FACULDADE DE ENGENHARIA DA UNIVERSIDADE DO PORTO

Task Allocation and Resource Optimization for Transparent and Distributed Cyber-Physical Systems

José Rafael Fidalgo Fonseca Matias



Programa Doutoral em Engenharia Informática

Orientador: Gil Gonçalves and Vítor Pinto

25 de julho de 2022

Task Allocation and Resource Optimization for Transparent and Distributed Cyber-Physical Systems

José Rafael Fidalgo Fonseca Matias

Programa Doutoral em Engenharia Informática

25 de julho de 2022

Resumo

Existem diferentes áreas da indústria que efrentam problemas relacionados com a alocação de tarefas e otimização de recursos, tais como a área dos sistemas de transporte, "smart grids" e fabrico industrial. Da mesma forma, existem inúmeros algoritmos que lidam com estas dificuldades, mas raramente é possível integra-los em Sistemas Ciber-Físicos de Produção (CPPSs). Atualmente, as ferramentas e plataformas usadas para implementar CPPSs não integram mecanismos inteligentes para a gestão de dispositivos, impedindo o sistema de ser capaz de lidar com perturbações externas nem de otimizar a sua performance.

O elevado número de dispositivos interconectados traz novos desafios para a reconfiguração e reorganização de um CPPS. A velocidade e agilidade dos processos de desenvolvimento e partilha dos CPPSs são essenciais para permitir lidar com quaisquer mudanças que possam ocorrer, como acontece quando se introduzem novos proecssos ou tarefas.

Um dos principais problemas aquando do desenvolvimento e implementação de CPPSs é o da alocação de tarefas de software aos dispositivos disponiveis sem comprometer a fluidez da execução de tarefas no sistema. Este problema é particularmente relevante, porque uma má alocação de tarefas a dispositivos tem como consequência o abrandamento da performance de todo o sistema, consumindo vários recursos computacionais (ex: CPU, RAM). Além disso, os CPPSs consistem frequentemente em redes de dispositivos com recursos computacionais beixos, fazendo com que uma alocação errónea possa por a execução da "pipeline" em risco.

Um dos principais problemas aquando do desenvolvimento e implementação de CPPSs é o da alocação de tarefas de software aos dispositivos disponiveis sem comprometer a fluidez da execução de tarefas no sistema. Este problema é particularmente relevante, porque uma má alocação de tarefas a dispositivos tem como consequência o abrandamento da performance de todo o sistema, consumindo vários recursos computacionais (ex: CPU, RAM). Além disso, os CPPSs consistem frequentemente em redes de dispositivos com recursos computacionais beixos, fazendo com que uma alocação errónea possa por a execução da "pipeline" em risco.

Esta dissertação pega no DINASORE, Pereira et al. (2020), uma ferramenta que permite partilhar "pipelines" de software com os dispositivos que compõem um CPPS. Um conjunto de funcionalidades foram desenvolvidas que tornaram o DINASORE uma ferramente que otimiza a alocação de tarefas aos dispositivos de um CPPS, minimizando os recursos consumidos pelos dispositivos. No que toca à metodologia adotada, o programa monitoriza a execução de blocos de código em cada dispositivo e recolhe métricas relevantes, tais como o tempo de execução, o tempo de comunicação e os níveis de consumo de recursos. Um Algoritmo Genético (GA) foi desenvolvido para explorar o espaço de pesquisa do problema de alocação. O algoritmo tenta encontrar a solução ótima para o problema que tenha o menor custo, dadas as métricas recolhidas até ao momento. Embora a primeira otimização possa não resultar numa distribuição ótima dado o número reduzido de métricas recolhidas, à medida que a "pipeline" corre, é continuamente reconfigurada de forma a seguir o esquema da melhor solução atual. Depois de retornar a alocação ótima de tarefas, se ocorrer alguma perturbação no CPPS (ex: uma nova tarefa é introduzida), o algoritmo

calcula a nova solução para o novo cenário.

Como principais cenários de validação, o DINASORE foi testado com uma “pipeline” que tenta resolver um problema de eficiência energética usando “Dual Annealing”. Diferentes testes foram executados com diferentes dispositivos e variantes da “pipeline”. Uma das principais métricas usada na análise foi o tempo de execução da “pipeline”, que resulta da soma do tempo execução das tarefas com o tempo de comunicação, permitindo a comparação de diferentes mapeamentos de tarefas.

Keywords: Sistemas Ciber-Físicos de Produção, Programação em Blocos, Alocação de Tarefas, Otimização de Recursos

Abstract

Different industrial areas face problems related to task allocation and resource optimization, such as transportation systems, smart grids and manufacturing. There are numerous algorithms to deal with these difficulties, but they are seldom applied to or integrated with distributed Cyber-Physical Production Systems (CPPSs). Currently, the tools and platforms used to implement CPPSs lack integration with smart mechanisms to manage the devices, not allowing the reaction to external perturbations or the system performance optimization.

The reconfiguration and reorganization of CPPSs is accompanied by new challenges, due to the increasing number of interconnected devices. The speed and swiftness of the development and deployment processes of CPPSs are essential to handle several disturbances that might occur, as it happens when introducing new procedures/requirements.

One of the main issues to handle during the development and deployment of CPPSs is the allocation of software tasks to available devices, while guaranteeing a fluid task execution throughout the CPPS. This issue is particularly relevant because an unsuitable match between tasks and devices slows down the performance of the whole system, consuming numerous computational resources (e.g., CPU, RAM). Furthermore, CPPSs are usually comprised of devices with low computational power, so a poor allocation of tasks can put the execution of the pipeline in jeopardy.

This dissertation picks up on DINASORE Pereira et al. (2020), a tool that allows the deployment of software pipelines across distributed CPPSs. A set of features were built, which made DINASORE a tool that optimizes task allocations across a distributed CPPS, minimizing the devices' consumption of resources. When it comes to the adopted methodology, the program monitors the execution of function blocks in each device and collects relevant metrics such as the execution time, the networking time and the level of resource consumption. A Genetic Algorithm (GA) was developed to explore the search space of the allocation problem. The algorithm tries to find the solution for the problem with the least cost, given the metrics it has collected thus far. While the first optimizations might not result in an optimal distribution given the low amount of metrics collected, as the pipeline runs it is continuously reconfigured to match the scheme of the current best solution. After obtaining the optimal task placement, if we have a perturbation in the CPPS (e.g., a new task is introduced), the algorithm computes a new solution for the new scenario.

As main validation scenarios, the DINASORE tool was tested with a software pipeline that tries to solve an energy efficiency problem using Dual Annealing. Different tests were carried out with a different number of devices and variants of the pipeline. One of the primary metrics used for the evaluation was the pipeline execution time, resulting from the execution time of function blocks plus networking execution time, allowing the comparison of different task placements.

Keywords: Cyber-Physical Production Systems, Function Block Programming, Task Allocation, Resource Optimization

Agradecimentos

Quero deixar uma palavra de agradecimento a todos os que tornaram a conclusão desta etapa possível e que, de uma maneira ou de outra, me ajudaram neste percurso.

Aos meus supervisores que estiveram lá para mim desde a criação da proposta até à entrega.

À minha família, por sempre me dar tudo o que preciso para ter sucesso nos meus desafios.

Aos meus amigos, capazes de fazer de momentos díficeis motivos de alegria.

E em geral, um agradecimento a todos, estranhos e conhecidos, que foram mais além para me ajudar de uma ou outra forma.

José Rafael Fidalgo Fonseca Matias

"I have not failed,
but found 1000 ways to not make a light bulb"

Thomas Edison

Conteúdo

1 Introduction **1**
- 1.1 Context . . . 1
- 1.2 Motivation . . . 2
- 1.3 Objectives . . . 2
- 1.4 Document Structure . . . 3

2 State of the Art **4**
- 2.1 Background . . . 4
- 2.2 Related Work . . . 5
  - 2.2.1 Solutions . . . 6
  - 2.2.2 Similar Approaches . . . 7
  - 2.2.3 Summary . . . 8

3 Developed solution **9**
- 3.1 Technologies . . . 9
  - 3.1.1 Python . . . 9
  - 3.1.2 Eclipse 4diac . . . 9
  - 3.1.3 Eclipse Mosquitto . . . 10
- 3.2 Solution Architecture . . . 10

4 Implementation **13**
- 4.1 Resource synchronization . . . 13
- 4.2 Transparency and distribution . . . 14
- 4.3 OPC-UA data model and methods . . . 15
- 4.4 Monitoring component . . . 16
- 4.5 Optimal task allocation . . . 18
  - 4.5.1 Genome structure . . . 18
  - 4.5.2 Fitness function . . . 18
  - 4.5.3 Crossover and mutation functions . . . 19
- 4.6 Dynamic reconfiguration . . . 20

5 Experiments and results **22**
- 5.1 Single device scenario . . . 23
- 5.2 Distributed scenario with two devices . . . 23
- 5.3 Distributed scenario with four devices . . . 25
- 5.4 Distributed scenario in pipeline with branching . . . 25
- 5.5 Summary . . . 26

6 Conclusions and future work **29**
6.1 Contributions 29
6.2 Future work 30

Referências **31**

Lista de Figuras

2.1 Function block structure Eclipse 4diac Documentation 4
2.2 Function block 5

3.1 4diac IDE 10
3.2 High-level architecture diagram 11

4.1 Service Interface Blocks 14
4.2 "connections.json"file 15
4.3 Wrapper function block 16
4.4 Method caller function block 17

5.1 Basic optimization pipeline 22
5.2 Pipeline used in the first scenario 23
5.3 Initial mapping in the second scenario 24
5.4 Initial mapping in the fourth scenario 25
5.5 Execution times 26
5.6 Resource consumption 27

Lista de Tabelas

2.1 Related work categorization 6

5.1 Evolution in scenario two 24
5.2 Evolution in disadvantageous scenario two 25

Abreviaturas e Símbolos

AI	Artificial Intelligence
CPPS	Cyber-Physical Production System
DINASORE	Dynamic Intelligent Architecture for Software and Modular Reconfiguration
DNN	Deep Neural Network
DT	Digital Twin
EA	Evolutionary Algorithm
FB	Function Block
GA	Genetic Algorithm
IOT	Internet-of-Things
IIOT	Industrial Internet-of-Things
IT	Information Technology
MQTT	MQ Telemetry Transport
OPC-UA	OPC Unified Architecture
RI	Reinforcement Learning
SIB	Service Interface Block
WWW	*World Wide Web*

Capítulo 1

Introduction

1.1 Context

Current technological developments have resulted in what has been defined as a fourth industrial revolution, commonly referred to as "Industry 4.0". This setting brings forth a new approach to manufacturing, where smart factories will take the place of current regular ones. The machines present in these factories are equipped with a multitude of sensors, are capable of communicating with each other and run artificial intelligence (AI) processes which give them a high degree of autonomy. These AI processes allow for predictive maintenance, automatic fault detection, new business models, among other key advantages.

Another core component of Industry 4.0 are Cyber-Physical Production Systems (CPPSs). These systems manage different Internet-Of-Things (IoT) devices, the information they exchange and the processes they are running. From processes to the devices themselves, all of these components are digitalized in a CPPS. CPPSs are also decentralized and must be self-organized, capable of reconfiguring themselves dynamically.

It is in this context that the DINASORE tool was developed. This software allows the deployment and distribution of pipelines of function blocks across devices connected in a network. The behavior of the devices and their processes are also closely monitored in this tool. As such, using the DINASORE tool, developers can deploy and configure their software pipelines in Industrial Internet-Of-Things (IIOT) devices.

The DINASORE tool was written in Python and an instance of this program is to be running on each device of the target system. Developers can create code blocks according to the IEC61499 standard and use 4DIAC-IDE to assemble software pipelines. By deploying the pipeline to the devices via the IDE, the DINASORE tool will create an OPC-UA model of the code blocks it received on each device. Using an OPC-UA client, users can clearly monitor not only the values of the inputs and outputs of the code blocks but also different hardware resources of the devices such as memory and CPU utilization.

1.2 Motivation

According to H. et al. (2014), smart manufacturing systems are decentralized and capable of self-organization. As a result, it is a requirement for CPPSs' tools to be distributed and transparent. Through distribution, these systems execute complex processes in different devices, resulting in task parallelism and increased efficiency. At the same time, transparency means that said devices are capable of freely communicating with each other, without requiring previously set up communication channels between devices that are expected to exchange information between them.

In addition, the capability to perform dynamic allocation of resources to tasks comes with many advantages to the manufacturing factories. A smart allocation of resources to tasks results in lower resource consumption, which in trade leads to an improvement in manufacturing efficiency and a reduction of production costs et al. (2018).

As it is described in Chapter 2, current solutions lack the features described in the previous paragraphs. These tools do not react to external perturbances (physical addition or removal of devices from the system), and as such cannot be considered self-organized. Furthermore, these systems rarely focus on resource optimization across a network of heterogeneous devices, such is the scenario of this thesis. However, it must be noted that there are some solutions outside the field of CPPSs – mainly in the field of video-processing software - that implement some of the described features Patel e Tiwari (2020); Romero et al. (2021), which served as inspiration for the implementation of this work.

1.3 Objectives

The objectives of this work are to improve DINASORE, transforming it into a CPPS tool that allows the deployment of software pipelines across heterogeneous devices connected in a network. It should enable the CPPS to configure itself both to reach an optimal state in terms of pipeline execution time and resource consumption, as well as to deal with external disturbances to the system. The objectives are fourfold and are listed below.

Enabling auto-configuration of software pipeline across devices. Developers will be able to develop their own software pipelines, which are made up of function blocks. Each of these blocks contains a set of inputs and outputs. Currently, for a pipeline to run in multiple devices using the DINASORE tool, users must make use of what are called "Server Interface Function Blocks". These blocks preemptively specify the destination of each block's outputs, making it impossible for a reconfiguration of the system. As such, the first objective will consist in dropping the dependency of these function blocks for communication of outputs and inputs. Once accomplished, the DINASORE tool will be capable of dynamically allocating the tasks – code blocks – to the devices it sees fit, achieving transparency.

Minimizing pipeline execution time. One of the main goals of this work will be to minimize the execution time of a given software pipeline across a given CPPS. The system will profile each task, drawing conclusions from the computational resources required for the code block. Given

the heterogeneous devices it has at hand and once transparency is achieved, it will assign the tasks to the ideal devices such that the execution time of the pipeline is minimized. The execution time of a software pipeline is made up of the sum of each code block's execution time and the network execution time.

Optimizing devices' consumption of resources. Often, users might wish to specify time restrictions for the execution of a given pipeline. Once the criteria are satisfied, the system should find the optimal configuration such that the minimal amount of resource utilization takes place.

Handling system perturbations. A key aspect of the system is that it should be prepared for failure or addition of devices. Naturally, when a new device is added to the CPPS, it is expected that the execution time either stays the same or diminishes. On the other hand, if a device is removed or fails, the execution time should remain unchanged or increase. In the latter scenario, the system should be able to handle situations where the device that is the broker for communication crashes.

1.4 Document Structure

The next chapter goes over the state of the art, with background and related work. Chapter 3 covers the solution architecture and technologies while chapter 4 covers the implementation of the most relevant features. Lastly, chapter 5 covers the experiments conducted and yielded results and chapter 6 presents the conclusions of this work.

Capítulo 2

State of the Art

2.1 Background

This section will cover the basic underlying topics of this work. What follows is a theoretical explanation of the Information Technology (IT) related concepts that comprise the research context of this work, which are function block programming and both the IEC 61499 and OPC-UA standards.

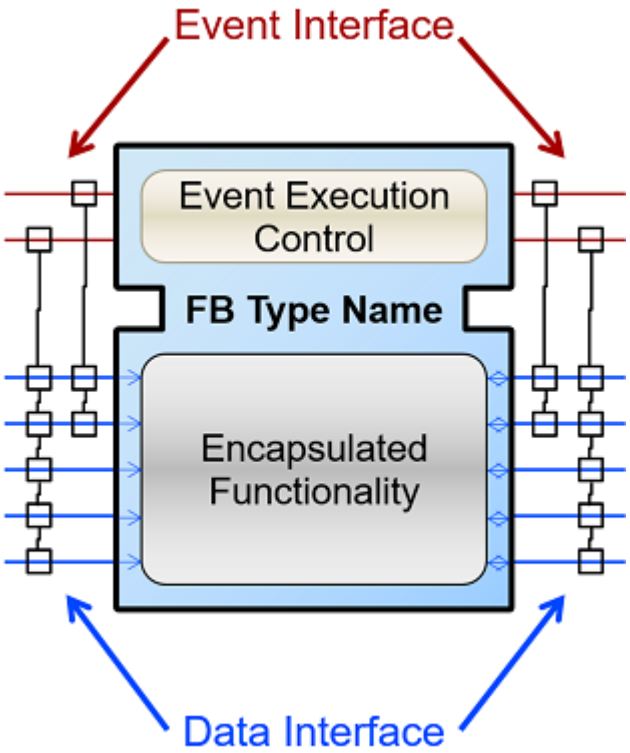


Figura 2.1: Function block structure Eclipse 4diac Documentation

Function block programming and IEC 61499. Function block programming is the practice

of writing blocks of code that will work in a block-diagram type of way. A given code block has input and output variables and an execution process that is specified by the developer. As such, a code block will execute its previously defined functions using the input variables and transmits the outputs to the next code block in a software pipeline. The IEC 61499 standard for distributed automation systems Vyatkin (2011) has function block programming as one of its main design components. In the context of this standard, there are no shared variables, meaning each block is completely independent from each other. Furthermore, the function blocks are event driven, meaning they have event variables as inputs and outputs. Only when these events are triggered, do the function blocks execute, otherwise they remain inactive.

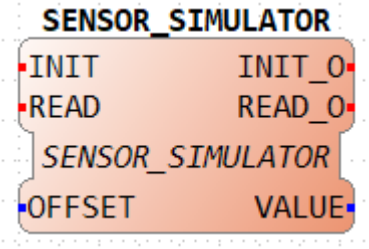


Figura 2.2: Function block

This further contributes to the independence of function blocks from the rest that make up a pipeline. These features make code reusability one of the strengths of this approach to software development since developers do not need to worry about hidden dependencies when using the same piece of code in a different context. Figure 2.1 shows the general structure of function blocks, while Figure 2.2 shows an example of a function block.

OPC-UA. The Unified Architecture of the Open Platform Communications foundation (OPC-UA) is a standard that defines a software architecture for industrial applications. It makes it possible for all sorts of information to be shared in a single unified address Leitner e Mahnke (2006). In the context of this work, the DINASORE tool follows this standard by uploading – among other data – information pertaining to the consumption of resources to an OPC-UA server. This data can be accessed by querying the server but it can also be explicitly visualized with the help of an OPC-UA client application.

2.2 Related Work

In this section different research works that attempted to come up with a solution for a problem similar to the one present in this thesis are analyzed. As the reader will be able to notice and as has been explained in previous sections, there is no software tool that contains all the features that are proposed to be developed in this work. Nevertheless, different works were found that are relevant for the proposed solution.

The challenge of allocating the proper tasks to the most effective resources goes beyond the context of CPPSs. Multiple solutions could be found in the field of video analysis, where software

pipelines are used to process and execute certain actions – such as object detection Romero et al. (2021); Patel e Tiwari (2020) – on high amounts of video data. Some of these works also address the concern of resource failures, but never address the issue of dynamic pipeline reconfiguration in the case of addition of new resources to the system.

Below is a table that categorizes the most relevant works that were found during the research stage of this thesis. Further subsections analyze these works by indicating their most relevant characteristics to the present developed solution.

Name	**Heterogeneous Devices**	**Resource Partitioning**	**Space search algorithm**	**System Perturbations**
IIOT et al. (2018)	Yes	No	Contract-Net Protocol	No
LLAMA Romero et al. (2021)	Isolated resources	No	Per-operation Invocation	Failures only
TAPP Mao et al. (2021)	Isolated resources	No	Deep Reinforcement Learning	No
CAAI Strohschein et al. (2021)	Different Kubernetes containers	No	Cognitive Services	No
CLITE Patel e Tiwari (2020)	No	Yes	Bayesian Optimization	No
KIT Jahn et al. (2015)	No	Yes	Dynamic Programming	Failures only
Developed Solution	Yes	No	Genetic Algorithm	No

Tabela 2.1: Related work categorization

2.2.1 Solutions

In et al. (2018) we observe what is a proposal for a configuration of a CPPS that implements dynamic management of resources. It is important to note that this work does not include any implementation, but rather offers the guidelines on which technologies and algorithms could be used to implement the mentioned features. It advises developers to use the OPC-UA standard as the means to publish information pertaining to the status of each device connected in the system. Furthermore, it appeals to the usage of a Software Defined Industrial Network (SDIN) for better network customization and control and Device-to-device (D2D) communication between system devices. As to the chosen algorithm to use in order to find the optimal configuration of tasks and resources, it appeals to the usage of the Contract-Net Protocol (CNP) to decide on which devices should perform which tasks. Each device acts as an agent bidding for the execution of a task, and according to the evaluation strategy – e.g., opting for low energy consumption – the master node in the protocol selects the winning device. The work also mentions the need of defining and using

an ontology model with which to feed the Jena tool, so the operating mode avoids overloading certain nodes and a system of priorities is used.

LLAMA presents itself as an already implemented tool that aims to improve the execution time of pipelines dedicated to video analysis. In order to reduce the amount of time spent searching for the ideal configuration for a certain pipeline, it decides which device to assign to a task only once the task is about to be called. This means that the search space is not that of an entire pipeline but of a single code block instead. Before the pipeline is executed, a one-time profiling operation is carried out on each block to determine its computational requirements. These requirements do not change depending on the rest of the pipeline, and so they are used to determine the device that is going to execute the invoked task. Additionally, the tool is used in a context where a latency target for the execution of a pipeline is given. To ensure the target is met, it calculates the latency target for each of the operations that comprise the pipeline. If a given task fails for whatever reason, it will recalculate the latency target for said operation. This new target will be smaller, and as such, another device might be chosen for the execution of the block. This strategy is what helps LLAMA deal with possible device crashes. Lastly, it should be noted that LLAMA was tested in clusters of serverless CPUs and GPUs, meaning there was no testing on devices that contain multiple computational resources.

TAPP is a tool that was developed to shorten the training time of Deep Neural Networks (DNNs). So as to achieve this goal, the tool starts by reading information pertaining to the inputs for the DNN model that is going to be trained. It creates a model based on this information and trains it with a single batch in a closed environment, following the basics of Reinforcement Learning (RI). Finally, the model is used to predict the training plan to be used by the DNN. The resulting training pipeline allocates tasks to the appropriate resources according to the plan, resulting in a lower training time.

Lastly, CAAI is presented as a tool that reduces the effort of implementation of AI algorithms in CPPSs. Its users can specify the goals of the task they wish to compute across the CPPS, such as resource optimization or predictive maintenance, among other possibilities. Then, using cognitive services, it selects an optimization algorithm to search for the optimal configuration. It can select a hill climbing algorithm, a trajectory type algorithm (e.g., simulated annealing), a population-based algorithm (e.g., particle swarm optimization) or an optimizer that uses a surrogate model. CAAI was tested in a network of several Kubernetes containers, where it was capable of automatically choosing an algorithm for pipeline optimization.

2.2.2 Similar Approaches

As is the case with CLITE, these solutions differ from the ones analyzed previously due to the fact that they do not operate on a set of heterogeneous devices. Instead, they were tested on specific given resources and the software attempts to partition the resources effectively and assign tasks to these partitions. CLITE, similarly to LLAMA, is used in a context where there are latency targets to be met. However, CLITE uses a Bayesian Optimization algorithm to find

the ideal configuration in the large search space of configurations. This allows it to meet the quality-of-service (QoS) requirements while maximizing the performance of the targeted jobs.

Finally, in Jahn et al. (2015) it is possible to observe an approach to maximizing the throughput of software pipelines that is twofold. Firstly, the concept of stage fusion is introduced, where consecutive blocks of a pipeline are executed as a single task. The computational requirements of the fused stage is the sum of the requirements of each fused block, but time spent in communication is saved. Secondly, the work uses an algorithm based on dynamic programming to recursively find the optimal configuration of tasks assigned to resource partitions.

2.2.3 Summary

In conclusion, the main challenge that these solutions - much like the work of this thesis - come across is how to efficiently explore the exponential search space of configurations. Several algorithms and approaches are presented, but it is important to stress that there is no implemented solution that allows the deployment of software pipelines across a CPPS, the dynamic reallocation of tasks to resources so as to optimize system performance and the handling of system perturbations upon addition or removal of resources.

Capítulo 3

Developed solution

3.1 Technologies

As mentioned previously, this work consists in the improvement of the DINASORE tool. This software uses different technologies for the application itself, the user interface and communication. The next subsections cover the technologies used for each environment.

3.1.1 Python

Python is a well-known programming language used by a great number of developers. The DINASORE tool can be ran on any version of Python3.6 or higher. It is language famous for its ease of development and package libraries. In reality, many tasks performed by devices in a CPPS may involve complex machine learning algorithms used for purposes such as predictive maintenance or optimizations. As such, by being written in Python, the DINASORE becomes a more useful tool to its users. There are also Python libraries for some of DINASORE's main components, such as communication across devices, running the OPC-UA server and monitoring the execution of the pipeline.

3.1.2 Eclipse 4diac

It is through the 4diac IDE that users can import and assemble their function blocks into a pipeline (see figure 3.1). Since a function block is made up of an "fbt" file for its structure and a Python file for its execution, both these files must be added to 4diac's folders in order to use them in a pipeline. When running the IDE, it is possible to drag and drop elements to a canvas and create the system's network diagram. This diagram is made up of the different devices in the system, as well as their addresses and ports for communication. The same drag-and-drop method is used to place function blocks in the canvas and connect them to each other. Furthermore, it is also possible to map the function blocks to the previously specified devices. Function blocks placed in 4diac, much like the ones provided by the IDE itself, abide by the IEC 61499 standard.

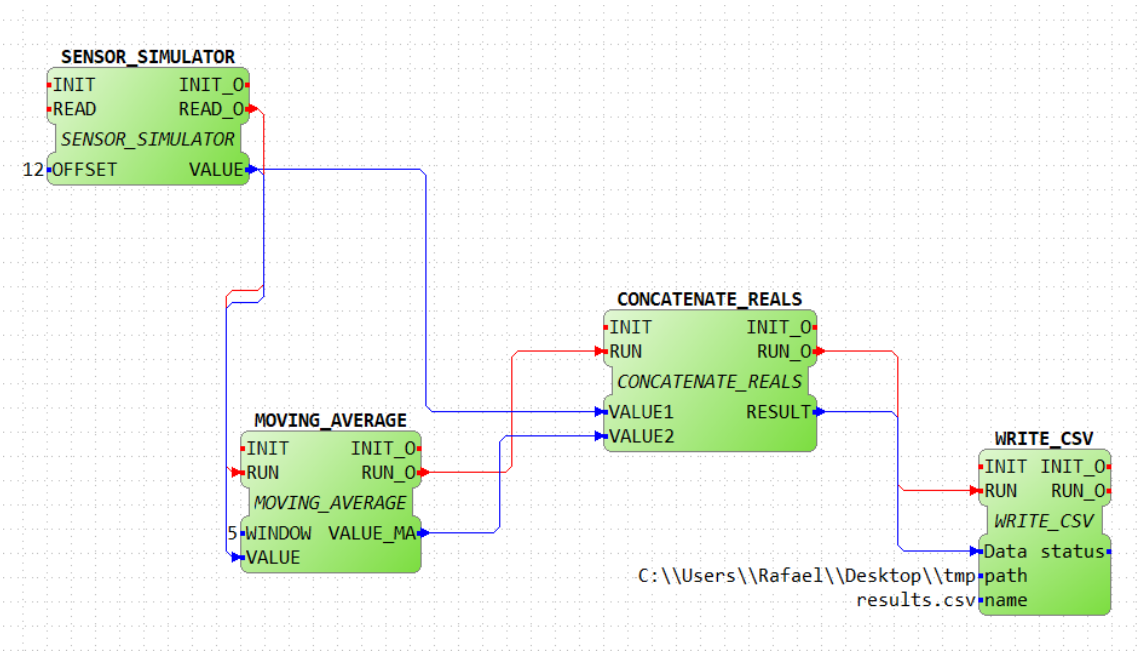


Figura 3.1: 4diac IDE

3.1.3 Eclipse Mosquitto

Mosquitto is a message broker that uses the MQTT protocol for communication. This protocol is lightweight, making Mosquitto an ideal solution for IIOT devices. Each device connected in the network used for the experiments (see 5) had Mosquitto installed and a broker running. This solution was required to enable the dynamic reconfiguration of the pipeline, where task assignments change and so do broker addresses.

3.2 Solution Architecture

First, it is important to clarify that the tool uses two main processes, which we will designate as the main process and the optimization process. The main process is running on each DINASORE instance in the network and is mainly responsible for orchestrating the execution of the pipeline. The optimization process only executes in the master machine, and results in the optimal distribution of tasks across machines.

Moving on to the behavior of the application and expected workflow, users start by developing their own function blocks that comply with the IEC61499 standard. All developed function blocks should reside in the folders of the DINASORE program. With this in mind, a repository was created with a bank of function blocks available for use, as well as a synchronization script that enables developers to transfer locally developed function blocks to DINASORE folders located in devices that are present in the network. Developers have only to specify the location of the folders in a JSON configuration file and all function blocks will spawn in the appropriate machines.

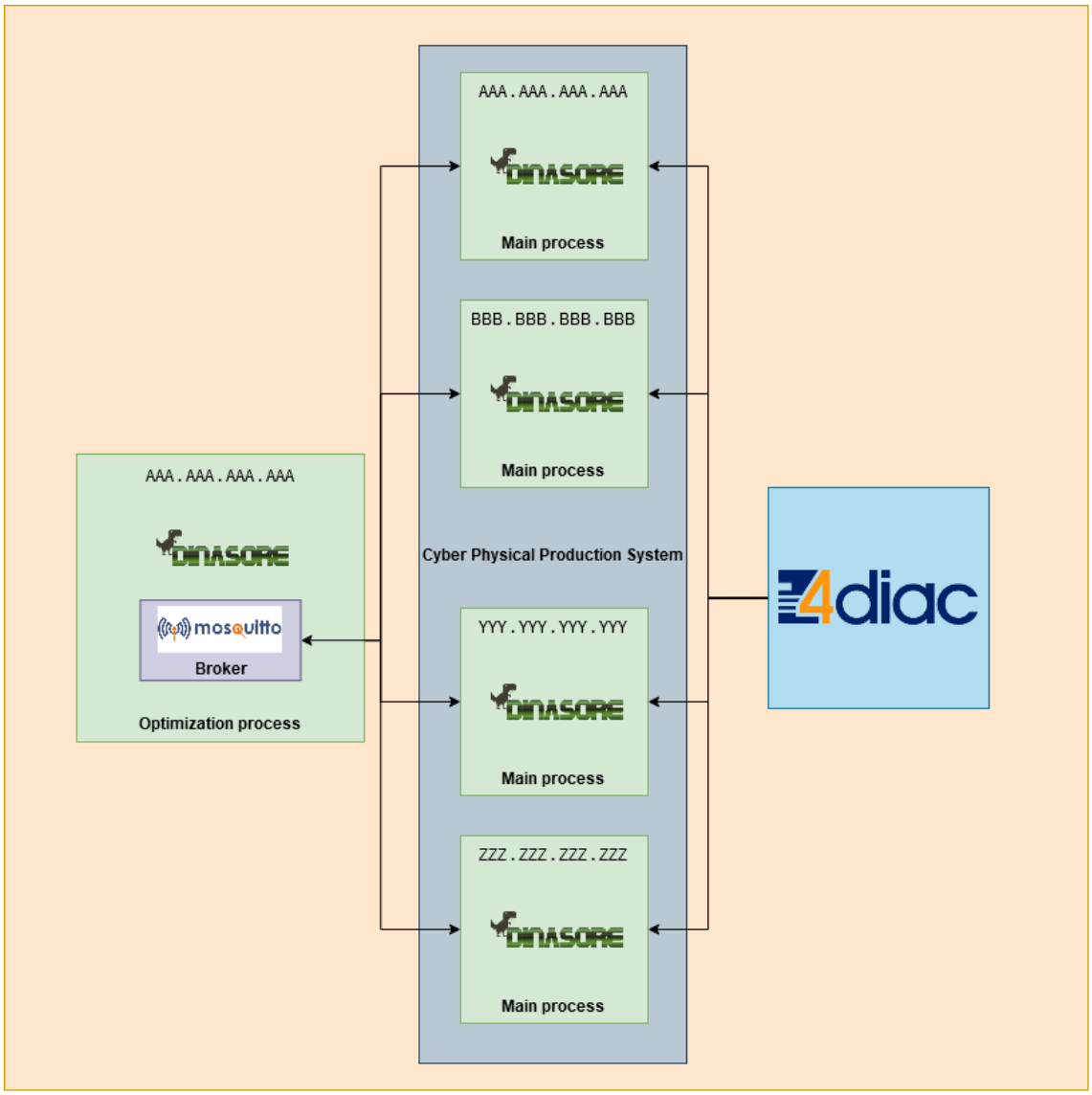


Figura 3.2: High-level architecture diagram

Once the function blocks have been developed and are made available at the target machines, it is time to assemble the pipeline that is to be distributed across the devices. As mentioned in 3.1.2, the 4diac IDE is used for this purpose. It allows users to connect the different function blocks to each other by events, while allowing the creation of connections between outputs and inputs of different function blocks. The IDE also allows users to create a scheme of the network and specify which devices are interconnected, as well as the ports that should be used for communication between the machines and the IDE.

Users can then deploy the pipeline across the system through 4diac. The IDE will send XML type messages to each machine that indicate the function blocks and connections required to be created. However, connections between function blocks assigned to different machines are not registered by the IDE. Consequently, before deploying the pipeline, users must ensure each DI-

NASORE folder contains a JSON file specifying these connections so that the DINASORE can create them without the instructions from 4diac. Lastly, the DINASORE folder must also contain one last JSON file that contains a list of all DINASORES in the network. This last file will be used to determine the master.

After the set up files have been created and properly placed, each DINASORE instance can be started. The main process runs on each instance, and starts by determining the master of the network from the possible choices available in the respective configuration file. Once the master is chosen, the decision is communicated across devices. Assuming no pipeline configuration files exist in the DINASOREs yet, users can deploy the pipeline on 4diac and each machine will receive the instructions on which function blocks and connections to create via XML type messages. These messages are stored on each DINASORE and shared with the master, who will merge them in a single file creating a representation of the pipeline and the initial task placement.

The pipeline starts immediately after the deployment, and with the execution of each function block, metrics pertaining to the execution time and resources used are collected and shared with the master. At the same time, the master DINASORE starts a new process – the optimization process. This separate program is the one collecting and merging the metrics from each DINASORE in a single file. Its purpose is to gather, collect and manage the metrics collected by each DINASORE (including itself) and also run the optimization algorithm fed with this data.

As more data is collected and shared across DINASOREs, running the optimization algorithm will yield progressively better results. Whenever an optimization is triggered, the process will use the collected data thus far to generate a new assortment of tasks and machines. Much like when the system is deployed using 4diac, after a change in configuration, each DINASORE needs to know which function block it needs to create and execute. As such, the master DINASORE will send XML type messages to each DINASORE with the information related to the new configuration. Each machine updates its set-up file which triggers the program to stop the current configuration and start running the new one.

Capítulo 4

Implementation

This chapter aims to describe the implementation of the main features of the DINASORE tool. Each description will explain in detail how it was possible to achieve the behavior described in 3.2. These features range from synchronization of files and transparent and distributed communication to the monitoring component, the optimization algorithm, and the process for dynamic reallocation of tasks.

4.1 Resource synchronization

This feature was developed in order to make DINASORE a more practical tool and remove some of the previously punishing complications. Since the tool is made to be running in different devices connected in a local network, users must ensure that each device contains the required project folder, as well as the folder with all required function blocks, not to mention pipeline-specific set-up files. Furthermore, these devices are often disconnected from the internet in an industrial context, and so it is impossible for users to download the folder with a simple command. This meant that the only other option users were left with was manually dragging and dropping folders from the working device to the target machines.

Thus, a synchronization algorithm was developed that allows users to specify the location of both the target machines and the folder with the function blocks resulting in the sharing of these function blocks across the machines. The machines can be local or have SSH communication available, through which the files will be synchronized. Nonexistent function blocks in the target machines will be added, and newer versions of the files will replace old ones.

Allowing users to swiftly update the DINASORE folders opened an opportunity for further increase in practicality. A feature was added where whenever a function block is started, a new sniffer thread is created that keeps check of the last time the file with the function block's code was last modified. Whenever this date changes, the function block is reloaded with its new content and will start executing the latest functions. This makes it so that users can make changes to the behavior of the function blocks in a pipeline without having to shut down the system. Changes can be made locally to the function block's Python file and then, with the help of the synchronization

method, each DINASORE will update its library of function blocks. Lastly, the sniffer thread will detect changes and will reload the function block in question.

4.2 Transparency and distribution

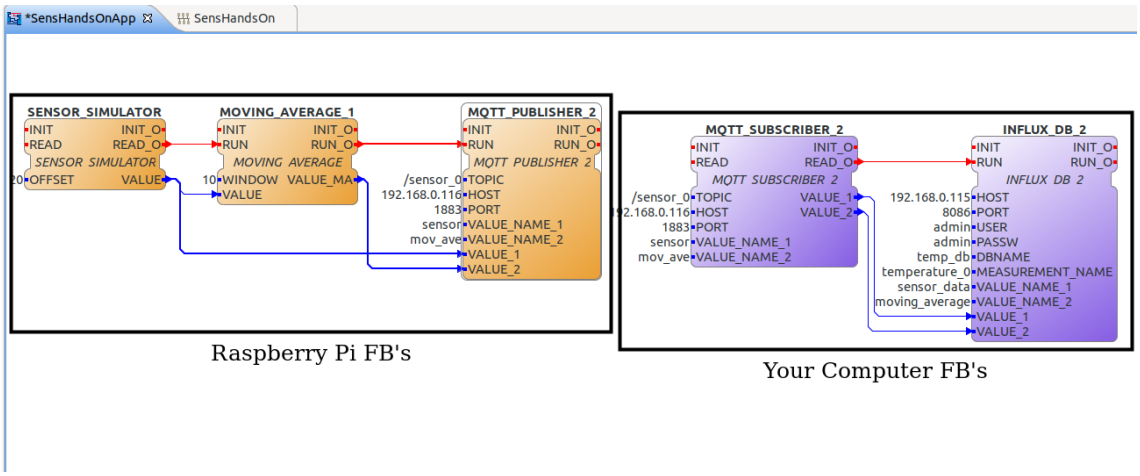


Figura 4.1: Service Interface Blocks

As mentioned in 3.2, the 4diac IDE allows users to specify the machines connected in the network so that XML type messages are sent to these destinations with information related to the function blocks and connections to be created. However, in order for such a feature to work properly, it is necessary to map each function block to the target machine. Connections between function blocks assigned to the same machine are processed by 4diac and the XML type message that tells the machine to create a connection between these function blocks will be sent to the machine they are assigned to. However, a connection between function blocks assigned to different machines leaves 4diac clueless about which machine should the message ordering the creation of the connection be sent to.

Previously, function blocks were developed to overcome this difficulty. These function blocks, called Service Interface Function Blocks (SIFB's), consisted in function blocks that published and subscribed to MQTT topics for the communication of inputs and/or outputs. Users had to specify different fields such as the port and address of the host, as well as the topic, and connect these function blocks to others in the pipeline. Not only was it cumbersome to set up and use, but it made dynamic reallocation (see 4.6) much harder.

Consequently, the DINASORE tool was updated so that certain actions are taken in case we have connections between function blocks hosted in different machines. First, the user must specify these connections in a JSON file. This file must also contain the groups of function blocks that must make up a cluster – that is, groups of function blocks connected to each other that cannot be assigned to different machines. The need for clusters might come from having to share values that are Python objects as inputs/ouputs of function blocks. An example of the contents of this file

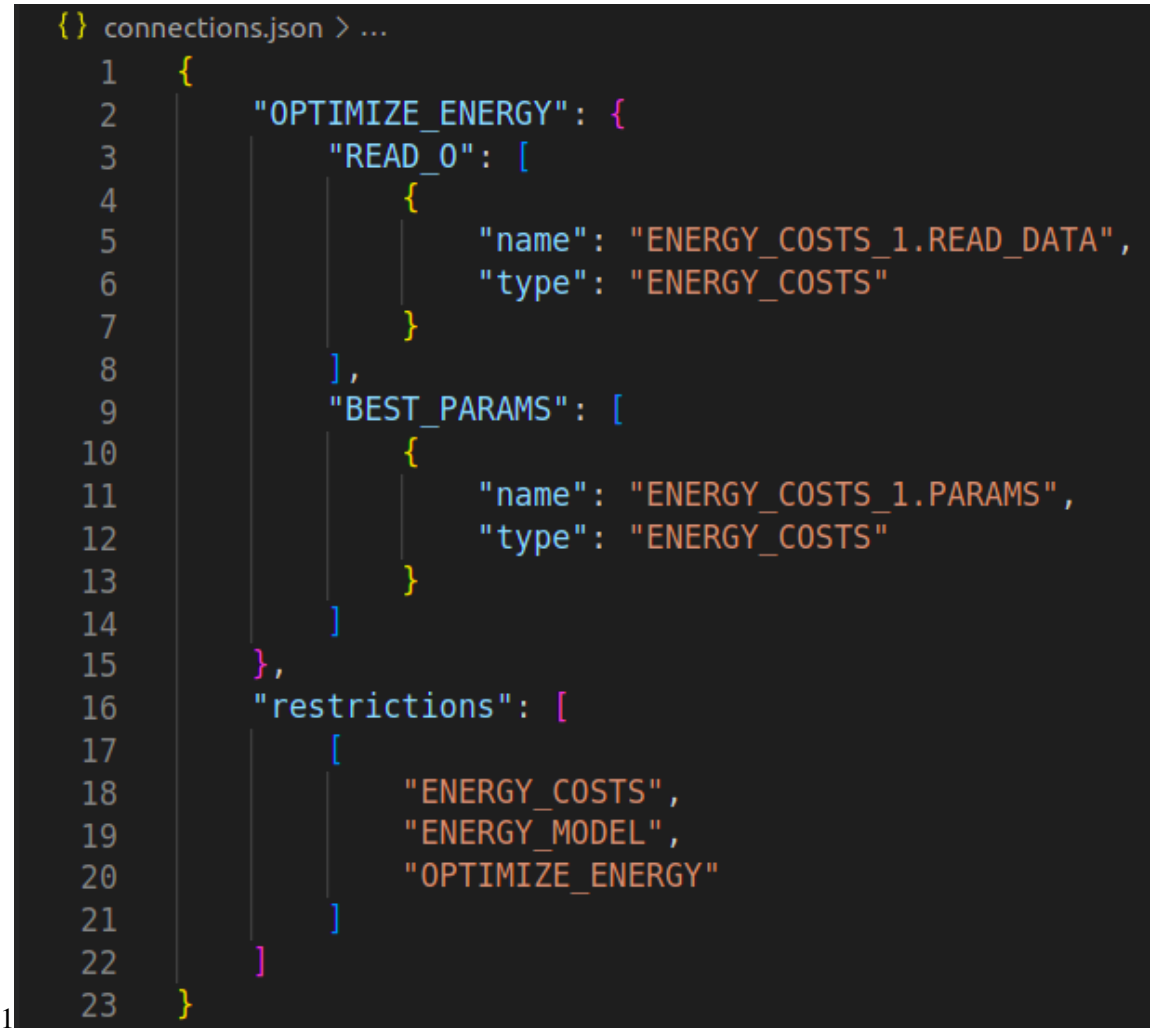


Figura 4.2: "connections.json"file

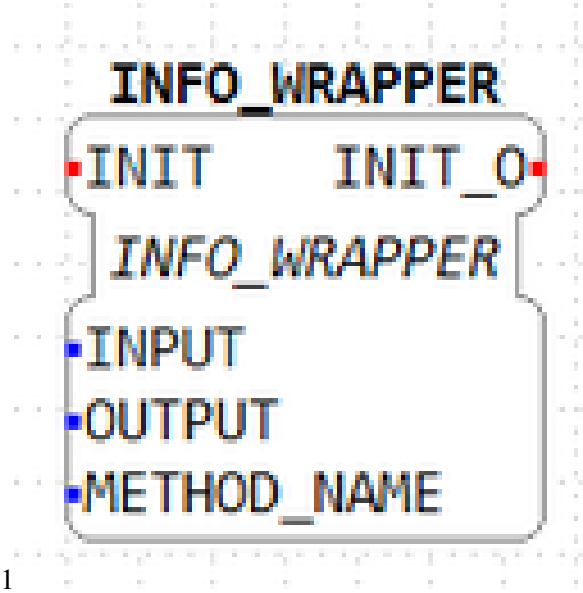
can be seen in 4.2. Much like other set-up files, the connections file can easily be placed in other machines by using the synchronization script described in the previous subsection.

Once the DINASORE analyzes the fboot file with the instructions on which connections and function blocks to create, it parses the connections file.

4.3 OPC-UA data model and methods

Section 2.1 explains how this standard is useful when sharing device related information in the network. The DINASORE tool starts an OPC-UA server that can be accessed by other devices in a single address. The information that the DINASORE uploads to the server follows a specific structure and contains metrics pertaining to system performance, pipeline execution, as well as callable methods.

When a pipeline is deployed and DINASORE puts it to work, a folder is created in the OPC-UA server with objects representing each function block. Additionally, these objects' properties are representations of the function block's attributes. Users can check their values and monitor the execution of the pipeline this way. The aforementioned OPC-UA methods consist in pipelines that can be triggered via the OPC-UA server. More specifically, a user can push an event in each function block. But if the function block is a part of a bigger pipeline, then users can call any given process or one of its subparts through the server.



11

Figura 4.3: Wrapper function block

This gave room for an opportunity to develop function blocks built specifically to create these callable methods. The first function block developed wraps a pipeline by connecting its events to the beginning and end events of the pipeline (see 4.3). Afterwards, it generates a callable method object in the server where users can define the input values and trigger the first function block in the pipeline. The result will be the outputs of the last function block wrapped by the pipeline. Secondly, a function block was developed to call OPC-UA methods (see 4.4). This way, users can create pipelines having a single function block representing a larger pipeline hosted in an OPC-UA server. This adds to the modularity of the pipelines, and makes it possible to infinitely wrap larger pipelines while keep the system balanced across devices and the pipeline design simple.

4.4 Monitoring component

The monitoring component is the set of features that are related to the gathering and analysis of metrics related to the performance of the pipeline. The performance is measured by the execution time of each function block, the delay in communication between devices and the resources used by each function block. Firstly, the data dependent on the execution of each function block is

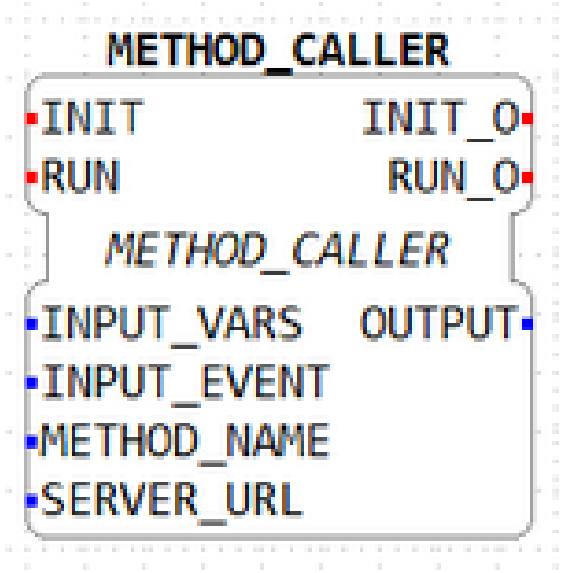


Figura 4.4: Method caller function block

gathered. For the execution time, a timestamp is created before and after the execution of the function, and the difference between these values makes up the function block's execution time. As for the resources, both CPU and RAM levels are also monitored during the execution of the function block.

Each DINASORE creates a JSON file to store these values. A thread is created when starting up the configuration whose sole responsibility is to access and store the file in a safe manner. Other threads can communicate with the one in question by a message queue, where data is passed so it can be saved. Much like the values related to the execution times and resource usage are shared on this message queue, so are the values related to the network delays. Whenever an output of a function block needs to be sent to a different machine, the message will contain the timestamp at which it was sent. The receiving DINASORE will once more store the difference between these values in its metrics file.

Although each DINASORE keeps its metrics file in case the master DINASORE crashes, these metrics are still shared with the master DINASORE who merges them in a single file. This file contains not only the metrics gathered by each machine, but also merged metrics which provide a list of all execution times and resource metrics related to a single function block. These lists are useful when we wish to quickly calculate averages independently of the machine. Furthermore, this file contains nearly all necessary information and data to feed the genetic algorithm. As such, many methods were developed to access the file and retrieve important values, lists of values or even averages.

Finally, as the merged metrics file changes, its different versions are stored under a graphical representation. If the user wishes to stop the optimization process, an animation is generated with all versions of the metrics file, and as such the user can easily observe the impact the algorithm had in the execution time and resource usage of the pipeline. These last features were mostly useful for

the tracking of the optimization algorithm but also for obtaining an understandable representation of the pipeline evolution, which was paramount when describing the experiences and results of the project.

4.5 Optimal task allocation

After analyzing current solutions to similar problems, it was decided that the DINASORE would use a Genetic Algorithm (GE) to find the optimal distribution of tasks to machines. Genetic Algorithms are Evolutionary Algorithms that attempt to apply the process of natural selection in determining continuously better solutions to the problem. Initial possible solutions are randomly generated to make up a population, and through crossing and mutating the solutions, new generations will rise with different task assignments. In a scenario such as our problem, where finding the optimal solution becomes exponentially more costly depending on the amount of function blocks, these algorithms are very useful to traverse the solution space effectively.

The algorithm starts by generating a group of genomes, which will make up the population. Then, for as many generations as the user specifies – or until the minimum solution value is reached – two parents are selected from the population. The population is sorted ascendingly by the fitness value of each genome – the value which tells us how good a solution is – and so the parents are merely the first two genomes of the population. These two genomes are added to the new population after going through a crossover process and being mutated. The process is repeated until the population has the initial size and once the generation limit has been reached, the algorithm presents the ideal solution.

4.5.1 Genome structure

A genome has the structure of a possible solution for the problem. Since our problem is about assigning tasks (function blocks) to machines, the structure of a genome is a list of tuples. The first element of the tuple is the name of a function block and the second the name of a machine.

Consequently, a chromosome is represented by each single tuple in the list. In order to generate a random genome, the algorithm needs only to assign each task to a random machine. However, the algorithm checks the list of pipeline restrictions – groups of function blocks that must be assigned to the same machine – so as to make correct and possible assignments only. Generating the population is as simple as generating as many genomes as the desired population size.

4.5.2 Fitness function

As mentioned in 4.5, the fitness function is the method that determines how valuable a solution is. In order to perform an evaluation of the pipeline, the algorithm starts by turning the pipeline into a graph. The optimization process does this by analyzing the general fboot file that contains all the XML type messages required to create function blocks and connections on each DINASORE. From these messages it creates a weighted directed graph where each event of a function block is

connected to another. It is important to mention that while this file is used to create the structure of the graph, the genome is consulted to figure out the assignments of tasks to machines and consequently the node and edge weights. If two events are assigned to different machines, the weight of the edge that connects them is zero. Otherwise, the weight is the average networking time is calculated from the file with the collected metrics thus far. The weight of a node is its average execution time given the task and machine it holds. If a metric is not found in the metrics file – if an assignment has never been attempted – its value will be zero.

Once the graph is constructed, the algorithm starts collecting the properties and values necessary for estimating the solution value. These properties were selected with the optimization problem in mind, which is to minimize the pipeline's execution time and resource consumption. As such, the first property that is retrieved from the graph is the largest possible cost (LPC) of the pipeline. A pipeline can have multiple paths and each path might result in a different execution time. The largest possible cost then becomes an important property that basically specifies the worst possible execution time for the entire pipeline. Usually, the highest cost path between two nodes is the one with the highest total of edges costs, but it is important to take the weight nodes into account too.

It is also important to ensure the minimum resource consumption across the devices in the system. For this matter, the difference between machine costs (DMC) is calculated. Each machine cost is the sum of the average percentual value of CPU and RAM consumption for each task assigned to it. Naturally, machines with more and heavier tasks will have a higher machine cost. The DMC is relevant to ensure a uniform placement of tasks across the machines, since it is the result of the difference between machine costs between each machine in the system. If no machine is overloaded, the DMC's value should be close to zero.

Lastly, we analyze the cluster difference (CD). A cluster is a set of function blocks connected to each other assigned to the same machine. Ideally, each machine should have a single cluster, meaning it is solely responsible for executing part of the pipeline while minimizing networking costs. The number of clusters in a genome is calculated with the help of the graph structure, and the CD is the sum of the absolute value of each difference between one and each machine's number of clusters.

Naturally, we end the fitness function by retrieving the overall average resource (CPU and RAM) consumption across the pipeline and the average networking time. Each of the collected properties has its own magnitude. In order to avoid incurring bias over a specific metric because of its order of magnitude, all properties are normalized before being added together. The result is the value – or actually the cost – of the solution. A value of zero represents an ideal distribution of tasks to machines.

4.5.3 Crossover and mutation functions

When generating the next generation, there are three main functions that play a role in changing the genomes. First, the population is sorted ascendingly by their fitness value, leaving the genomes with the least cost in the first elements. As such, the first two elements chosen for the

next generation – the parents – are the first two elements of the population. Then, in order to fill the rest of the new population, copies of the parents are successively subject to crossover and mutations before being added as genomes for the next generation.

The crossover step consists in joining the two parents, creating new offspring with the strongest genes. A random index is generated at which to split the genomes, and the chunks of each genome are joined to other other's chunks. Furthermore, there is a chance for these genomes to go through mutations. The first type of mutation is random resetting, where a random chromosome is picked, and its machine is changed to a random one. This allows exploring new solutions outside the current solution space being explored. The second type of mutation is swap mutation, where two chromosomes are picked, and their machines are swapped. This type of mutation enables the algorithm to explore new solutions that keep the distribution of the system balanced.

4.6 Dynamic reconfiguration

Once the new solution for the distribution problem has been found, it has to be put in practice. Naturally, this requires a mechanism that keeps each DINASORE instance updated of the tasks it needs to run. It makes sense to decide that in a network of connected devices, the same device running the optimization process – the master DINASORE – should also orchestrate the communication between these devices. Afterall, communication will be used to share the results of the solution across each device which has been produced by the optimization process.

As mentioned before, the instructions for each DINASORE configuration (that is, which function blocks and connections each machine must create and execute) are contained in the datamodel.fboot file. Each line in this file is an XML type message and its structure is the name as the one used by 4DIAC to deploy the pipeline across the devices in the network. These instructions range from creating function blocks of specific types, to assigning values to attributes and creating connections. It is possible to state that a pipeline can be represented by the set of all of its instructions, and each device should contain in its datamodel.fboot file the necessary instructions for the function blocks it is going to execute.

For each chromosome in the selected genome, the optimizer checks which function block is assigned to which machine. A dictionary is created where each machine has a corresponding list of instructions. For each function block a machine has, it will have a corresponding instruction to create said block. When it comes to connections, if a pre-defined value is to be assigned to a block's attribute, the corresponding instruction will be present in the machine's set of instructions. If a connection between attributes of different function blocks hosted in the same machine, the corresponding instruction will be added to the machine's set of instructions too. However, in the case of transparent connections (connections between two blocks hosted in different machines), the corresponding instruction can be added to no machine. Instead, it must be added to the connections.json file which contains information about not only the transparent connections but also the restrictions for function block clusters.

In order to have these changes take into effect, each DINASORE process must be in communication with the optimization process. Since it makes sense to have the optimization process act as an orchestrator of this communication, its host machine will start an MQTT broker in its address. This server will be used to exchange messages between the two processes. The messages vary from fboot instructions to collected execution metrics and they are exchanged under the appropriate topics.

The first topic is called “fboots” and it is under this topic that each DINASORE process sends messages to the optimizer. These messages follow a structure that calls for “type”, “content” and “origin” properties. If the type is “FBOOT”, then content is the list of lines the DINASORE received from the deployment of a pipeline. The optimizer will use this information to build the generaldm.fboot file, where it keeps a record of all the instructions in the pipeline. Should the message type be “METRICS”, then its content will be the content for the sender’s metrics file. This update on the file performed by the optimizer will guarantee that the next optimization will have the latest metrics to work with. Naturally, the “origin” attribute enables the optimizer to know the address of the sender.

Lastly, the “reconf” topic is used by the optimizer to share configuration instructions once it generates a new solution for the problem. Each message exchanged in this channel has a property for each device address. Its content is the list of instructions required for executing the latest configuration. There is one additional property whose content is meant to replace the content of the connections.json file. Each device connected in the network is always listening to this topic, and upon receiving a message, it replaces its previous configuration instructions with the new ones and starts executing its assigned tasks.

Capítulo 5

Experiments and results

As mentioned in previous sections, the DINASORE tool manages virtual representations of real devices, commonly used in a manufacturing context. This typically implies a connected network of low computational power devices, which is the target for the distribution of the pipeline. For that matter, the tests were carried out in a cluster of four Raspberry Pis connected to a switch, which in turn was connected to the local device from which the pipeline was deployed. However, the local computer was not running an instance of DINASORE. Much like in an industrial context, the machines (Raspberry Pis) were not connected to the internet, but the network connection allowed devices to access each other via SSH. This enabled the sharing of files when developing the DINASORE and the optimizer and the synchronization methods described in 4.1. Naturally, this also made it possible for devices to communicate with the chosen MQTT broker hosted in any of the devices.

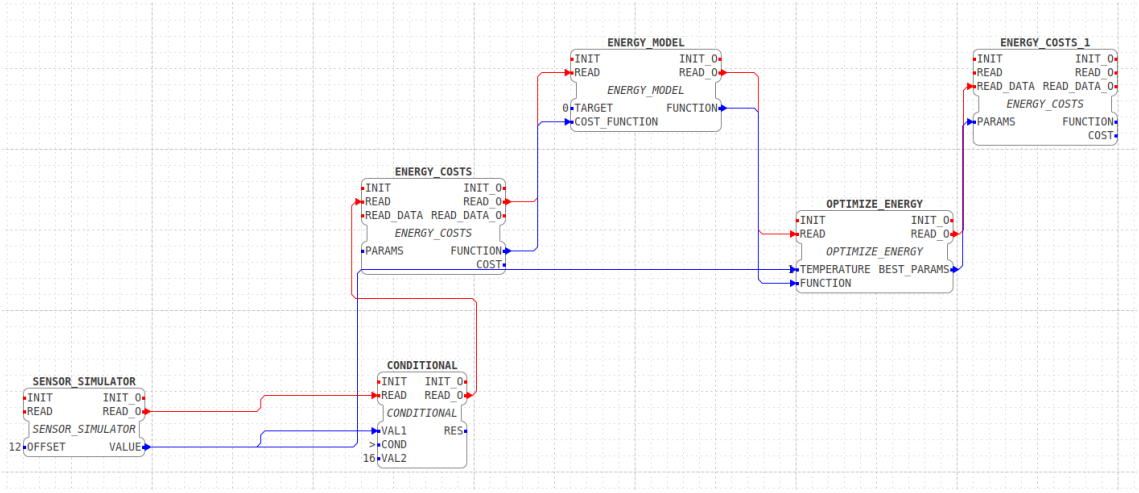


Figura 5.1: Basic optimization pipeline

The pipeline that was used across the tests emulates a temperature sensor in a room. When the temperature rises above a user-defined temperature level, the rest of the pipeline is triggered. The first function block in the pipeline (the starting node for the directed graph created by the

optimizer) is the “SENSOR_SIMULATOR”. This sensor generates values based in a normal distribution and constantly shares the output with the next function block, meaning it does not require any prior function block or manual input to trigger it. This function block feeds the “CONDITIONAL”, which determines whether the input received satisfies or not the condition specified by the user. In our scenario, we are stating that whenever the temperature rises above a given temperature, this function block should return true. Upon a passing condition, the energy related function blocks are triggered sequentially, resulting in a final cost. From all these function blocks, the “OPTIMIZE_ENERGY” – which performs a Dual Annealing algorithm – is the one that typically takes the longest to execute. In high powered devices this means a difference of a second or less, but in low powered devices this function block can take up to several minutes to execute.

5.1 Single device scenario

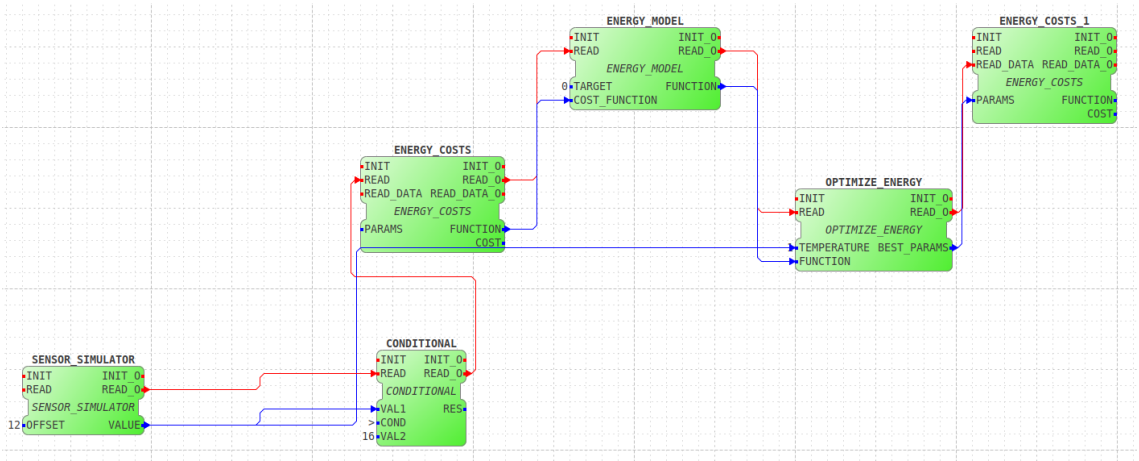


Figura 5.2: Pipeline used in the first scenario

The first scenario was comprised of the pipeline as shown previously, with all function blocks mapped to the one same machine. Naturally, this means that a single Pi will always run the same function blocks. The optimizer will always be resulting in the same solution, and its fitness fluctuates around a constant value given that rarely two metrics are the same. This simple scenario allowed a better understanding of the performance of the optimization algorithm as well as of the reconfiguration process.

5.2 Distributed scenario with two devices

The next experiment consisted in adding one Raspberry Pi to the CPS, as demonstrated in 5.3. It is possible to see how all function blocks are assigned to the first Pi except for the last one which is assigned to the newly added device. The dashed connection between the two function blocks shows the connection is remote. Once the optimizer collected enough metrics to run the

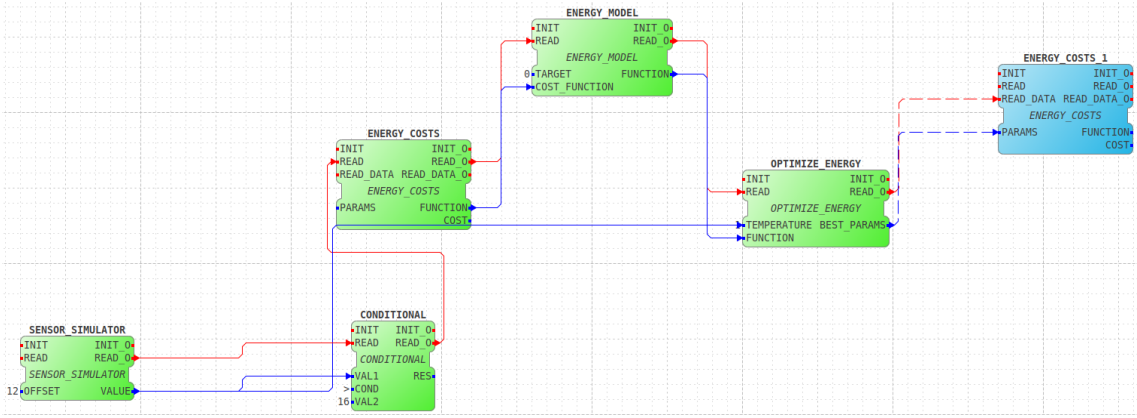


Figura 5.3: Initial mapping in the second scenario

EA, it completely flipped the mapping of each function block. This is the expected behavior since there are no metrics collected yet for these assignments and so the algorithm explores this possibility. At the third optimization, the algorithm returns a solution that is the quickest at running the pipeline and is evenly distributed across the devices. This usually means that each machine is assigned exactly one cluster of function blocks, minimizing the networking time and resource consumption. Should the network connection be poor or a device perform slowly, the algorithm will give preference to keeping all function blocks mapped to the same device.

1st generation

SS	COND	ECOSTS	EMODEL	OPT	ECOSTS1
Rasp1	Rasp1	Rasp1	Rasp1	Rasp1	Rasp2

2nd generation

SS	COND	ECOSTS	EMODEL	OPT	ECOSTS1
Rasp2	Rasp2	Rasp2	Rasp2	Rasp2	Rasp1

3rd generation

SS	COND	ECOSTS	EMODEL	OPT	ECOSTS1
Rasp1	Rasp2	Rasp2	Rasp2	Rasp2	Rasp2

Tabela 5.1: Evolution in scenario two

In 5.1 it is possible to better understand the evolution of the solutions generated by the optimizer. Each function block is represented by an acronym following the visual order in 5.2. One of the two Raspberry Pis is always under a task representing the task assignment.

One other experiment carried out in the same scenario consisted in hardcoding a disadvantage for the first device when executing the “ENERGY_MODEL” function block. The pipeline was deployed with all function blocks assigned to the first device. After the algorithm explored mapping the function blocks to the second device and realized executing the middle cluster of function blocks was quicker, it continued distributing optimal allocation which assigned the cluster to the second device. This evolution can be seen in 5.2.

1st generation

SS	COND	ECOSTS	EMODEL	OPT	ECOSTS1
Rasp1	Rasp1	Rasp1	Rasp1	Rasp1	Rasp1

2nd generation

SS	COND	ECOSTS	EMODEL	OPT	ECOSTS1
Rasp1	Rasp2	Rasp2	Rasp2	Rasp2	Rasp2

3rd generation

SS	COND	ECOSTS	EMODEL	OPT	ECOSTS1
Rasp1	Rasp2	Rasp2	Rasp2	Rasp2	Rasp2

Tabela 5.2: Evolution in disadvantageous scenario two

5.3 Distributed scenario with four devices

For the third scenario, two more devices were used making it a total of four Raspberry Pis. The results were similar, meaning that the algorithm started by making taking an exploratory approach. Naturally, since there were more devices, the search space increased meaning that more iterations are required to make a profiling of the devices. Given that it is a rather high number of devices for a small number of function blocks, the final solution might often trade an evenly distributed configuration over saving on networking time.

5.4 Distributed scenario in pipeline with branching

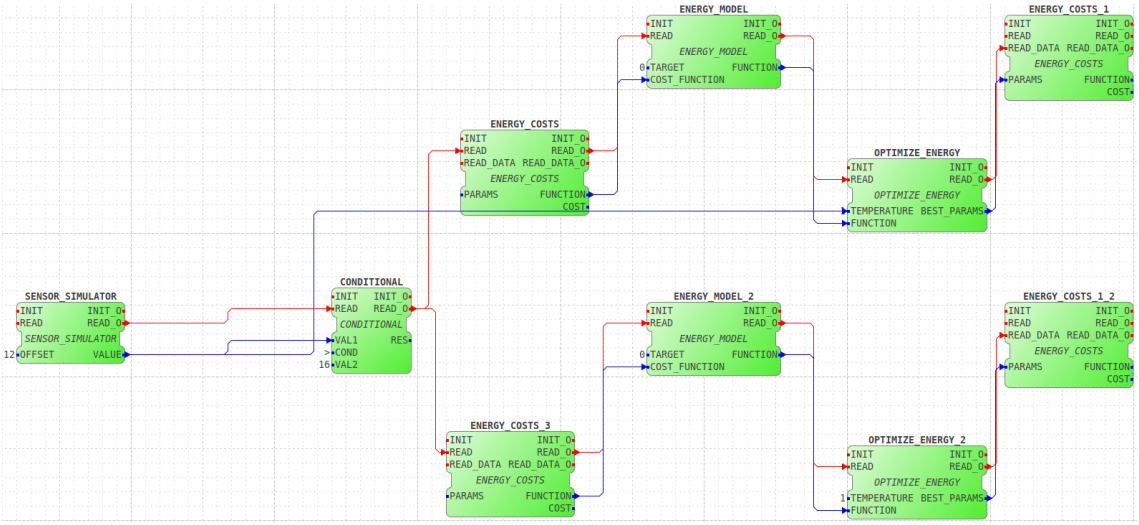


Figura 5.4: Initial mapping in the fourth scenario

The fourth scenario used a variant of the original pipeline, only this time the optimization process was duplicated, starting from the “CONDITIONAL” function block. This branching of the pipeline was particularly relevant since the optimizer excels at distributing workload across available devices. Given the low computational power of the devices, executing three or more energy

optimizations at once in a single device breaks the system. There is still a massive difference in execution time between executing one and two optimizations at once in a device, so the pipeline displayed in 5.4 was a viable use case. First, the pipeline was run with only one device in the system, to collect metrics relevant to running two energy optimizations at once. Afterwards, when the pipeline was run with all four Raspberry Pis in the system, the metrics used in the fitness function used by the optimizer dictated an even distribution of the pipeline across the devices. This resulted in assigning the optimizations to different devices and a faster and feasible execution.

5.5 Summary

As the optimizer ran in each test case, a copy of the file containing all the metrics pertaining to the pipeline execution was generated and stored. These files were stored for three optimizations in each test case and the most relevant data is the one we see in figures 5.5 and 5.6.

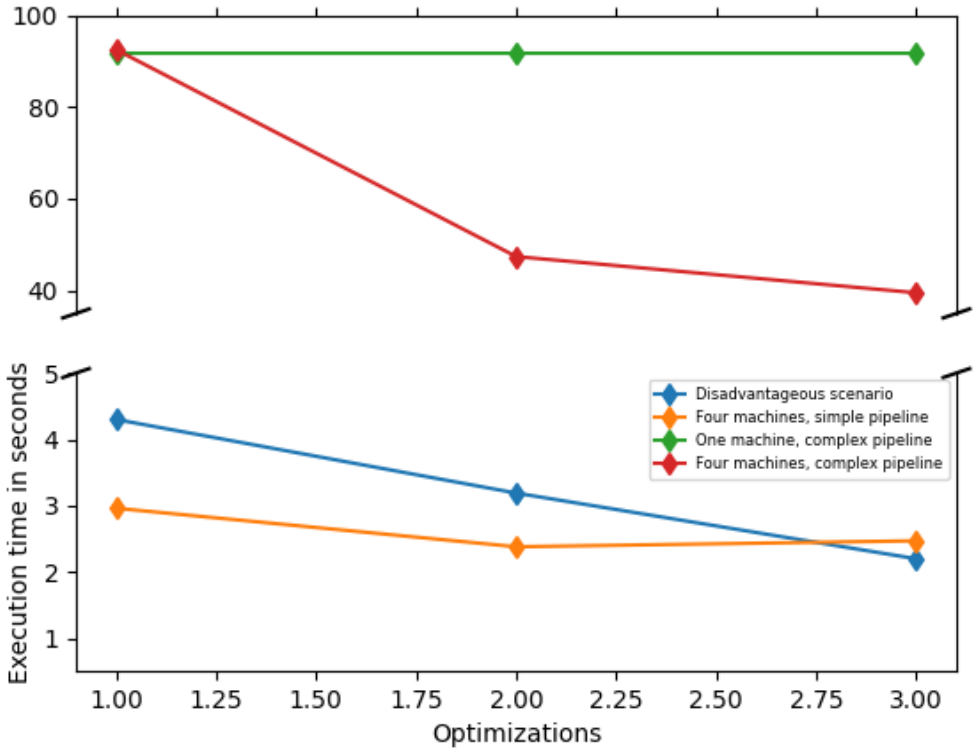


Figura 5.5: Execution times

In figure 5.5, the average execution time (obtained from adding execution times of function blocks and networking times in transparent connections) of the pipeline is represented for each optimization. In figure 5.6, each optimization has a corresponding percentual value for the average CPU and RAM across the pipeline. Bold lines were used for CPU values while dashed lines

were used for RAM values. Both graphics share the same color palette, where each test case is associated with a specific color.

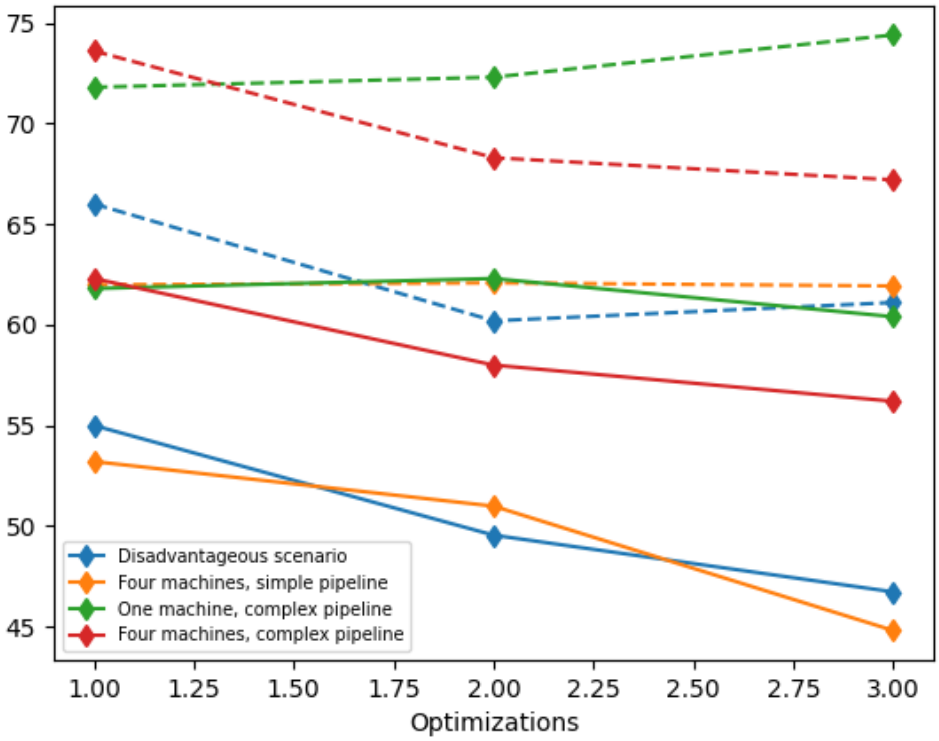


Figura 5.6: Resource consumption

It is possible to state that the DINASORE tool effectively searches for the best solutions given the parameters of the optimization problem. The optimizer will always try to return an improved solution each time it runs the algorithm, which is particularly obvious for the blue and red scenarios. For the disadvantageous scenario, the optimizer was clearly able to identify slow solutions and avoid them. For the scenario using four machines to run the complex pipeline, it is also clear that the DINASORE was able to distribute the workload of the pipeline across the available devices.

The scenario with a single machine shows the behavior of a pipeline that does not reconfigure upon execution results. It is possible to directly compare it to the red scenario, making it obvious the clear advantage of using this work's tool. A poorly mapped pipeline will constantly yield bad results, which does not happen when there is a smart CPPS managing tool capable of reconfiguring the pipeline according to its needs.

Lastly, it is also possible to state that a device will always yield different execution times and resource consumptions every time it runs a function block. This is not so clear for lighter function blocks, but heavier function blocks can somewhat fluctuate between values. As such,

some metrics are more reliable than others, and it shows the importance of collecting enough metrics before running an optimization.

Capítulo 6

Conclusions and future work

It is relevant to explain how this work contributed to the management of CPPSs through the DINASORE tool and what can be done next. The following sections summarize what has been achieved and its true impact and presents a set of system functionalities that could be developed next to further increase said impact.

6.1 Contributions

The first objective of this work as mentioned in 1.3 was to enable the dynamic configuration of the pipeline across the system. By creating a decentralized system where inputs and outputs are shared via MQTT, the DINASORE made it possible for function blocks to be hosted in different devices without requiring additional pipelines. Furthermore, these function blocks can be reassigned in real time to different machines and the pipeline will keep executing. This was all to enable to dynamic reconfiguration of the system upon generating new optimal allocation schemes.

By developing the optimizer, the remaining objectives concerning the minimization of execution time and resource consumption were also achieved. The fitness function of the optimizer takes into account different metrics that ensure the search for a solution that is evenly distributed across devices – which saves on resource consumption – while executing as fast as possible. While the DMC considers the resources (not) used up by each machine, the CD influences the number of tasks run in a machine. The remaining metrics obtained from analyzing the execution and networking times, as well as from the analysis of the pipeline as a graph, will directly impact the choice of the solution based on execution speed.

In an industrial context, a CPPS would run the configuration to which it was deployed until a new software pipeline is redeployed. As the number and variety of devices increase in a system, it becomes increasingly harder to map out the tasks each should run. Additionally, as software pipelines get more complex, it becomes urgent to ensure the pipeline is well distributed across de available resources. The DINASORE tool enables its users to be able to execute complex pipelines that would otherwise fail to execute given the low computational power of the devices.

The improvements made overcome many obstacles in the management of CPPSs and contribute to making this a valuable tool in the industrial context.

6.2 Future work

As it happens with almost any developed software, there are always functionalities that can be added that will improve the application. The DINASORE tool is no exception, and the improvements made in this work are a few from the set of relevant functionalities to be added.

Each configuration is specified in the "fboot" files which contain a set of instructions for which function blocks and connections to create. These instructions are initially shared by 4diac to the devices the function blocks are mapped to in the pipeline. However, when the DINASORE dynamically reconfigures the system, it manipulates these shared files making it unnecessary for the pipeline to be redeployed. As it has been done in previous works Pinheiro (2020), an application could be developed more closely knit to DINASORE that enables the sharing of function blocks but also of "fboot" files. Users could get different files from a marketplace and directly deploy it in the system.

Another interesting upgrade to the DINASORE tool would be to introduce speculative execution to the execution flow of the pipeline. With the goal of saving resources in mind, speculative execution could have function blocks executed prior to their scheduled time, saving resources for what might be needed most. This usually works through predictive execution where function outputs of a conditional branch are predicted, and the chosen path is followed until the real result is returned. This would further improve the resource management of the system.

Lastly, as the new improvements developed in this work are used for the first time by big amounts of users, there will naturally come the need to keep improving and upgrading the program. Nonetheless, the current improvements already yielded great results in utilization. The features that were developed proved to be complicated at times, having to orchestrate a distributed system and all its communication channels, but they all made it possible to enhance the performance of CPPSs through this work.

Referências

Eclipse 4diac Documentation. Function block interfaces. URL https://www.eclipse.org/4diac/documentation/html/before4DIAC/img/functionBlockInterfaces.png. [Online; accessed July 5, 2022].

J. Wan et al. Toward dynamic resources management for iot-based manufacturing. *IEEE Communications Magazine*, fevereiro 2018.

Lasi H., Fettke P., Kemper HG. e et al. Industry 4.0. junho 2014.

Janmartin Jahn, Santiago Pagani, Sebastian Kobbe, Jian-Jia Chen e Jörg Henkel. Runtime resource allocation for software pipelines. 2(1), 2015.

Stefan-Helmut Leitner e Wolfgang Mahnke. Opc ua–service-oriented architecture for industrial applications. *ABB Corporate Research Center*, 2006.

Yingchi Mao, Zijian Tu, Fagang Xi, Qingyong Wang e Shufang Xu. Tapp: Dnn training for task allocation through pipeline parallelism based on distributed deep reinforcement learning. *Applied Sciences*, 11(11), 2021.

T. Patel e D. Tiwari. Clite: Efficient and qos-aware co-location of multiple latency-critical jobs for warehouse scale computers. *2020 IEEE International Symposium on High Performance Computer Architecture (HPCA)*, 2020.

Eliseu Pereira, João Reis e Gil Gonçalves. Dinasore: A dynamic intelligent reconfiguration tool for cyber-physical production systems:. *Proceedings of the Third Conference on Object-Oriented Technologies and Systems*, 2020.

J.P.F.M Pinheiro. Internet of things software module marketplace, 2020.

Francisco Romero, Mark Zhao, Neeraja J. Yadwadkar e Christos Kozyrakis. *Llama: A Heterogeneous 'I&' Serverless Framework for Auto-Tuning Video Analytics Pipelines*, página 1–17. Association for Computing Machinery, 2021.

J. Strohschein, A. Fischbach, Bunte A. e et al. Cognitive capabilities for the caai in cyber-physical production systems. *Int J Adv Manuf Technol*, junho 2021.

V. Vyatkin. Iec 61499 as enabler of distributed and intelligent automation: State-of-the-art review. *IEEE Transactions on Industrial Informatics*, novembro 2011.