition in the synchronization process

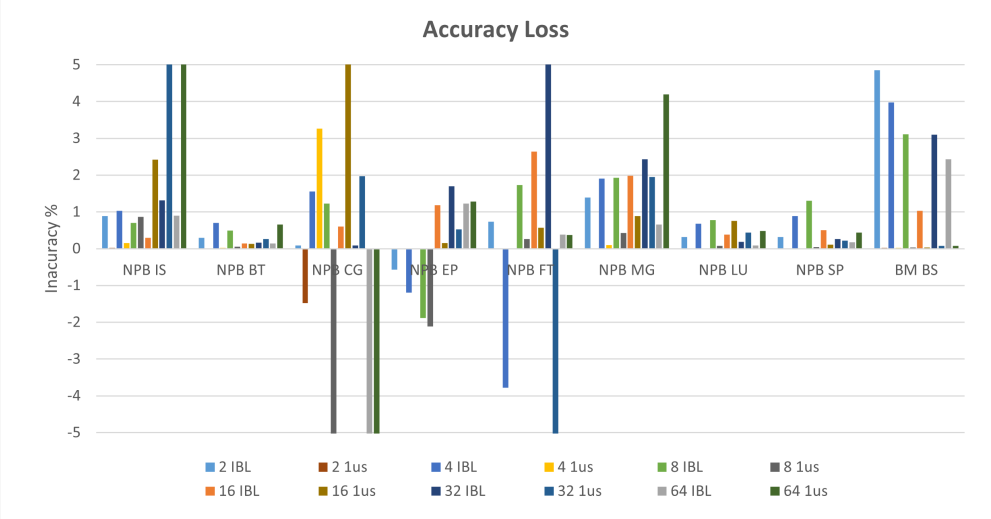
3.6.1 Results

The next graphs present a comparison between the improved baseline algorithm and the static mode with one microsecond synchronization time.

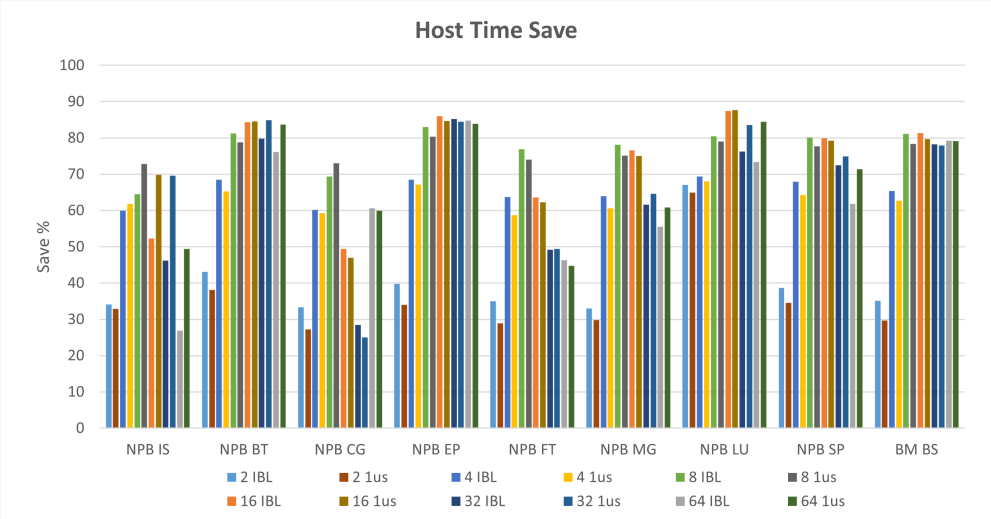
Starting with performance, the dynamic version outperformed the other one in most of the cases. Nevertheless, in the cases where it did not happen, in particular, NPB BT, LU, and SP with 32 and 64 simulated cores, the performance difference was huge. This is related to the Step Ladder design, as previously explained in this chapter. If the previous cases are considered, when calculating the mean of performance gain and comparing it with the static version, the result would be about -9%. Without the previous tests, the gain average would be almost 10%.



(a) Performance gain



(b) Accuracy lost



(c) Host time save

Figure 3.23: Improved baseline and static mode comparison

Moving to accuracy, in all workloads, the dynamic approach only exceeded the 5% inaccuracy in two cases. On the other hand, the static version had a worst performance, having six occurrences. In the remaining cases, on average, there was an increment of 0.5% of inaccuracy when using the developed algorithm. As previously discussed, the issue of negative inaccuracy persists in both approaches, as has been observed thus far.

It is important to mention that, in the dynamic case, all of these occurrences are related to simulation issues inherent in Par-gem5 nature. Going into detail, the sum/subtraction of the sequential target time with the worst-case scenario may give a smaller/higher value than the obtained target time. Figure 3.24 demonstrates the previously described scenario. The target time obtained must be within the maximum inaccuracy, which sometimes does not verify. In these cases, the simulation result should not be considered, and, since both fit in this scenario, the dynamic version occurrences were discarded. Meanwhile, the static version exhibited similar cases as well, with two of six representing this particular situation.

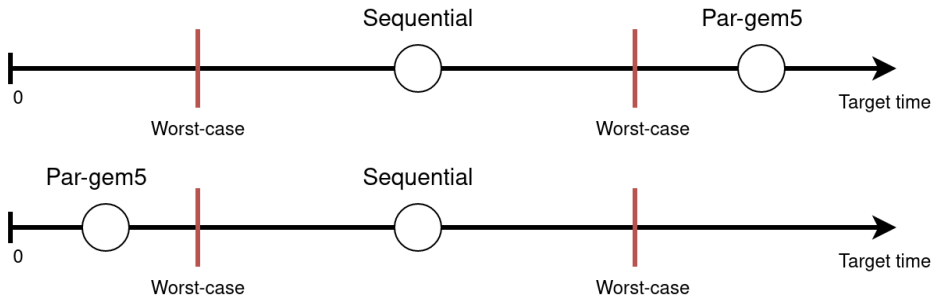


Figure 3.24: Simulation issue in Par-gem5

Finally, in terms of host time, the results are again very positive for the dynamic version. Until 4 simulated cores, only one case did not show improvements, and, in the left cases, the major part also shown gains. There are still situations where it does not verify, especially for 32 and 64 simulated cores. However, it is crucial to emphasize that accuracy should exceed 95%, a criterion that is not met in certain cases, such as NPB IS.

4 Co-Simulation on Gem5

This chapter presents a communication interface for different simulator domains. These will be integrated into Gem5 and SystemC, representing a new contribution, since no existing solutions are present in the current state of the art. Further, it was selected a case study, which serves as a stimulus for system design and demonstrates the practical application of the developed work. The chosen case study focuses on the CRC peripheral, developed on SystemC, which interfaces as an external peripheral on Gem5's board.

To close the chapter, Par-gem5 with both static and dynamic versions will be tested with the developed interface. The obtained results will be presented and compared to the ones conducted in sequential mode in order to have an evaluation reference. Concerning the results, an approach will be selected to address other situations.

4.1 Framework Proposal

As previously mentioned on chapter 2, co-simulation involves the integration of multiple simulation tools from different domains. Therefore, it is required to develop a system that could execute and communicate among them. This system must adhere to three key aspects to ensure a well-functioning environment. The first one is data integrity. Both tools must respect the defined data exchange standards, otherwise, data may become corrupted, creating a failure in the simulation ecosystem. The second aspect is data exchange. A proper delineation of the interface where data will flow is important as well. Missing this may result in faulty communication channels. The last one is the synchronization between both simulators, since improper synchronization may lead to causality errors.

Regarding these concerns, it was proposed the framework presented in the following picture. It will be composed of three interfaces: Gem5, SystemC, and remote port. Gem5 and SystemC are two simulators already introduced in the state-of-the-art chapter, however, other simulators could be used. The remote port interface was defined to allow communication between the tools.

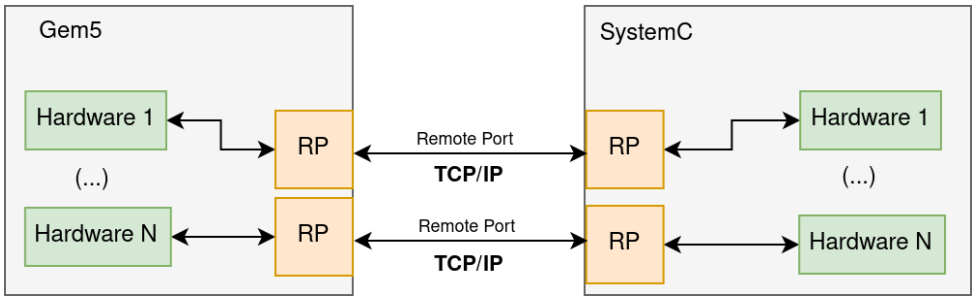


Figure 4.1: High-level proposal design of the framework

Each simulator will perform different tasks, meaning that they will operate at different abstraction levels. For this reason, the remote interface is required in order to receive the frames of one simulator and adapt to the other. This exchanges data utilizing the TCP/IP IPC protocol, due to its application independence and its versatility for various applications and purposes. Furthermore, this type of connection can be used remotely, with multiple simulators running on different machines. As shown in the previous figure, each hardware device will have a unique connection to avoid concurrency problems.

4.1.1 Remote Port Interface

As already mentioned, the remote port interface will establish the communication between Gem5 and SystemC. Although they are two simulators that use C++, they cannot communicate directly with each other. Gem5 is unable to interpret the TLM commands, just as SystemC does not understand Gem5 packages. For these reasons, this interface will require a wrapper to enable communication between the two platforms. The wrapper will be available in both tools, and every transaction must be created regarding its definition, otherwise, data integrity may be compromised.

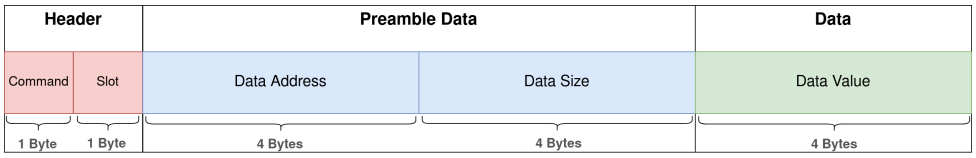


Figure 4.2: TLM wrapper payload definition

The main part of the wrapper is the payload definition. It provides a unique definition to transport data and perform operations, keeps the transactions easy to understand, and improves efficiency. For this case, its definition can be found in the prior image. It can be divided into three sections: the header, marked with red; the preamble data, marked with blue; and the data, marked with green. It is important to mention that the remote port's design was done regarding a 32-bit machine, reason why the data address, data size, and data value parameters have a four-byte length.

Every operation must begin with the payload header, which is composed of a command and a slot. The command indicates the operation to execute, and its list can be found on Table 4.1. The slot is used

as an ID to decode where the operation should be done. As shown in Figure 4.1, each hardware module connected to a remote port has a number associated, which can be indexed by the slot. Preamble data comes after the header, and it indicates the location where the execution will take place in the selected hardware. The data address or offset points to the desired hardware address, and the data size specifies the action range. The data is required in the case of writing, since the new value must be known.

Table 4.1: TLM wrapper commands

CMD Values	Command	Required Sections
00	TLM_INIT	Header
01	TLM_CLOSE	Header
10	TLM_READ	Header + Preamble data
11	TLM_WRITE	Header + Preamble data + Data

Every transaction is followed by a response, *TLM_ACK* (0) for success, or *TLM_NACK* (1) for error, and the operation latency, which is required to maintain the synchronization between the simulators.

The next figures demonstrate how the wrapper is implemented in both tools. First of all, it is mandatory to initialize the remote port interface. To accomplish that, Gem5 must start with the *TLM_INIT*, and wait for a positive response from SystemC. Only after that, the remote port is available, and write and read operations can proceed. Closing the remote port follows a similar path. Gem5 must call the *TLM_CLOSE*, and wait for *TLM_ACK* from SystemC. The wrapper always verifies if the remote port is available for transactions, and, in case of an error, it returns a permission-denied notification.

In read-and-write operations, the device's availability is crucial to maintain the tools synchronized. Picture a scenario where Gem5 sends two write requests and each write operation takes four clock cycles. When Gem5 executes the first request, SystemC will return a latency of four clock cycles. Consequently, whether another attempt is made in the next clock cycle, the device will still be processing the first one, resulting in a synchronization conflict. For this reason, if the device is unavailable, the busy notification is returned, indicating a need to retry the operation later.

In SystemC, while *TLM_INIT* and *TLM_CLOSE* are identical to Gem5 ones in the way of operating, *TLM_READ* and *TLM_WRITE* have some differences. First of all, instead of sending the frame, it receives it and verifies if the received slot is accessible. After that, it executes the respective TLM operation, which can result in an error, for example, the offset crossed the device's boundaries, or in a successful process, returning to Gem5 the respective data. Further in this chapter, these operations will be explained in detail.

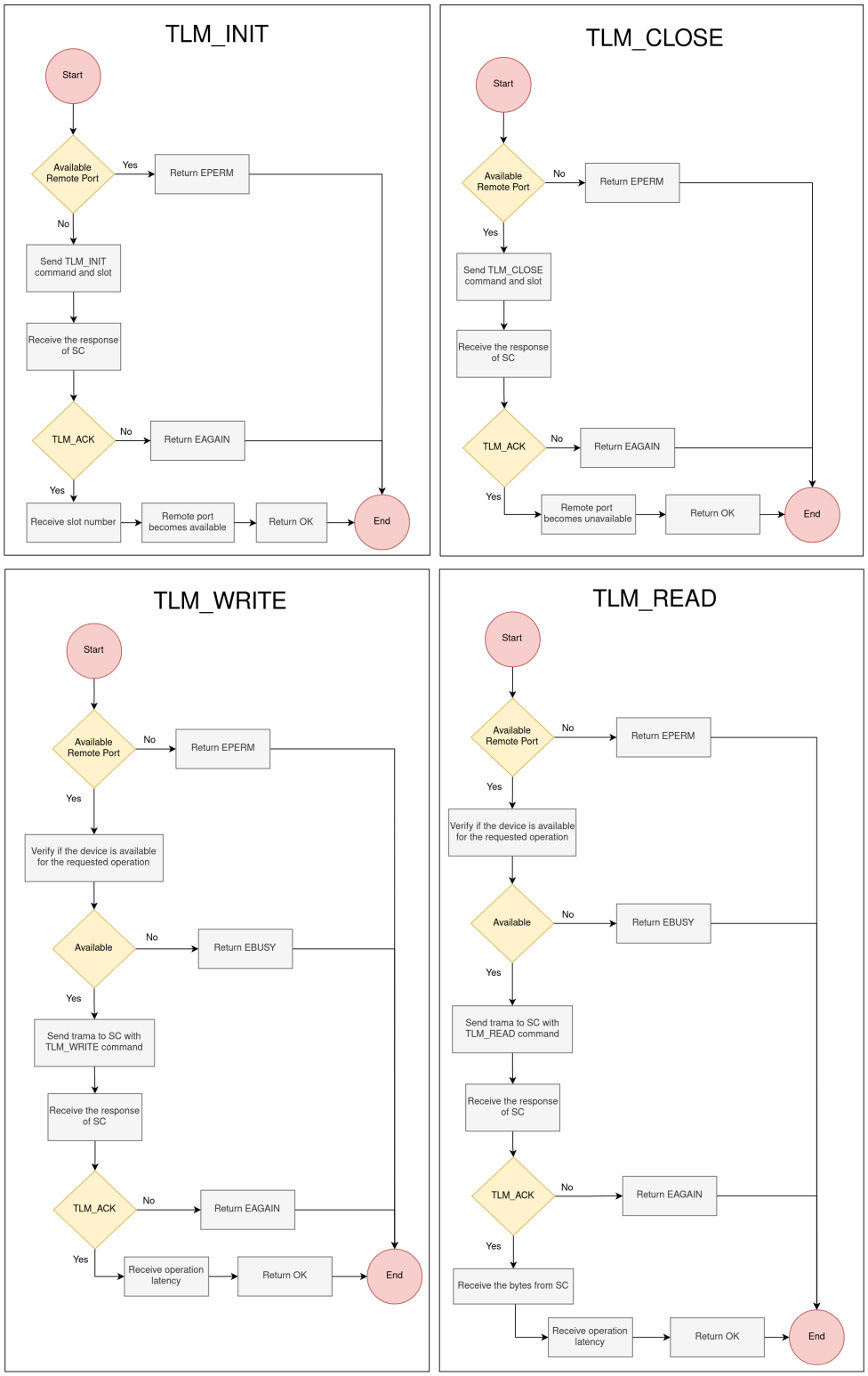


Figure 4.3: TLM wrapper in Gem5

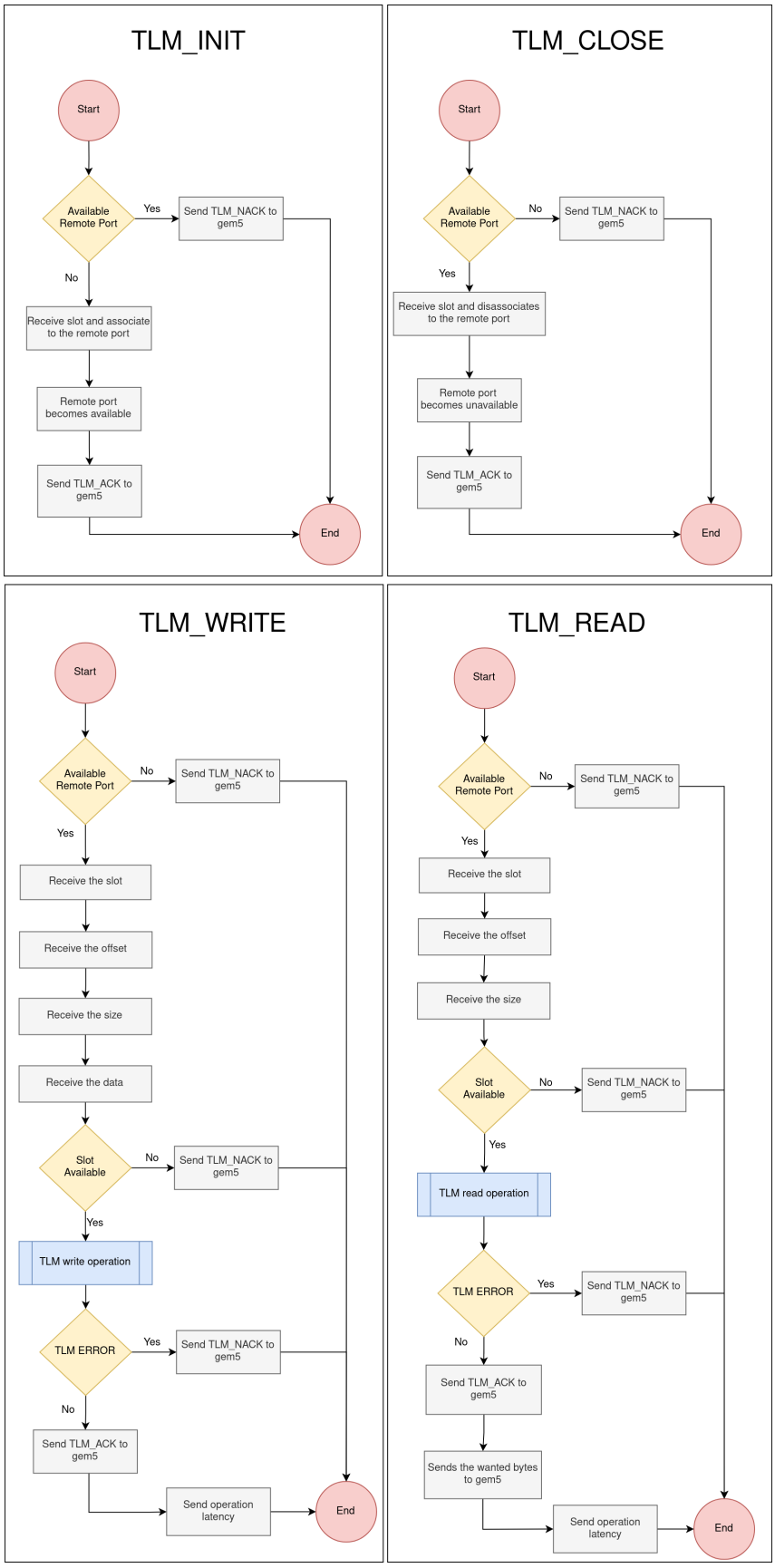


Figure 4.4: TLM wrapper in SystemC

In the end, the wrapper can be described as presented in the Figure 4.5. *TLM_Socket* provides an abstraction for socket write and read processes. Both have a dependency relationship since the *TLM_Wrapper* depends on the *TLM_Socket* functions to be able to communicate. When a remote port is initialized, the socket identification number is returned, and it is stored in the *data_fd* variable. In this way, it is possible to associate the slot with the respective remote port, providing coherent communication. When closing the remote port, this variable is reset to its default value.

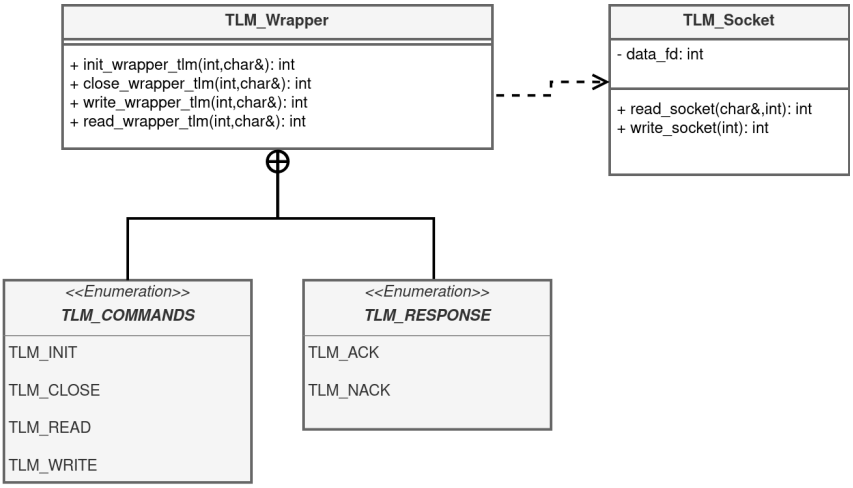


Figure 4.5: TLM wrapper class diagram

4.1.2 SystemC Interface

The peripheral development was carried out using the SystemC tool. It has the capability to simulate multiple and different hardware devices at the same time. In order to integrate this flexibility, the subsequent design was developed.

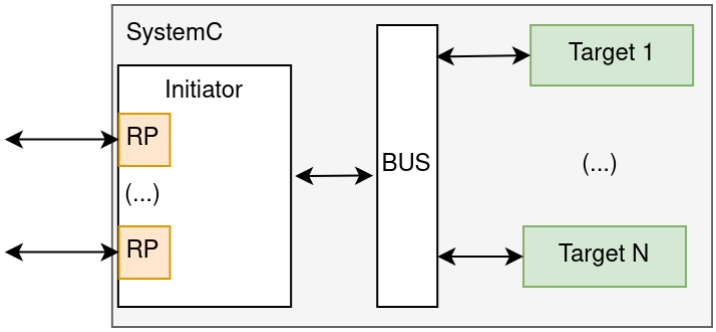


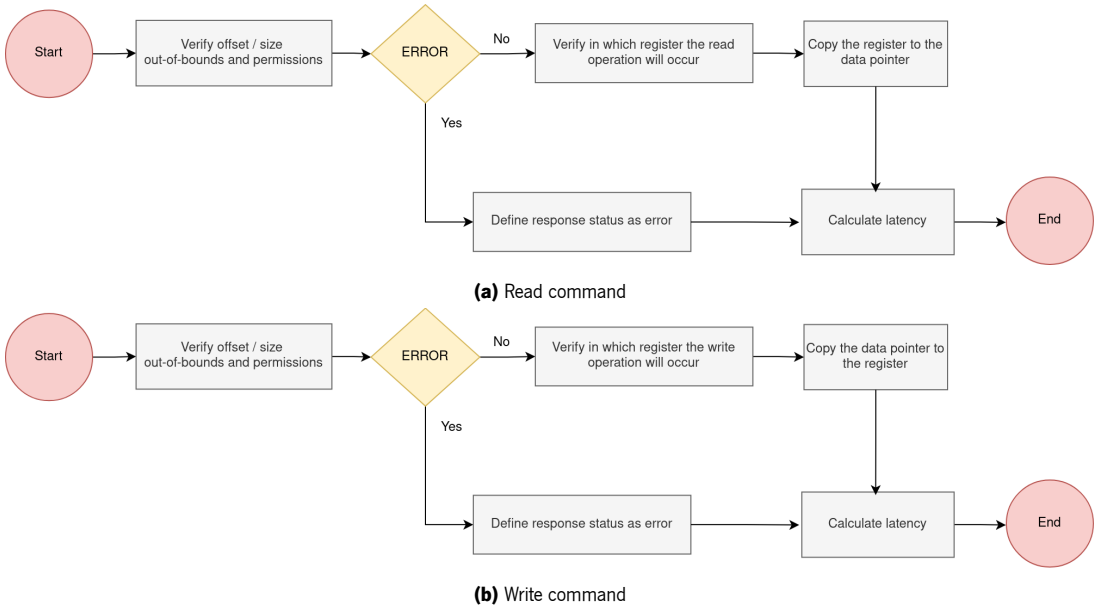
Figure 4.6: General SystemC design

SystemC design is composed of three main components: the initiator, the bus, and the targets. The first serves as the initial point of interaction for the tool, since it handles incoming frames received through the remote port. The remote port operates independently of the initiator, enabling asynchronous reception and transmission of bytes through the receiver and transmit buffers, respectively. Additionally, it also

performs an interpretation of the received frame and creates the TLM transaction. It analyses various parameters, such as the operation type, the desired target, and the memory region, among others, with the help of *TLM_wrapper*.

The bus is responsible for forwarding the TLM transaction to the correct target. Each target has a unique ID, that is assigned at the beginning of the simulation. As previously mentioned, the preamble data and data were designed for a 32-bit architecture, however, SystemC TLM transactions are 64-bit width, leaving 32 bits unused. Before the transmission, the initiator uses these bytes to set the target ID, which will be utilized by the bus to identify the targets.

Lastly, the targets are the peripherals, which can be different from each other. These receive the TLM commands and act accordingly. The next images demonstrate how peripherals respond to *TLM_READ* and *TLM_WRITE* operations.



(a) Read command

(b) Write command

Figure 4.7: Flowcharts of the available commands

The first step is to verify the offset/size out-of-bounds, and permissions in a way that the device's integrity is maintained. If no errors occurred in this process, the desired operation is done, otherwise, an error response is created, reporting where the problem was. At the end of each operation, the correspondent latency is returned, which may vary regarding the desired work. The latency is used to inform the other simulator how long it should wait until a new operation can be done. An example concerning this can be seen in the above subsection.

4.1.3 Gem5 Interface

Gem5 can integrate different boards with different characteristics. Nevertheless, the steps to integrate a device remain the same across all boards.

The first step is to identify the available memory in the MicroController Unit (MCU)'s Memory Mapped Input Output (MMIO) and to reserve the required memory for the device. Most microcontrollers have a lot of unused MMIO available, enabling the testing with new peripherals while retaining most of the original MCU.

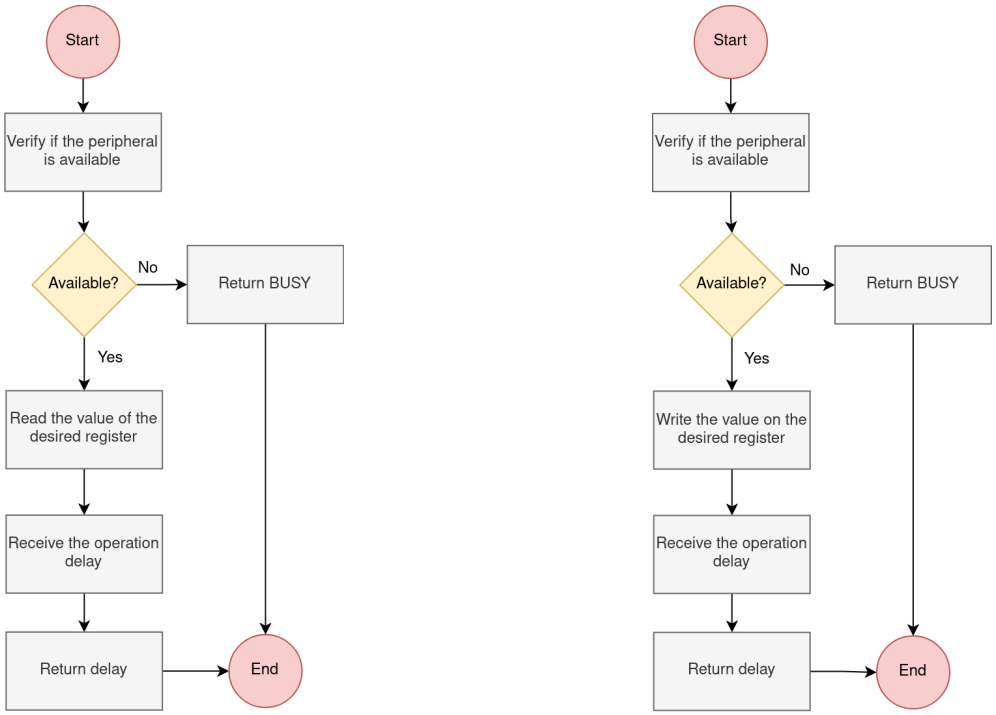
After planning this, Gem5 must recognize this hardware to enable communication with the other simulator. To achieve that, every peripheral should be defined and added to the list of off-chip devices in the board's configuration. In addition, it is also required to create a Page Table Entry (PTE) for each device, which can be done by following the code on 4.1. However, it is important to note that, after this process, the devices are only recognized as part of the board, and their implementation still needs to be completed.

```
1 LDR   r1,= DEVICE_ADDR   // Device address
2 LSR   r1, r1, #20        // Find which 1MB block it is in
3 LSL   r2, r1, #2         // Gives offset into the page tables
4 LSL   r1, r1, #20        // Put back in address format
5 LDR   r3, =L1_DEVICE     // Descriptor template
6 ORR   r1, r1, r3         // Combine address and template
7 STR   r1, [r0, r2]       // Store table entry
```

Code 4.1: Template for a PTE

To implement a device in Gem5, a few aspects are mandatory. In the first place, the peripheral interface, which describes the type where the implementation is, and, optionally, some parameters to be customized. In this context, these can be the port number for the remote connection or the action time delay. In the second place, it is needed the implementation itself. To accomplish this, the class *BasicPioDevice* should be used. It is the base class which all devices sensitive to an address range inherit from. It abstracts all device's implementation, from the creation of the SimObject to memory access protocols. Still, it is required to define the read and write operations, as these change from device to device. The figures 4.8a and 4.8b present a template for this implementation. Both start evaluating if the peripheral is available or not. If it is negative, a busy notification is returned informing that it is not available at the moment. Otherwise, the respective operation is performed, returning the amount of time spent.

The last step consists of the connection to the remote port interface. The connection process is the following: Firstly, a socket must be created; After that, the simulator listens to the socket and waits until a connection is made; Finally, it verifies if the other interfaces are ready by initializing the co-simulation environment. If it receives a positive response, the simulation can proceed, else the simulation is aborted. Note that all this process is done before the actual benchmark starts.



(a) Device's read flowchart

(b) Device's write flowchart

Figure 4.8: Redefinition of *BasicPioDevice* functions

4.2 CRC as a Case Study

To stimulate the developed interface, the CRC peripheral was chosen as a case study. This selection was based on its presence in several microcontroller families, such as STM32 [94] or Xilinx Zynq [95].

The CRC was created in 1961 by William Wesley Peterson [96]. As the name suggests, it utilizes systematic cyclic codes to encode messages by incorporating a fixed-length check value. In the end, his work contributed significantly to simplifying and enhancing the detection of accidental errors/changes in communication networks. CRC uses a generator polynomial, which is known by the sender and receiver, and it is used to perform the calculation. There are different standards, however, the most common ones are the CRC-8, CRC-12, CRC-16, CRC-32, and CRC-CCIT [97].

Another application for the CRC is the storage integrity. Due to defective components or electromagnetic fields, bits can change their value without notice. In the presented case study, this scenario will be explored, where the CRC peripheral is used to maintain a specific memory state and verify if there have been any changes. Furthermore, the board will not be only engaged in this operation, because in real-world scenarios, it has additional functionalities besides memory verification. Hence, as a proof-of-concept, parallel tasks will be included in the execution.

4.2.1 Peripheral Development on SystemC

The development of the CRC peripheral was done in SystemC, which took as a reference the STM32 microcontroller family's reference manual [94]. As presented in the figure below, the Figure 4.6 was redefined to the present case study. The initiator will only have one remote port associated, since only one device will be simulated.

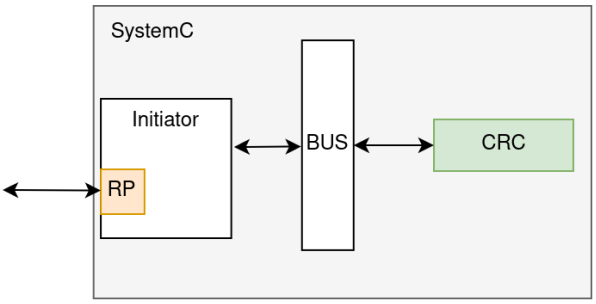


Figure 4.9: SystemC design with CRC

From the module datasheet, the peripheral is characterized by the following:

- CRC-32 (Ethernet) polynomial: 0x4C11DB7
- Programmable CRC initial value
- Single input/output 32-bit data register
- CRC computation done in 4 clock cycles
- General-purpose 8-bit register (can be used for temporary storage)
- Reversibility option on I/O data

Regarding the Figure 4.10, the module working principle is as follows. First of all, the user needs to write the input value in the data register (CRC_DR). After four CRC clock cycles, the correspondent CRC value is fully calculated, and its value is stored in the CRC_DR. In this case, the CRC frequency will be equal to the CPU. The data width must be 32-bit, hence, whether the user needs a CRC for five bytes, for example, two different computations will be needed.

Table 4.2: Reverse operation

(a) Input

REV_IN[1:0]	Input	Reverse Action	Reverse Input
0 0	0x1A2B3C4D	Not affected	0x1A2B3C4D
0 1	0x1A2B3C4D	Bit-reversal done by byte	0x58D43CB2
1 0	0x1A2B3C4D	Bit-reversal done by half-word	0xD458B23C
1 1	0x1A2B3C4D	Bit-reversal done by word	0xB23CD458

(b) Output

REV_OUT	Output	Reverse Action	Reverse Output
0	0x11223344	Not affected	0x11223344
1	0x11223344	Bit-reversal done by word	0x22CC4488

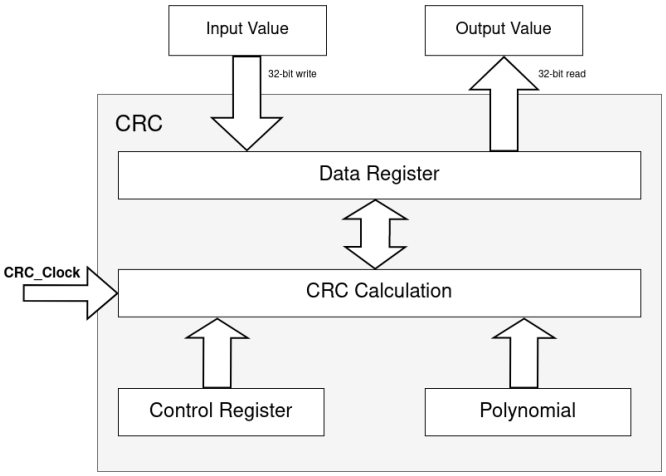


Figure 4.10: CRC block diagram

Moreover, the input and output data can be reversed, to manage the various endianness schemes. For the input data, the reverse operation is controlled by the REV_IN[1:0] bits, and for the output data, the REV_OUT bit is used. These, along with the reset bit, which is used to reset the CRC, are located in the control register (CRC_CR). This bit must be set by the software, and it is automatically cleared by the hardware. An example of a reverse operation can be found on Table 4.2.

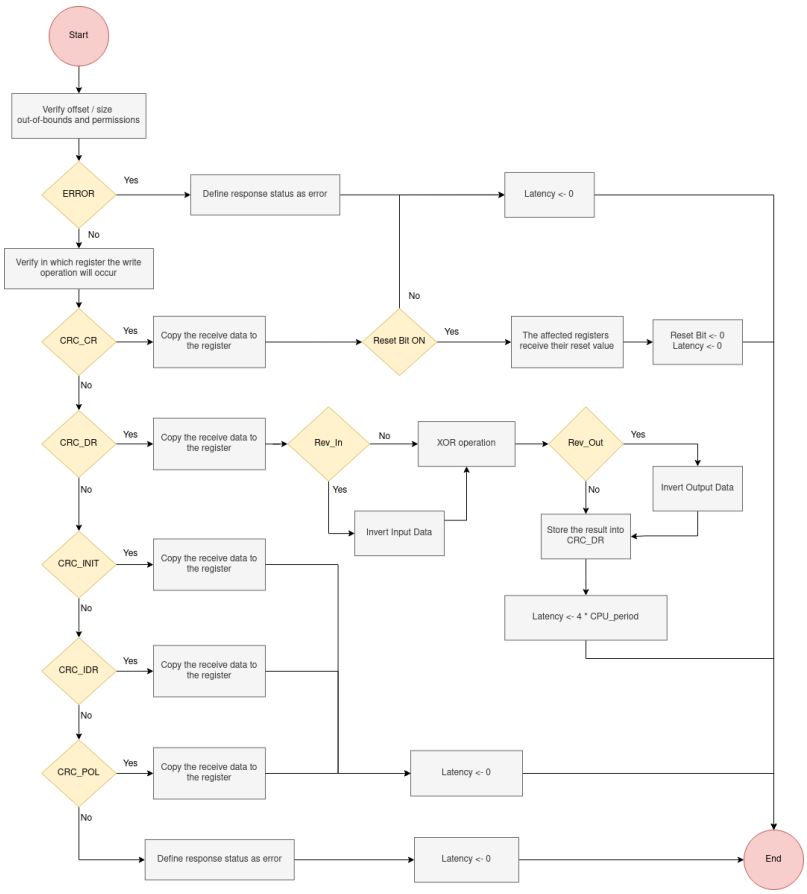


Figure 4.11: CRC write operation

By default, the polynomial coefficients are defined by 0x4C11DB7, nevertheless, it can be fully programmable through the CRC_POL register. It is important to mention that modifications in this register when a CRC computation is ongoing are not permitted, as it would compromise the output value. To complete the list of available registers, there are the CRC_INIT and CRC_IDR. These registers are used to initialize the CRC calculator during reset and to hold a temporary 8-bit value related to CRC calculation, respectively.

Finally, the write command must be redefined to adhere to the peripheral characteristics, as demonstrated in Figure 4.11. Since the read operation does not require a specific behavior depending on the desired registers, the aforementioned template does not demand any action.

4.2.2 Peripheral Development on Gem5

For the subsequent tests, the selected target platform was the VExpress_gem5 board. It is based on the ARM Versatile Express RS1, which consists of a motherboard and two daughterboards. The system provides a range of both on-chip and off-chip devices. On-chip devices include the Generic Interrupt Controller (GIC), an LCD controller, and system timers. Off-chip devices contain the Keyboard and Mouse Interface (KMI), Real-Time Clock (RTC), and Universal Asynchronous Receiver-Transmitter (UART). The platform's memory map is divided into the next sections.

1. Boot memory .....0x00000000 to 0x03FFFFFF
2. Reserved .....0x04000000 to 0x0FFFFFFF
3. Off-chip peripherals .....0x10000000 to 0x1FFFFFFF
    (a) Gem5-specific peripherals .....0x10000000 to 0x13FFFFFF
        i. Energy controller .....0x10000000 to 0x1000FFFF
        ii. Pseudo-ops .....0x10010000 to 0x1001FFFF
        iii. MHU .....0x10020000 to 0x1002FFFF
    (b) Reserved .....0x14000000 to 0x17FFFFFF
    (c) VRAM .....0x18000000 to 0x19FFFFFF
    (d) Reserved .....0x1A000000 to 0x1BFFFFFF
    (e) Peripheral block 1 .....0x1C000000 to 0x1FFFFFFF
4. On-chip peripherals .....0x20000000 to 0x3FFFFFFF
5. PCI memory .....0x40000000 to 0x7FFFFFFF
6. DRAM .....0x80000000 to 0xFFFFFFFF

The first aspect to take care of is the definition of the memory region. In accordance with the remaining devices, each peripheral should occupy 0xFFFF of memory space, unless it requires more space. In this case, another block of 0xFFFF should be used until its needs are fulfilled. Following the previously defined rule, for the understudy device memory will be reserved from 0x10030000 to 0x1003FFFF.

Then, the PTE entry must be created, with the *DEVICE_ADDR* parameter equal to the beginning of the device's memory, which is 0x10030000. The last part is the implementation of the write and read functions. Concerning the template present in the Figure 4.8, no modifications were needed to integrate this peripheral, therefore, these were employed.

4.2.3 Application API

At this point, the user on the application side can write and read directly from memory without any restriction. However, it can be dangerous, in the way that the user can, by mistake, do an unpermitted operation, for example, write in reversed memory, causing a segmentation fault. To avoid this, it was developed an API for the application side. Similar to STM32 microcontrollers that utilize the HAL library, the CRC device will also have its own hardware abstraction layer. This layer is designed to speed up development and enhance code clarity.

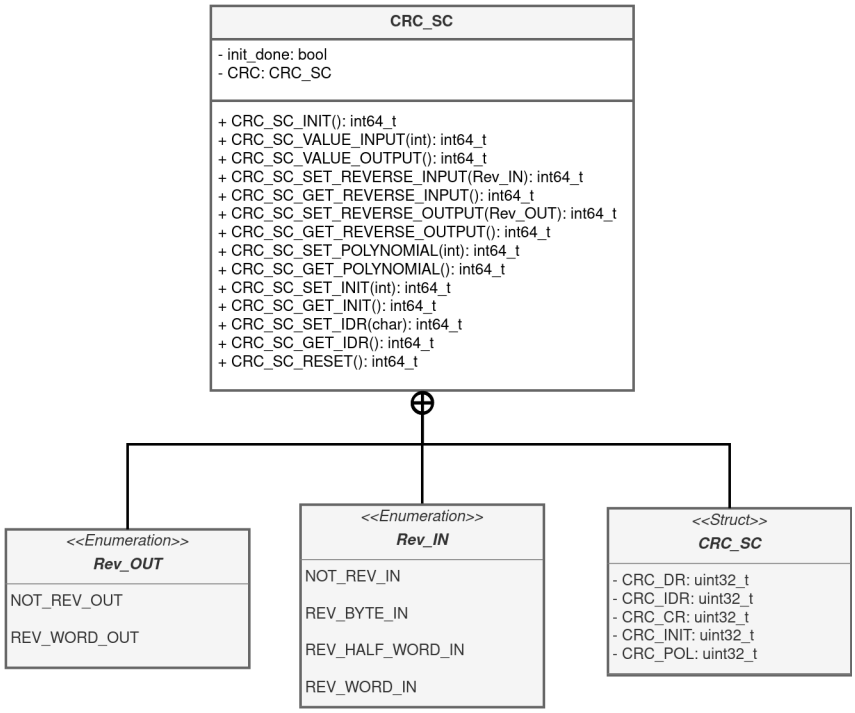


Figure 4.12: Class diagram for the CRC API

Primarily, initializing the peripheral is a mandatory step. Without initialization, any attempt to execute an operation will result in an error, and no action will take place. To calculate the CRC value of a given

number, the *CRC_SC_VALUE_INPUT* function must be called, with the corresponding number as a parameter. After that, the result can be obtained by calling the *CRC_SC_VALUE_OUTPUT* function. It is important to mention that the calculation takes four clock cycles, thus, the function can either return the CRC result or an error, warning that the value is not calculated yet. All the remaining functions are used to control the device.

4.2.4 Peripheral Validation

After development, the next step was to simulate and validate the peripheral. For this purpose, it was developed a validation test where all features of the device will be tested. At the first moment, the device will be tested isolated, that is, without the remote port and Gem5 interactions, as present in Figure 4.13. After that, it will be tested in a co-simulation environment, as shown in the Figure 4.15.

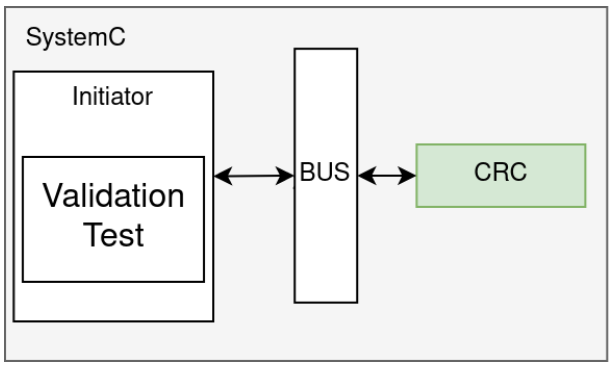


Figure 4.13: CRC peripheral validation

The designed validation test can be divided into three parts. In the first one, the reverse output will be tested. Then, the settings will change, and the calculation will use a reverse input and a different polynomial. To conclude, the reset function will be checked. By executing this benchmark, every functionality of the peripheral is checked. At the end of the simulation, the expected return values are 0x22CC4488 for the first part and 0xC66CE444 for the second part. The final output will be the result of the reset operation.

Taking this into account, the test was executed in the first-mentioned condition, obtaining the results demonstrated in the figure below. As expected, the obtained results matched with the expected ones, concluding that the peripheral is working properly.

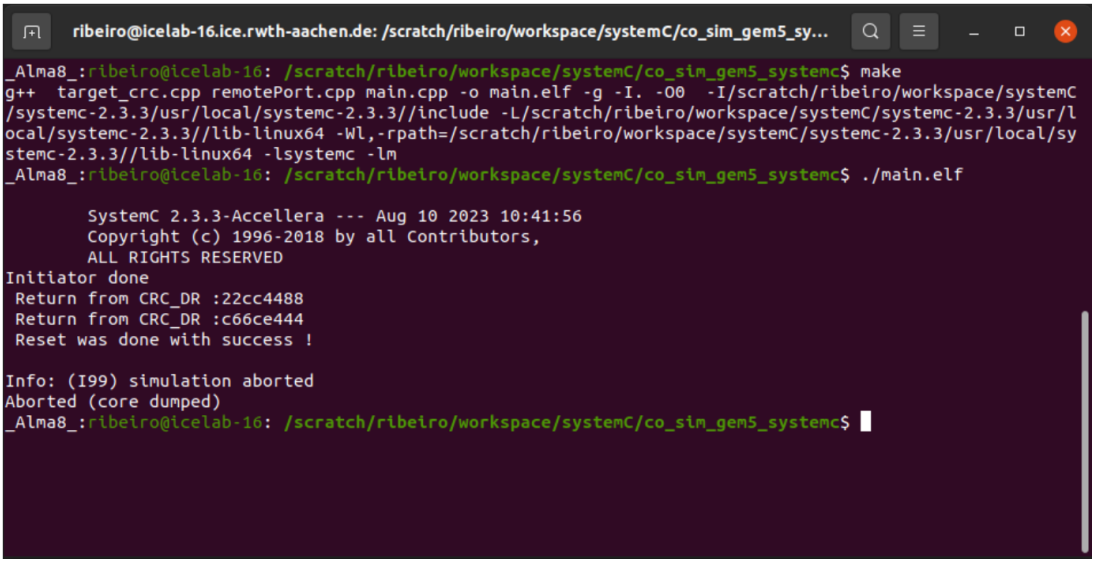


Figure 4.14: CRC peripheral validation results

Moving to the co-simulation validation test, two cores will be employed in Gem5's simulation for this purpose: one will be dedicated to the CRC, while the other will execute the bubble-sort benchmark. The simulation will be conducted in sequential mode, since the main objective is not performance, but accuracy.

It will be used one CRC and the UART, to communicate with the user by the terminal. As referred, UART is already implemented, hence, it only requires its initialization and configuration. For the test, it will be used the UART0, which is present in the memory map from 0x1C090000 to 0x1C09FFFF. To connect to the UART, the m5term can be utilized. It is a dedicated program that allows the user to connect to the simulated console interface. When executing the simulation, this program does not launch automatically, therefore, it must be manually called, with *./m5term <host> <port>*.

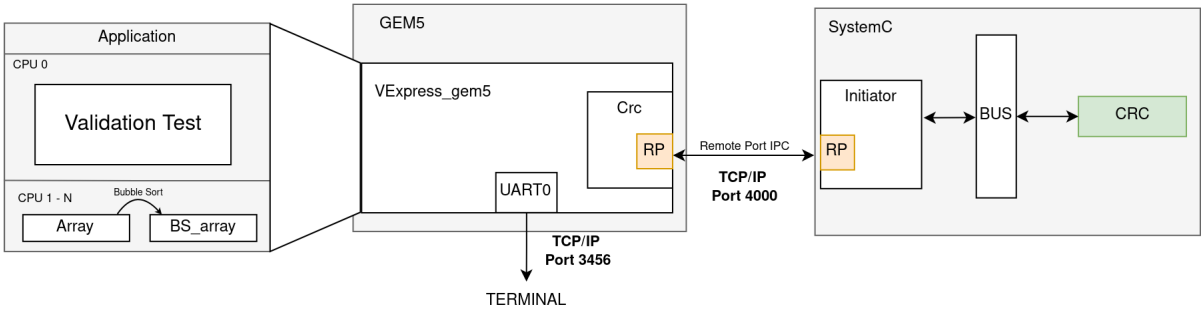


Figure 4.15: Validation co-simulation environment

The application will run the code present in 4.2, which describes the previous validation test.

```
CRC_SC_INIT();

CRC_SC_SET_REVERSE_OUTPUT(REV_WORD_OUT);
CRC_SC_SET_IDR(0x4C);
CRC_SC_VALUE_INPUT(0x15e32ef3);

do //Wait 4 tick
{
  CRC_value = CRC_SC_VALUE_OUTPUT();
} while (CRC_value == -EBUSY);

printf("Return from CRC_DR: %x \n", (uint32_t) CRC_value);

CRC_SC_SET_REVERSE_OUTPUT(NOT_REV_OUT);
CRC_SC_SET_REVERSE_INPUT(REV_HALF_WORD_IN);
CRC_SC_SET_POLYNOMIAL(0x12345678);
CRC_SC_VALUE_INPUT(0x1A2B3C4D);

do //Wait 4 tick
{
  CRC_value = CRC_SC_VALUE_OUTPUT();
} while (CRC_value == -EBUSY);

printf("Return from CRC_DR: %x \n", (uint32_t) CRC_value);

CRC_SC_SET_INIT(0x4C11DB7);

if(CRC_SC_GET_IDR() == 0x4C)
  CRC_SC_RESET();

if(CRC_SC_VALUE_OUTPUT() == CRC_SC_GET_POLYNOMIAL())
  printf("Reset was done with success! \n");
else
  printf("Failure in Reset \n");

break;
```

Code 4.2: CRC validation code

After executing the validation benchmark, the real results were the ones present in the Figure 4.16. It is possible to conclude that the peripheral passed the validation, since every expected output matches the real ones.

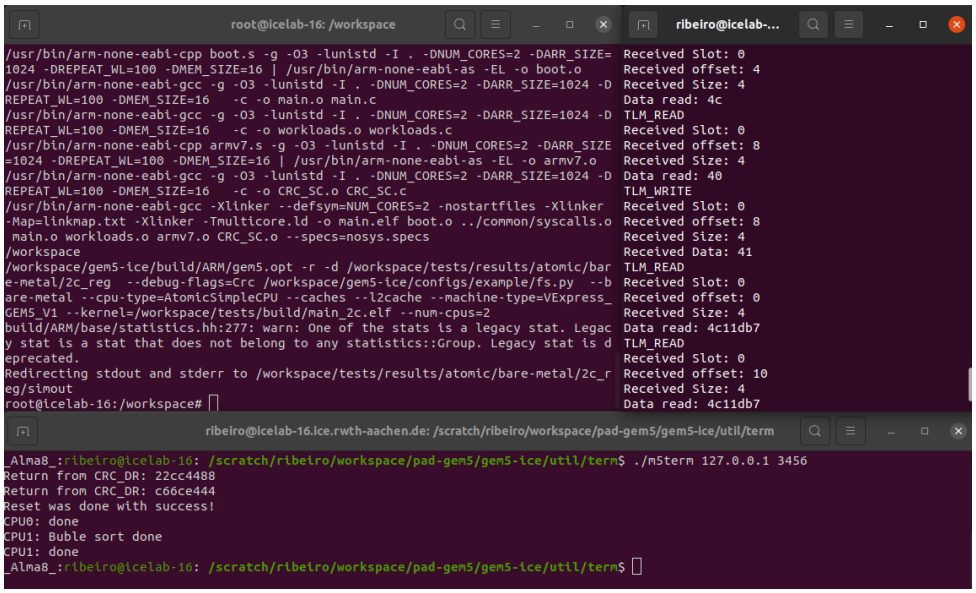


Figure 4.16: Co-simulation environment validation results

4.2.5 Memory Integrity

With the CRC's validation, it is possible to simulate the scenario described at the beginning of this section. Similar to the validation test, from the application point of view, the system will execute 2 distinct jobs. CPU0 will be responsible for performing the memory integrity checks, while the remaining ones will execute a bubble sort algorithm, as presented in the Figure 4.17.

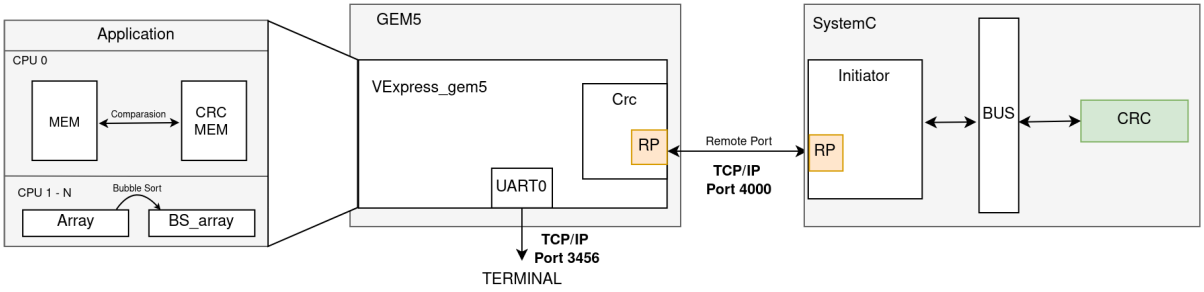


Figure 4.17: Case study co-simulation environment

When the benchmark starts, there is the creation of the intended CPUs. Each one has an ID, that allows one to identify themselves. To conduct the memory integrity test, CPU0 will start initializing the CRC and the memory that will be utilized to compare. Afterward, periodically, it will perform a memory comparison, having two possible outcomes. Either everything is all right and the operation is a success, or there is a flaw and the simulation ends immediately. A detailed view can be observed in 4.19.

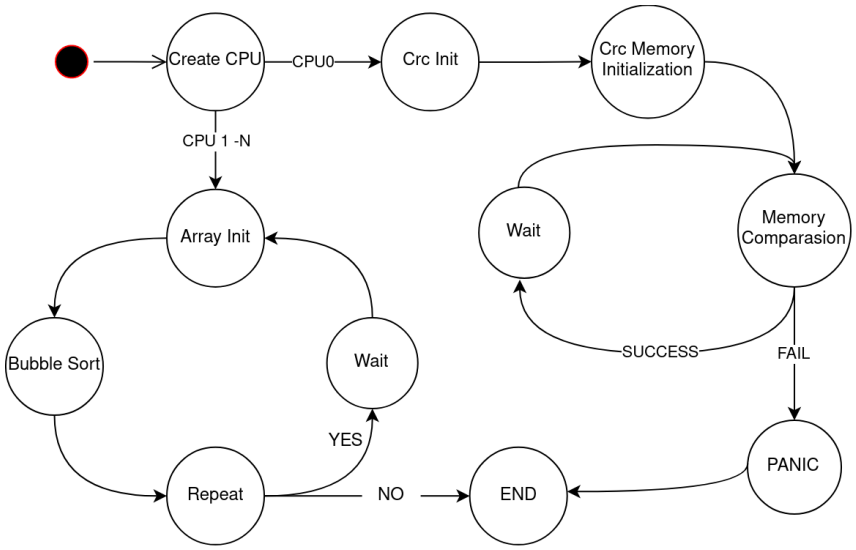


Figure 4.18: State machine diagram for the memory integrity test

The bubble-sort test runs in the remaining instaciated cores. Detail information about this benchmark can be found on subsection 3.1.2. If its workload is concluded successfully and there are still memory verifications to carry out, the simulation continues, since the primary objective is not to evaluate the bubble-sort algorithm.

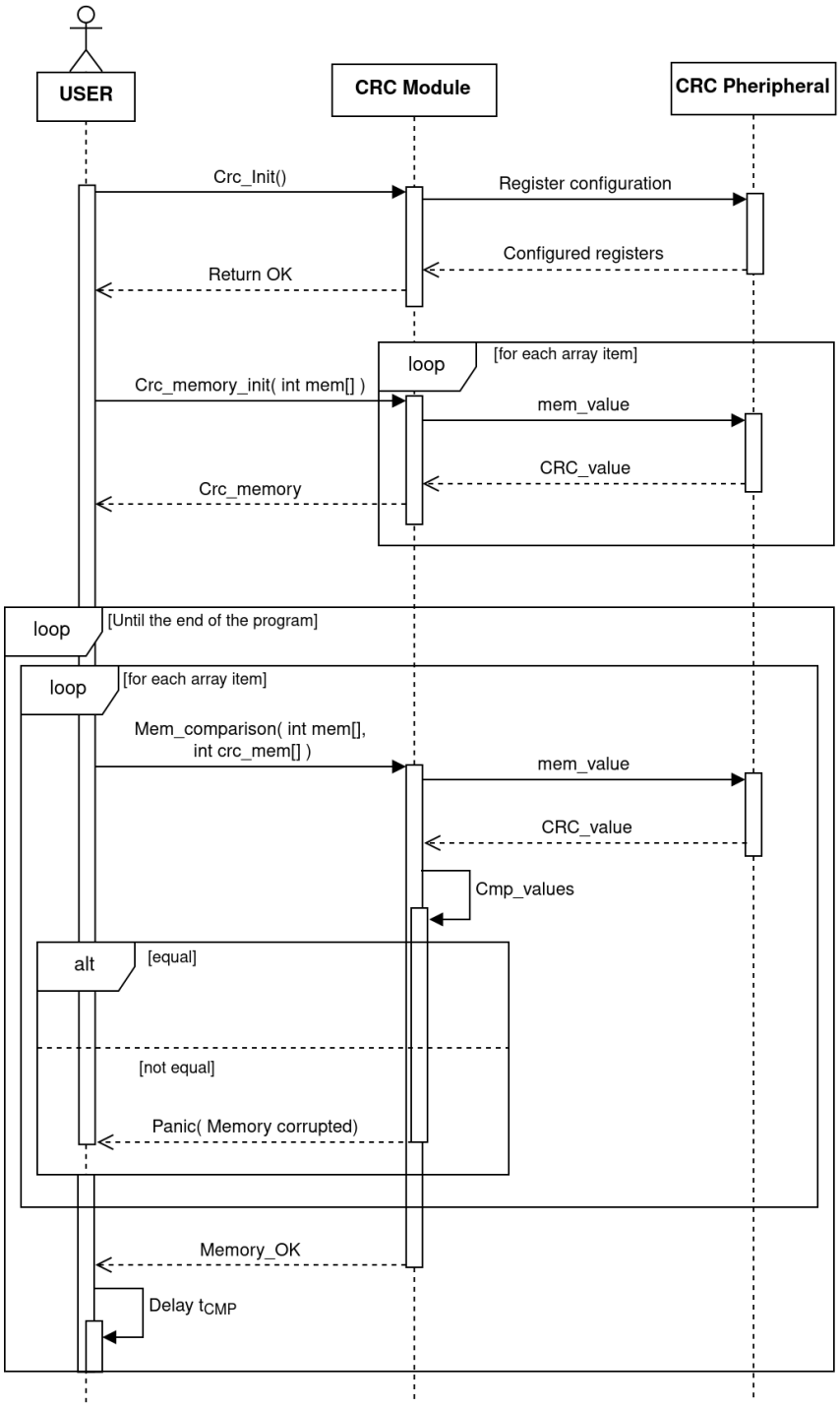


Figure 4.19: Sequence diagram diagram for the memory comparison

Success Modeling

In the first memory integrity test, the perfect scenario will be simulated. In other words, the memory will operate without any failures, ensuring a well-functioning system. As represented in the Figure 4.18, there will be an interval between both tasks, and this can be different from each other (4.20).

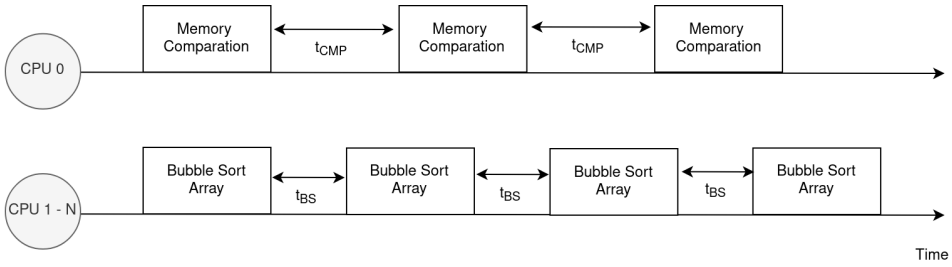


Figure 4.20: Application execution timely diagram

To perform this benchmark, the following parameters were defined. After completing the bubble-sort workload, the system executes a final memory check to ensure that no problems occurred between the last one, allowing the simulation to conclude safely. The simulation results are available in the subsequent images. As expected, the system executed normally the benchmark without any problems. Both CPUs conclude the designated tasks and the last memory comparison gives positive feedback.

- Repeat = 100
- t_{CMP} = 100 ms
- t_{BS} = 10 us
- Number of simulated cores = 2
- Memory size = 16
- Simulation mode = sequential
- Reverse input CRC = REV_HALF_WORD_IN
- Reverse output CRC = REV_WORD_OUT
- Polynomial = 0x04C11DB7

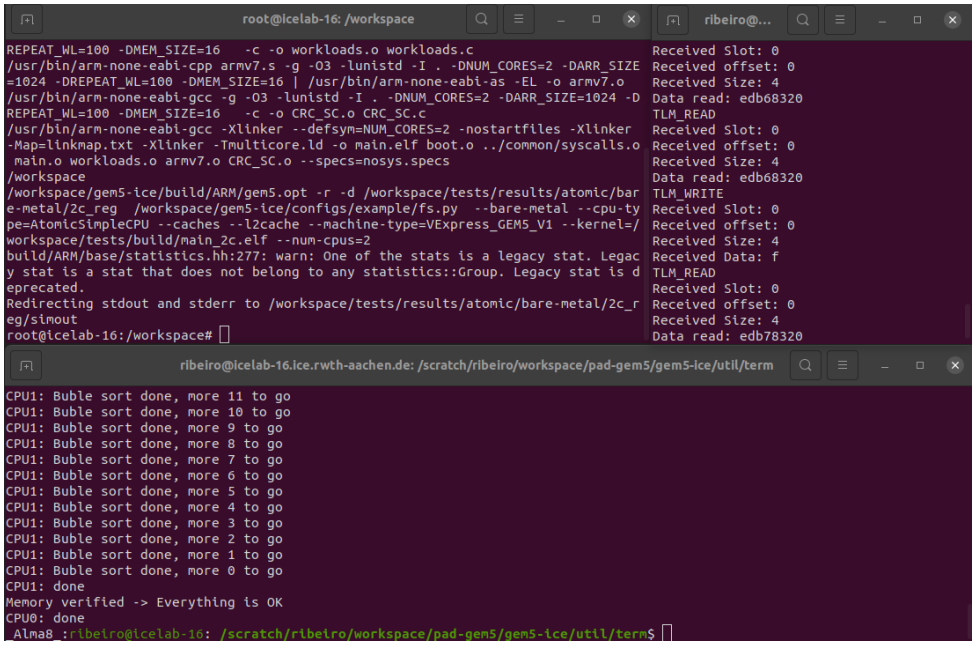


Figure 4.21: Success memory integrity test

Fault Modeling

In opposition to the first, this test will simulate a corrupted memory scenario, as shown in the Figure 4.22. In order to achieve that, a failure will be injected into the memory with the GNU Debugger (GDB) tool. GDB is a debugger that is supported by Gem5 and SystemC. It offers a variety of features but, for this purpose, the *set var* command will be utilized. This command allows a value modification of a variable in real-time, thus, in this way, the failure can be simulated.

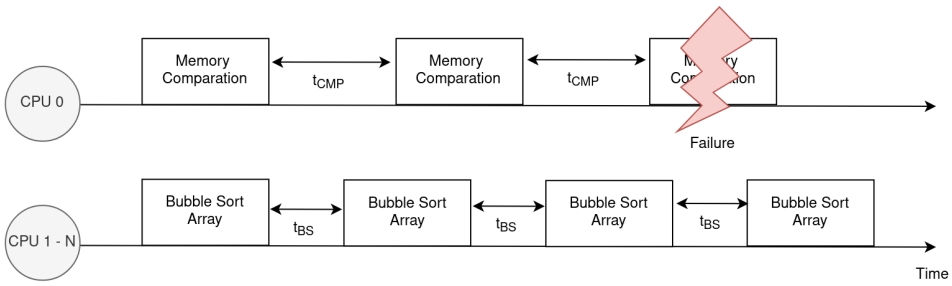
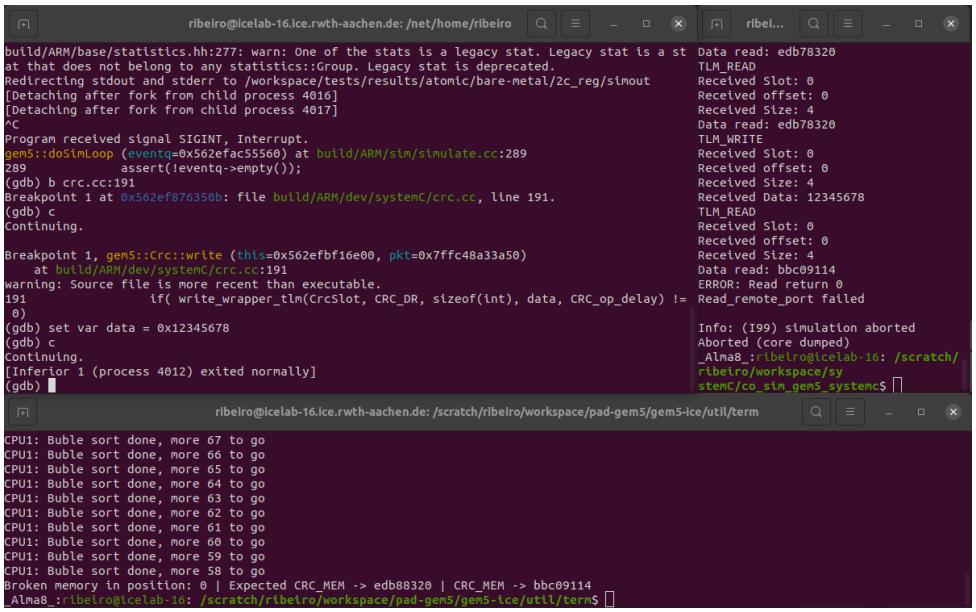
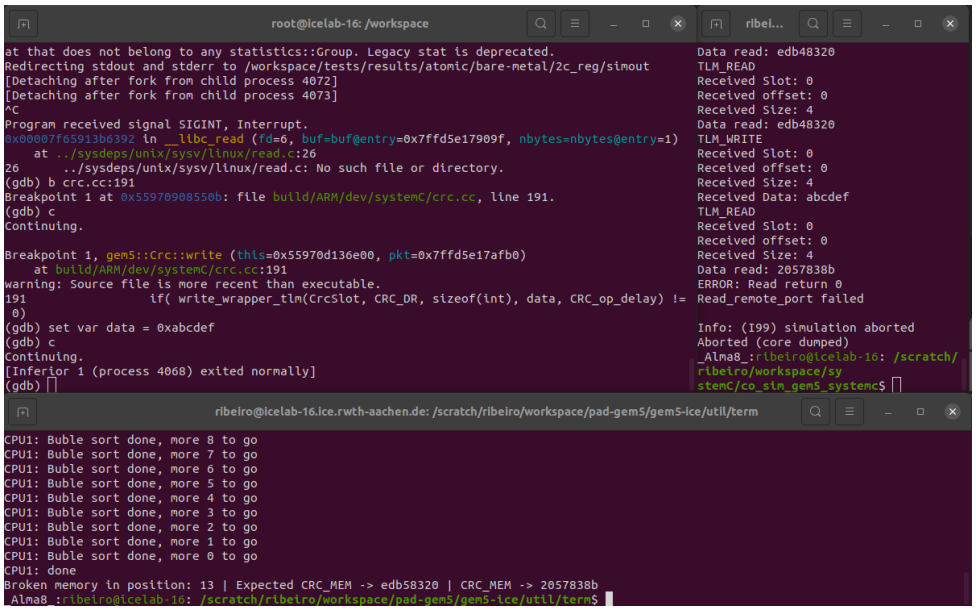


Figure 4.22: Application execution timely diagram with a failure

The test will be done under the same conditions as the previous one, nevertheless, in this case, for comprehensive debugging capabilities, the debug version of the Gem5 binary will be employed. In the end, the workload is expected to finish with an error from Gem5, and it can occur in two different moments. Either when the CPUs are still running, or when these already completed their tasks. Both cases will be tested, and, in either case, the program should terminate promptly. The Figure 4.23 demonstrates the obtained results, and it can be concluded that the system behaves in accordance with expectations.



(a) Case 1



(b) Case 2

Figure 4.23: Failure memory integrity test

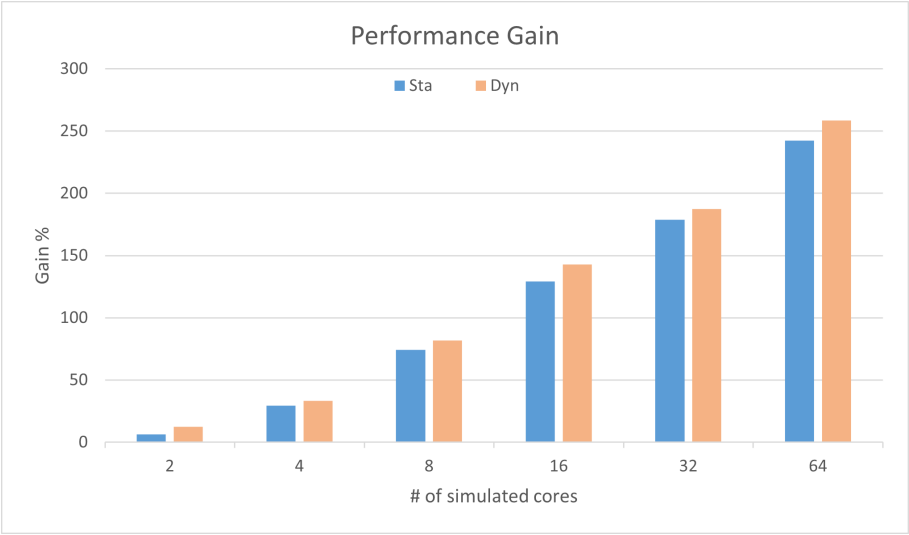
4.3 Dynamic Quantum Integration

Up to this point in the co-simulation, accuracy has been the primary concern. With this criteria, performance is sacrificed, giving a higher host time. In the co-simulation itself, the use of Par-gem5 would not provide greater benefits, as communication between the tools is the most time-consuming aspect. On the other hand, the remaining workload would take advantage of the parallel mode, as verified in the previous chapter (3.6).

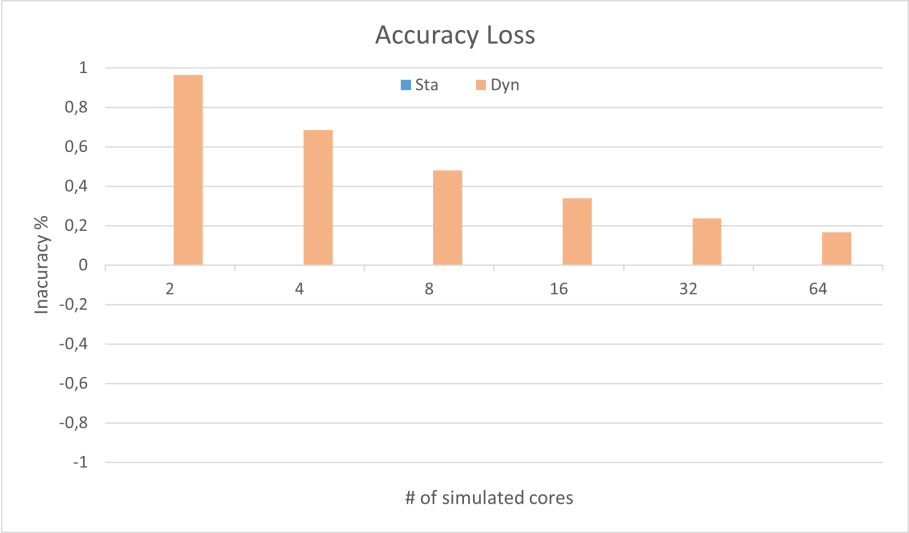
To assess the advantages of Par-gem5 and the dynamic quantum during co-simulation, a series of tests were conducted with various configurations. Compared to the prior ones, the differences will be resumed to the simulation modes, with the addition of static and dynamic parallel modes, and the number of simulated cores, which will range from 2, 4, 8, 16, 32, to 64 cores. Further, the same setup of the previous section was used to perform this new set of tests (Figure 4.17). The next figures exhibit the results obtained.

The graphs illustrate the gains in comparison to the sequential simulation. Observing the images can be settled that the parallel version, either with the static or dynamic version, had a better performance. With 64 simulated cores, the gains of the dynamic version overpassed 250%. In fact, it can be affirmed that the performance gains grow with the increasing number of cores, due to the rise of workload amount. In terms of accuracy, the static version had almost a perfect result, while the dynamic one had a maximum inaccuracy of 0,98%, nevertheless, it is considerably far from the 5%. Finally, as a consequence of the previously mentioned gains, the host time was reduced, getting, in the majority of the cases, at least, a reduction of 40%.

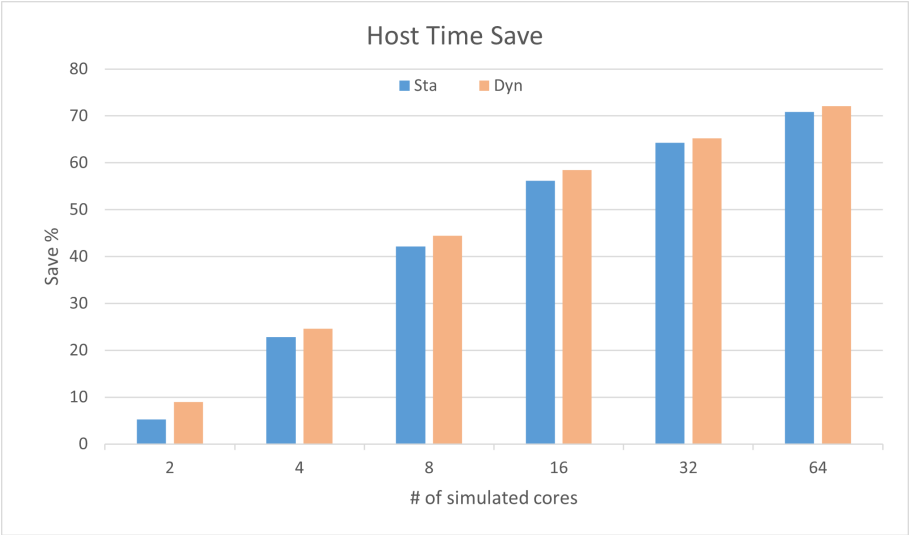
From these benchmarks other conclusions can be taken. Depending on the benchmark, the better



(a) Performance gain



(b) Accuracy lost



(c) Host time save

Figure 4.24: Co-simulation results

simulation mode can differ from the sequential and the parallel ones. It was verified that when the parallel tasks have considerable time consumption, the parallel mode gives more advantages, since it can accelerate them, keeping the inaccuracy below 5%. On the opposite way, if the communication between tools is where the simulation spends most of the time, the sequential mode will fit better, as it has a greater tradeoff between accuracy and performance. Further, the delay times between the tasks also play a significant role. In cases where $T_{CoSim} >>> T_{ParTask}$, the parallel version is the most appropriate, otherwise, the sequential one should be chosen.

In the end, to decide which mode is better for a certain workload, *a priori* information is needed. If it does not exist, either because there is no documentation or the access to the source code is restricted, the better simulation mode to use, based on this case study, is the parallel mode with a dynamic quantum. The reason for that is that it always had an inaccuracy below 5% and had a higher performance gain when compared to the sequential version. Nevertheless, if perfect accuracy is the main requirement, the sequential simulation mode should always be chosen.

5 Conclusions

This chapter concludes this dissertation, summarizing the former developed work, and giving up some topics that could be interesting for future work.

From a general perspective, the development of this project contributed to solve the principal problem verified in Par-gem5, which is the quantum definition to get the best tradeoff between performance and accuracy. Further, it provided a flexible and well-functioning framework that enables the interaction between different tools. With these advancements, it is no longer required to study and understand the target system and benchmarks in detail, and Gem5 can now be utilized in co-simulation environments more frequently. For this reason, the dissertation goal was accomplished.

5.1 Developed Work

Concerning the dynamic quantum development, there were designed and tested four different algorithms. The ADALINE-based algorithm demonstrated the capability to adapt the quantum without any prior information and proved to be lightweight; The Step Ladder usage enabled a higher performance gain in the majority of the cases, decreasing host time; IFP allowed a deeper analysis of the simulation flow, highlighting potential regions where accuracy could be affected; And the Loop Detection algorithm provided loop identification while executing the benchmark, creating the possibility of adapting the quantum for the best value. After gathering all the results, the final decision was to incorporate all algorithms, with the exclusion of the loop detection algorithm. Due to its overhead in the simulation performance and weak accuracy gain, it was excluded from the final solution. However, it was shown that it can be an option when the workload has lots of repetitive tasks, enabling specific optimizations.

Several tests were performed to evaluate it. In the end, it can be concluded that the use of the dynamic version for the quantum choice brings greater benefits. When the NPB BT, LU, and SP with 32 and 64 simulated cores are not considered, there are gains in performance almost reaching 10%, only sacrificing 0.5% of accuracy. As mentioned on subsection 3.1.2, the previous tests have a particular characteristic, which is that they execute nonlinear partial differential equations. This workload does not require inter-process communications, thus, quantum can be increased without losing significant accuracy. The dynamic algorithm was designed to make it harder to increase the quantum value as the number of cores

increases, due to the conclusions obtained in the Par-gem5 [4] work. For this reason, this type of benchmark obtains a meaningful drop in performance when more than 16 cores are being simulated. However, the algorithm's design must be flexible, in a way that all benchmarks can have the best performance within accuracy limits.

Negative inaccuracy may continue to be present, as a consequence of Par-gem5 use. The dynamic quantum development was not able to solve this issue, making necessary a deeper analysis of Par-gem5 design in order to find the problem.

In conclusion, the better algorithm depends on the *a priori* available information. If there is no information available, the dynamic version should be used because, even if the performance might be lower than it could be, it is guaranteed that accuracy will be above 95%. On the other side, knowing the details of the benchmark enables us to calculate the optimal quantum before initiating the simulation. It is important to remember that when increasing the quantum, while performance as a function of the quantum behaves similarly to a sigmoidal, inaccuracy grows linearly [63]. If perfect accuracy is mandatory, the sequential mode should be selected, since the usage of Par-gem5 implies an accuracy cost. Also, simulations with a single simulated core should always be executed with the later mode. It was concluded that parallelization with only one simulated core results in a loss of accuracy without a significant gain in performance.

The developed co-simulation interface provided a new work environment between two different tools. Gem5 and SystemC were chosen to validate this, and the result was positive. Data integrity, data exchange, and synchronization between the tools were maintained during all the simulations. As a case study to stimulate this cooperation, the CRC device was chosen, passing all the requirements successfully. In addition, as part of the case study, memory integrity tests were designed and performed, where either for the success or the failure modeling, the expected results matched with the real ones. On the other part, it was possible to verify that the dynamic quantum yielded improved results even with a significant time consumption in the communication between tools. Further, accuracy was always above 99%, enchanting its performance.

It is important to mention that this work regarding co-simulation between Gem5 and other tools is very poor, only existing a few works in the literature. Gem5 already has a defined co-simulation environment with SystemC in its official repository, nevertheless, it does not fully adhere to the previously mentioned co-simulation definition because the two simulators are not running simultaneously in separate processes. For this reason, this work contributes to the first steps in this subject.

5.2 Future Work

Concerning future work, there are some aspects where improvements and new additions can be developed to have either an algorithm or a framework more robust and versatile.

The first aspect concerns the execution of more benchmarks. Although the algorithm was tested with a set of different and distinct benchmarks, more tests are required to have more performance results. These

may reveal points where it can be improved, either with the development of a new support algorithm or with a better tunning of the available parameters. PARSEC, SPEC2017, and STREAM are some examples of benchmarks where experiments can be conducted.

The second aspect regards the adaptation for the timing CPU model. A future work of Par-gem5 [4] consists of extending this solution to the timing mode, thus, the developed approach should also function in this condition. Only the IFP algorithm requires an adaptation, due to its PC analysis method. The timing mode uses multiple events to model a transaction, which makes it difficult to perform a correct examination of the executed instructions. In-order and O3 demand an adaptation as well, however, until the present date, no publications have been made on this matter regarding Par-gem5.

Another point to highlight is the usage of other targets. The developed algorithm was designed to tackle any target board, nevertheless, as aforementioned, only VExpress_gem5 was used for all the tests. Different boards with different characteristics may highlight flaws that could not be spotted with the present one. Such usage may require its design and development in the platform with the creation of all the necessary devices and connections.

Finally, improvements in the co-simulation environment can be implemented. The main problem observed was the time consumption in the communication between tools. From a host time perspective, the workload on the memory comparison task corresponds to 87 % of the whole simulation, where 82.2% of this value, on average, corresponds to the waiting time for the payload response. As part of the solution to speed up co-simulation, other IPCs techniques can be used, like shared memory. Although it is the fastest IPC available, it is the unsafest one due to the lack of permission verifications, hence, for this reason, a careful design must be employed. Nevertheless, it might not be enough to solve the problem, and other approaches must be considered.

References

[1] B. Liu, H. Zhang, and S. Zhu, "An incremental v-model process for automotive development," in *2016 23rd Asia-Pacific Software Engineering Conference (APSEC)*, IEEE, 2016, pp. 225–232.

[2] R. P. A. A. F. A. N. C. O. O. Sugiono Sugiono Andi S. Putra, "New concept of product design by involving emotional factors using eeg: A case study of xomputer mouse design," *ACTA NEUROPSYCHOLOGICA*, vol. 19, no. 1, pp. 63–80, 2021.

[3] D. N. J. A. K. P. K. S. P. I. S. A. S. V. Mani Azimi Naveen Cherukuri, "Tera-scale computing," *Intel Technology Journal*, vol. 11, no. 3, pp. 173–184, 2007.

[4] N. Zurstraßen, C.-C. Jose, J. M. Joseph, R. Leupers, X. Xinghua, and L. Yichao, "Par-gem5: Parallelizing gem5 s atomic mode," English, in *2023 Design, Automation and Test in Europe Conference (DATE)*, 2023.

[5] C. Gomes, C. Thule, D. Broman, P. G. Larsen, and H. Vangheluwe, "Co-simulation: State of the art," *arXiv preprint arXiv:1702.00686*, 2017.

[6] Z. M. Research, *Virtual training and simulation market size, growth, trends, share*.

[7] S. Robinson, *Simulation: The Practice of Model Development and Use*. Hoboken, NJ, USA: John Wiley & Sons, Inc., 2004, ISBN: 0470847727.

[8] M. Verkuyl, N. Dubois, S. Goldsworthy, T. Merwin, T. Willet, and T. Job, *Virtual Simulation: An Educator's Toolkit*. 2022.

[9] J. Banks, "Introduction to simulation," in *Proceedings of the 31st conference on Winter simulation: Simulation—a bridge to the future-Volume 1*, 1999, pp. 7–13.

[10] J. Banks, J. Carson, B. Nelson, and D. Nicol, *Discrete-Event System Simulation*, English, 5th ed. Prentice Hall, 2010, ISBN: 0136062121.

[11] M. Barr, *Programming Embedded Systems in C and C++* (O'Reilly Series). O'Reilly, 1999, ISBN: 9781565923546.

[12] R. Camposano and J. Wilberg, "Embedded system design," *Design Automation for Embedded Systems*, vol. 1, pp. 5–50, 1996.

[13] J. Corbet, A. Rubini, and G. Kroah-Hartman, *Linux Device Drivers, 3rd Edition*. O'Reilly Media, Inc., 2005, ISBN: 0596005903.

[14] X. Zhai, F. Bensaali, and K. McDonald-Maier, "Automatic number plate recognition on fpga," in *2013 IEEE 20th International Conference on Electronics, Circuits, and Systems (ICECS)*, 2013, pp. 325–328. DOI: 10.1109/ICECS.2013.6815420.

[15] J. Cong and J. Peck, "On acceleration of the check tautology logic synthesis algorithm using an fpga-based reconfigurable coprocessor," in *Proceedings. The 5th Annual IEEE Symposium on Field-Programmable Custom Computing Machines Cat. No.97TB100186)*, 1997, pp. 246–247. DOI: 10.1109/FPGA.1997.624629.

[16] J. Cong and J. Peck, "On acceleration of the check tautology logic synthesis algorithm using an fpga-based reconfigurable coprocessor," in *Proceedings. The 5th Annual IEEE Symposium on Field-Programmable Custom Computing Machines Cat. No.97TB100186)*, 1997, pp. 246–247. DOI: 10.1109/FPGA.1997.624629.

[17] M. Silva, D. Cerdeira, S. Pinto, and T. Gomes, "Operating systems for internet of things low-end devices: Analysis and benchmarking," *IEEE Internet of Things Journal*, vol. 6, no. 6, pp. 10 375–10 383, 2019.

[18] IEC, *Iec, 192-01-22 dependability*.

[19] A. Tanenbaum and H. Bos, *Modern Operating Systems, Global Edition*. Pearson Education, 2015, ISBN: 9781292061955.

[20] R. Barry, *Mastering the freertos real time kernel. real time engineers ltd*, 2016.

[21] A. A. Adenowo and B. A. Adenowo, "Software engineering methodologies: A review of the waterfall model and object-oriented approach," *International Journal of Scientific & Engineering Research*, vol. 4, no. 7, pp. 427–434, 2013.

[22] W Royce, "Winston," *Proceedings, Managing the Development of Large Software Systems, IEEE WESCON*, 1970.

[23] S. Balaji and M. S. Murugaiyan, "Waterfall vs. v-model vs. agile: A comparative study on sdlc," *International Journal of Information Technology and Business Management*, vol. 2, no. 1, pp. 26–30, 2012.

[24] G. B. Regulwar, P. Deshmukh, R. Tugnayat, P. Jawandhiya, and V. Gulhane, "Variations in v model for software development," *International Journal of Advanced Research in Computer Science*, vol. 1, no. 2, pp. 134–135, 2010.

[25] S. Mathur and S. Malik, "Advancements in the v-model," *International Journal of Computer Applications*, vol. 1, no. 12, pp. 29–34, 2010.

[26] W. Van Casteren, "The waterfall model and the agile methodologies: A comparison by project characteristics," *Research Gate*, vol. 2, pp. 1–6, 2017.

[27] B. Boehm, "A survey of agile development methodologies," *Laurie Williams*, vol. 45, p. 119, 2007.

[28] B. W. Boehm, “A spiral model of software development and enhancement,” *Computer*, vol. 21, no. 5, pp. 61–72, 1988.

[29] W. M. Zabołotny, “Development of embedded pc and fpga based systems with virtual hardware,” in *Photonics Applications in Astronomy, Communications, Industry, and High-Energy Physics Experiments 2012*, SPIE, vol. 8454, 2012, pp. 259–265.

[30] *Ghdl main/home page*.

[31] S. Williams and M. Baxter, “Icarus verilog: Open-source verilog more than a year later,” *Linux Journal*, vol. 2002, no. 99, p. 3, 2002.

[32] F. Bellard, “Qemu, a fast and portable dynamic translator.,” in *USENIX annual technical conference, FREENIX Track*, Califor-nia, USA, vol. 41, 2005, p. 46.

[33] N. Binkert *et al.*, “The gem5 simulator,” *SIGARCH Comput. Archit. News*, vol. 39, no. 2, pp. 1–7, 2011, ISSN: 0163-5964. DOI: 10.1145/2024716.2024718.

[34] O. Bringmann *et al.*, “The next generation of virtual prototyping: Ultra-fast yet accurate simulation of hw/sw systems,” in *2015 Design, Automation & Test in Europe Conference & Exhibition (DATE)*, IEEE, 2015, pp. 1698–1707.

[35] L. Jünger, C. Bianco, K. Niederholtmeyer, D. Petras, and R. Leupers, “Optimizing temporal decoupling using event relevance,” in *Proceedings of the 26th Asia and South Pacific Design Automation Conference*, 2021, pp. 331–337.

[36] A. Varga, “Discrete event simulation system,” in *Proc. of the European Simulation Multiconference (ESM’2001)*, 2001, pp. 1–7.

[37] E. Babulak and M. Wang, “Discrete event simulation,” *Aitor Goti (Hg.): Discrete Event Simulations. Rijeka, Kroatien: Sciyo*, p. 1, 2010.

[38] W Boughton and O Droop, “Continuous simulation for design flood estimation—a review,” *Environmental Modelling & Software*, vol. 18, no. 4, pp. 309–318, 2003.

[39] F. Henning, L. Kärger, D. Dörr, F. J. Schirmaier, J. Seuffert, and A. Bernath, “Fast processing and continuous simulation of automotive structural composite components,” *Composites Science and Technology*, vol. 171, pp. 261–279, 2019.

[40] M. Helal, *A hybrid system dynamics-discrete event simulation approach to simulating the manufacturing enterprise*. University of Central Florida, 2008.

[41] T. Q. P. Developers, *Qemu’s documentation*.

[42] C. Schumacher, R. Leupers, D. Petras, and A. Hoffmann, “Parsc: Synchronous parallel systemc simulation on multi-core host architectures,” in *Proceedings of the eighth IEEE/ACM/IFIP international conference on hardware/software codesign and system synthesis*, 2010, pp. 241–246.

[43] "Ieee standard for standard systemc language reference manual," *IEEE Std 1666-2011 (Revision of IEEE Std 1666-2005)*, pp. 1–638, 2012. DOI: 10.1109/IEEESTD.2012.6134619.

[44] A. Yoga and S. Nagarakatte, "Parallelism-centric what-if and differential analyses," in *Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation*, 2019, pp. 485–501.

[45] T. Zhou, *Sequential and parallel discrete event simulation on computer communication networks*. Western Michigan University, 1992.

[46] N. R. Tallent, L. Adhianto, and J. M. Mellor-Crummey, "Scalable identification of load imbalance in parallel executions using call path profiles," in *SC '10: Proceedings of the 2010 ACM/IEEE International Conference for High Performance Computing, Networking, Storage and Analysis*, 2010, pp. 1–11. DOI: 10.1109/SC.2010.47.

[47] L. Rose, B. Homer, and D. Johnson, "Detecting application load imbalance on high end massively parallel systems," Aug. 2007, pp. 150–159, ISBN: 978-3-540-74465-8. DOI: 10.1007/978-3-540-74466-5_17.

[48] K.-Y. Chen, J. M. Chang, and T.-W. Hou, "Multithreading in java: Performance and scalability on multicore systems," *IEEE Transactions on Computers*, vol. 60, no. 11, pp. 1521–1534, 2010.

[49] K. Du Bois, J. B. Sartor, S. Eyerman, and L. Eeckhout, "Bottle graphs: Visualizing scalability bottlenecks in multi-threaded applications," *ACM SIGPLAN Notices*, vol. 48, no. 10, pp. 355–372, 2013.

[50] S. Eyerman, K. Du Bois, and L. Eeckhout, "Speedup stacks: Identifying scaling bottlenecks in multi-threaded applications," Apr. 2012. DOI: 10.1109/ISPASS.2012.6189221.

[51] R. Ciesla, "Bits, sample rates, and other fundamentals of digital audio," in *Sound and Music for Games: The Basics of Digital Audio for Video Games*, Springer, 2022, pp. 1–24.

[52] R. M. Fujimoto, "Parallel discrete event simulation," *Communications of the ACM*, vol. 33, no. 10, pp. 30–53, 1990.

[53] G. Busnot, T. Sassolas, N. Ventroux, and M. Moy, "Standard-compliant parallel systemc simulation of loosely-timed transaction level models," in *2020 25th Asia and South Pacific Design Automation Conference (ASP-DAC)*, IEEE, 2020, pp. 363–368.

[54] M. Jung, F. Schnicke, M. Damm, T. Kuhn, and N. Wehn, "Speculative temporal decoupling using fork ()," in *2019 Design, Automation & Test in Europe Conference & Exhibition (DATE)*, IEEE, 2019, pp. 1721–1726.

[55] A. Mohammad, U. Darbaz, G. Dozsa, S. Diestelhorst, D. Kim, and N. S. Kim, "Dist-gem5: Distributed simulation of computer clusters," in *2017 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*, IEEE, 2017, pp. 153–162.

[56] J. H. Weinstock, R. Leupers, G. Ascheid, D. Petras, and A. Hoffmann, "Systemc-link: Parallel systemc simulation using time-decoupled segments," in *2016 Design, Automation & Test in Europe Conference & Exhibition (DATE)*, IEEE, 2016, pp. 493–498.

[57] J. Lowe-Power *et al.*, "The gem5 simulator: Version 20.0+," *arXiv preprint arXiv:2007.03152*, 2020.

[58] N. Binkert, R. Dreslinski, L. Hsu, K. Lim, A. Saidi, and S. Reinhardt, "The m5 simulator: Modeling networked systems," *IEEE Micro*, vol. 26, no. 4, pp. 52–60, 2006. DOI: 10.1109/MM.2006.82.

[59] M. M. K. Martin *et al.*, "Multifacet's general execution-driven multiprocessor simulator (gems) toolset," vol. 33, no. 4, pp. 92–99, 2005, ISSN: 0163-5964. DOI: 10.1145/1105734.1105747.

[60] D.-I. G. Hempel and I. J. Castrillon, "Simulation of risc-v based systems in gem5,"

[61] W. Jakob, J. Rhinelander, and D. Moldovan, "Pybind11–seamless operability between c++ 11 and python," *URL https://github.com/pybind/pybind11*, 2022.

[62] S. Knight, "Scons design and implementation," in *Tenth int'l python conf*, 2002.

[63] N. Zurstraßen, R. Brandhofer, J. C. Cascante, J. M. Joseph, N. Bosbach, and R. Leupers, *Beyond the Quantum of Temporally-Decoupled Simulations*,

[64] G. Glaser, G. Nitsche, and E. Hennig, "Temporal decoupling with error-bounded predictive quantum control," in *2015 Forum on Specification and Design Languages (FDL)*, 2015, pp. 1–6.

[65] D. R. Jefferson, "Virtual time," *ACM Transactions on Programming Languages and Systems (TOPLAS)*, vol. 7, no. 3, pp. 404–425, 1985.

[66] F. Morales and J. L. Bismarck, *Evaluating gem5 and qemu virtual platforms for arm multicore architectures*, 2016.

[67] A. Herrera, *"running trusted firmware-a on gem5*, 2020.

[68] T. Wieman, B. Bhattacharya, T. Jeremiassen, C. Schroder, and B. Vanthournout, "An overview of open systemc initiative standards development," *IEEE Design & Test of Computers*, vol. 29, no. 2, pp. 14–22, 2012.

[69] A. Akram and L. Sawalha, "A comparison of x86 computer architecture simulators," 2016.

[70] R. Ubal, B. Jang, P. Mistry, D. Schaa, and D. Kaeli, "Multi2sim: A simulation framework for cpu-gpu computing," in *Proceedings of the 21st international conference on Parallel architectures and compilation techniques*, 2012, pp. 335–344.

[71] T. E. Carlson, W. Heirman, and L. Eeckhout, "Sniper: Exploring the level of abstraction for scalable and accurate parallel multi-core simulation," in *Proceedings of 2011 International Conference for High Performance Computing, Networking, Storage and Analysis*, 2011, pp. 1–12.

[72] M. T. Yourst, "Ptlsim: A cycle accurate full system x86-64 microarchitectural simulator," in *2007 IEEE International Symposium on Performance Analysis of Systems & Software*, IEEE, 2007, pp. 23–34.

[73] D. Sanchez and C. Kozyrakis, "Zsim: Fast and accurate microarchitectural simulation of thousand-core systems," *ACM SIGARCH Computer architecture news*, vol. 41, no. 3, pp. 475–486, 2013.

[74] A. Patel, F. Afram, and K. Ghose, "Marss-x86: A qemu-based micro-architectural and systems simulator for x86 multicore processors," in *1st International Qemu Users' Forum*, Citeseer, 2011, pp. 29–30.

[75] M. R. Bachute and J. M. Subhedar, "Autonomous driving architectures: Insights of machine learning and deep learning algorithms," *Machine Learning with Applications*, vol. 6, p. 100 164, 2021.

[76] A. L. Samuel, "Some studies in machine learning using the game of checkers," *IBM Journal of research and development*, vol. 3, no. 3, pp. 210–229, 1959.

[77] B. Mahesh, "Machine learning algorithms-a review," *International Journal of Science and Research (IJSR).[Internet]*, vol. 9, pp. 381–386, 2020.

[78] S. Haykin, *Neural networks and learning machines, 3/E*. Pearson Education India, 2009.

[79] B. Widrow and M. E. Hoff, "Adaptive switching circuits," Stanford Univ Ca Stanford Electronics Labs, Tech. Rep., 1960.

[80] B. Widrow and S. D. Stearns, "Adaptive signal processing prentice-hall," *Englewood Cliffs, NJ*, p. 52, 1985.

[81] B. Widrow and M. A. Lehr, "Perceptrons, adalines, and backpropagation," *Arbib*, vol. 4, pp. 719–724, 1995.

[82] K. Mitchell-Wallace, M. Jones, J. Hillier, and M. Foote, *Natural catastrophe risk management and modelling: A practitioner's guide*. John Wiley & Sons, 2017.

[83] L. von Rueden, S. Mayer, R. Sifa, C. Bauckhage, and J. Garcke, "Combining machine learning and simulation to a hybrid modelling approach: Current and future directions," in *Advances in Intelligent Data Analysis XVIII: 18th International Symposium on Intelligent Data Analysis, IDA 2020, Konstanz, Germany, April 27–29, 2020, Proceedings 18*, Springer, 2020, pp. 548–560.

[84] P. Benner, S. Gugercin, and K. Willcox, "A survey of projection-based model reduction methods for parametric dynamical systems," *SIAM review*, vol. 57, no. 4, pp. 483–531, 2015.

[85] E. Tsymbalov, S. Makarychev, A. Shapeev, and M. Panov, "Deeper connections between neural networks and gaussian processes speed-up active learning," *arXiv preprint arXiv:1902.10350*, 2019.

[86] F. Noé, A. Tkatchenko, K.-R. Müller, and C. Clementi, "Machine learning for molecular simulation," *Annual review of physical chemistry*, vol. 71, pp. 361–390, 2020.

[87] K. Albertsson *et al.*, "Machine learning in high energy physics community white paper," in *Journal of Physics: Conference Series*, IOP Publishing, vol. 1085, 2018, p. 022 008.

[88] Xilinx, *Xilinx/libsystemctlm-soc: Systemc/tlm-2.0 co-simulation framework*.

[89] M. Komalan *et al.*, “Main memory organization trade-offs with dram and stt-mram options based on gem5-nvmain simulation frameworks,” in *2018 Design, Automation & Test in Europe Conference & Exhibition (DATE)*, IEEE, 2018, pp. 103–108.

[90] C. Menard, J. Castrillon, M. Jung, and N. 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)*, IEEE, 2017, pp. 62–69.

[91] D. Bailey *et al.*, “The nas parallel benchmarks rnr-94-007,” *NASA Advanced Supercomputing Division, Tech. Rep*, 1994.

[92] S. Haykin, “Linear prediction,” *Adaptive filter theory*, pp. 562–588, 1996.

[93] M. Stella, D. Begusic, and M. Russo, “Adaptive noise cancellation based on neural network,” in *2006 International Conference on Software in Telecommunications and Computer Networks*, 2006, pp. 306–309. DOI: 10.1109/SOFTCOM.2006.329765.

[94] *Rm0385 reference manual -based 32-bit mcus*.

[95] G Xilinx and S Guide, “Zynq-7000 all programmable soc technical reference manual (ug585),” Tech. rep., Xilinx, 2014, https://www. xilinx. com/support/documentation ..., Tech. Rep., 2014.

[96] W. W. Peterson and D. T. Brown, “Cyclic codes for error detection,” *Proceedings of the IRE*, vol. 49, no. 1, pp. 228–235, 1961.

[97] C. Borrelli, “Ieee 802.3 cyclic redundancy check,” *application note: Virtex Series and Virtex-II Family, XAPP209 (v1. 0)*, 2001.