Novel Learning-Based Task Schedulers for Domain-Specific SoCs

by

Conrad Mestres Holt

A Thesis Presented in Partial Fulfillment
of the Requirement for the Degree
Master of Science

Approved July 2020 by the
Graduate Supervisory Committee:

Umit Ogras, Chair
Chaitali Chakrabarti
Ali Akoglu

ARIZONA STATE UNIVERSITY

August 2020

ABSTRACT

This Master's thesis includes the design, integration on-chip, and evaluation of a set of imitation learning (IL)-based scheduling policies: deep neural network (DNN) and decision tree (DT). We first developed IL-based scheduling policies for heterogeneous systems-on-chips (SoCs). Then, we tested these policies using a system-level domain-specific system-on-chip simulation framework [11]. Finally, we transformed them into efficient code using a cloud engine [1], and implemented on a user-space emulation framework [61] on a Unix-based SoC. IL is one area of machine learning (ML) and a useful method to train artificial intelligence (AI) models by imitating the decisions of an expert or Oracle that knows the optimal solution. This thesis's primary focus is to adapt an ML model to work on-chip and optimize the resource allocation for a set of domain-specific wireless and radar systems applications. Evaluation results with four streaming applications from wireless communications and radar domains show how the proposed IL-based scheduler approximates an offline Oracle expert with more than 97% accuracy and 1.20× faster execution time. The models have been implemented as an add-on, making it easy to port to other SoCs.

To my parents, Cristina and Perry, because without your love and support I wouldn't be where I am today.

ACKNOWLEDGEMENTS

The present Master's thesis is part of the Defense Advanced Research Projects Agency (DARPA) Domain-Focused Advanced Software Heterogeneous SoC (DASH) program. I was part of the intelligent scheduler team at eLab at Arizona State University lead by Dr. Umit Ogras, collaborating with the Center for Wireless Information Systems and Computational Architecture (WISCA). First, I would like to thank professor Dr.Umit Ogras and the department for giving me the opportunity to work in such a groundbreaking project. I want to thank Anish NK for sharing his work and supporting through all my way at the eLab. I also thank Nirmal Kumbhare and Joshua Mack from the University of Arizona, and Khyati Mardia from Arizona State University for getting involved and for the time spent helping me finish this thesis. Lastly, this work would not have been possible without the knowledge, patience, and help of all my peers at ASU.

This material is based on research sponsored by Air Force Research Laboratory (AFRL) and Defense Advanced Research Projects Agency (DARPA) under agreement number FA8650-18-2-7960. The U.S. Government is authorized to reproduce and distribute reprints for Governmental purposes notwithstanding any copyright notation thereon. The views and conclusion contained herein are those of the authors and should not be interpreted as necessarily representing the official policies or endorsements, either expressed or implied, of Air Force Research Laboratory (AFRL) and Defense Advanced Research Projects Agency (DARPA) or the U.S. Government.

TABLE OF CONTENTS

Page

LIST OF TABLES . . . vi

LIST OF FIGURES . . . vii

CHAPTER

1 INTRODUCTION . . . 1

1.1 Motivation . . . 4

1.1.1 Novelty . . . 4

1.1.2 Implementing IL Policies on Hardware . . . 4

1.2 Problem Statement . . . 4

1.3 Main Contributions . . . 5

1.4 Organization of the Thesis . . . 7

2 RELATED WORK . . . 8

3 TASK-SCHEDULING BACKGROUND . . . 11

3.1 Linux Process Fundamentals . . . 11

3.1.1 Linux Processes . . . 12

3.1.2 The Linux Scheduler: Data Structures . . . 13

3.2 A New Paradigm for Scheduling . . . 14

4 HEURISTIC SCHEDULING ALGORITHMS . . . 17

4.1 First-In First-Out (FIFO) . . . 17

4.2 Earliest Finish Time (EFT) . . . 18

4.3 Heterogeneous Earliest Finish Time (HEFT) . . . 19

4.4 Earliest Task First (ETF) . . . 21

4.5 Completely Fair Scheduler (CFS) . . . 22

5 DATA-DRIVEN SCHEDULING ALGORITHMS . . . 24

5.1 Genetic Algorithms . . . 25

5.2 Decision Tree Algorithms 26

5.3 Artificial Neural Networks 28

6 IL-BASED SCHEDULING FRAMEWORK 31

6.1 Building the Oracle 32

6.2 Describing the Architecture of the ML Model 34

6.3 Summary of Training Results 35

6.3.1 Training the Deep Neural Network 36

6.3.2 Training the Decision Tree 39

6.4 Converting the ML Model to a Low Abstraction Language 41

6.5 Extending the User-Space Emulation Framework (USEF) 42

6.6 The System: Zynq UltraScale+ MPSoC ZCU102 44

7 EXPERIMENTAL RESULTS 46

7.1 Domain-Specific Applications 46

7.2 Evaluation of the IL Policies on USEF 48

7.3 Comparison of Scheduling Policies 50

7.4 Algorithm Complexity Analysis 52

8 Conclusion and Direction for Future Works 54

REFERENCES 56

APPENDIX

A EXCLUDING PULSE DOPPLER FROM EVALUATIONS 65

LIST OF TABLES

Table Page

I Table of Static and Dynamic Features of the ML Models From Task T_i's Point of View 34

II Summary of the Features Inputs of the Proposed IL Policies 34

III Accuracy Comparison of the ML Models Based on Test Data 35

IV Execution Time Profiles of Domain-Specific Applications on Arm Cortex-A53 Core and FFT Accelerator [11] 47

V Characteristics of the Target Domain-Specific Applications and the Workload Used for Evaluation 49

VI Scheduling Overhead Cost and Memory Comparison of The Proposed IL Policies on Single Pulse Doppler Frame 49

LIST OF FIGURES

Figure Page

1 Scheduling Structures Used in the Real-Time Linux Kernel Scheduler . . 14

2 Example of Decision Trees Used for Classification 26

3 Perceptron Model Minski-Papert 1969 [71] 28

4 Example of Multiple Layers Stacked Forming a Neural Network 30

5 Overview of the Proposed Imitation Learning Scheduling Framework . . 31

6 (Left) Representation of the Fully-Connected DNN. (Right) Table Summarizing Architecture Insight 37

7 (Left) Plot of Model Accuracy Using the Training and Validation Datasets. (Right) Plot of Model Loss Function Using the Training and Validation Datasets 38

8 DNN Confusion Matrix Showing Accuracy and Error per Class 39

9 Weka Software GUI: Example of Training Data Used for the J48 Algorithm 40

10 DT Confusion Matrix Showing Accuracy and Error per Class 41

11 Extending the User-Space Emulation framework. The Proposed Approach has Five Worker Threads (WT), one Overlay Processor (OP) and Two FFT Accelerators 43

12 Overhead Study of the Different Scheduling Policies 50

13 Comparison of the Average Job Execution Time for Varying Throughput Across Different Scheduling Policies 52

14 Overhead Study of the Different Scheduling Policies Excluding Pulse Doppler Application. 66

15 Comparison of the Average Job Execution Time for Varying Throughput Across Different Scheduling Policies 67

Chapter 1

INTRODUCTION

The continuous demand for performance and efficiency keeps computing systems on constant improvement. Several years ago, distinct industry specialists observed a trend in many-core systems [69]. Nowadays, multi-core architectures have become more and more ubiquitous. This type of architecture benefits from Moore's Law [83] since speed-up on a system can be more significant than increasing the operating frequency of a single core. Matthew Reilly, Co-founder of SiCortex, Inc., mentioned that increasing the clock-speed has its limits [69]. The endless-growing demand for power efficiency and performance pushed the industry to explore combinations of multi-core architectures. Heterogeneous systems have several advantages over homogeneous counterparts. They offer a combination of low, mid, and high-performance processing elements (PE) that can be utilized based on the power and performance goals [54]. In contrast, homogeneous architectures use only one type of PE; hence they optimize a smaller selection of applications [40]. Homogeneous architectures do not benefit from domain knowledge, unlike heterogeneous architectures that can potentially be optimized for specific application domains.

However, there is still a gap in performance and efficiency between homogeneous systems and hardware-accelerated functions, such as computing a fast Fourier transform (FFT) operation on a dedicated accelerator. For example, when FFT and matrix multiplication tasks are computed on fixed-function accelerators, they achieve faster execution time and better energy-efficiency than when computed on general-purpose processors. Therefore, the acceleration of domain-specific tasks is becoming highly crucial. Some of these emerging domains are machine learning, autonomous driving,

and communication protocols.

We want to explore the benefits of a multi-core heterogeneous system with custom fixed-function accelerators by developing learning-based task scheduling techniques. The complexity of task scheduling is NP-complete, but some approaches can solve the problems in pseudo-polynomial time [33]. Optimization-based schedulers offer optimal solutions, but their complexity makes them infeasible. Therefore, most scheduling algorithms are tailored to a particular set of applications or domains. The proposed methods are evaluated using wireless communication and radar applications. We implement the proposed intelligent scheduling techniques on a recent user-space scheduling framework [61]. In our implementation, the process scheduling activity is completed inside the process manager. It inserts new tasks to the ready task queues, assigns the available tasks to PEs following our policies, and removes them after their execution finishes. Process scheduling plays an essential role in multi-threaded programming operating systems. This type of operating system enables loading more than one process into the executable memory at a time. Consequently, the loaded processes share the CPU and other PEs using time multiplexing.

Many runtime techniques, such as dynamic voltage and frequency scaling (DVFS), dynamic thermal and power management (DTPM), and power-saving modes (idle, sleep), have been proposed to lower the power consumption [98, 18, 74, 46, 25, 84, 19, 37, 88]. Readers interested in dynamic energy management can refer to a recent survey by Pasricha et al. [77]. The effectiveness of these techniques depends critically on how the tasks are scheduled to the PEs since computation and communication schedules determine the slack that can be exploited by dynamic power management techniques [42, 75, 31]. Hence, application mapping and scheduling have a significant impact on performance as well as in power [12, 43, 94]. Operating systems manage task scheduling in SoCs, i.e., they control the mapping of tasks to the various

processing elements. For example, servers have scheduling policies that allow many programs to run, which means that the CPU is continuously switching from one program to another. Emergency applications like nuclear reactors alarms, on the other hand, will probably give priority to the application of managing those alarms. The future of heterogeneous systems, a novel machine learning scheduler, and a combination of multiple scheduling policies that ideally switch dynamically depending on the system's current or future situation will lead to faster and more efficient performance.

This study implements the machine learning techniques of deep neural network and decision trees to address the following problems: task allocation and task acceleration. Artificial intelligence models can be used to schedule policies and predict the optimal schedule for incoming jobs. However, models need to be trained under different injection rates and application characteristics, like task dependencies, communication time, and estimated runtimes. AI models are integrated with the scheduling framework [5] to schedule tasks to resources. Full-system simulators, like Gem5 [20], perform detailed instruction-level simulations. Precise cycle-accurate simulations can lead to very high simulation times. Moreover, Gem5 offers details beyond the requirements of this study and is not suited for this particular work. Instruction-level simulations lead to significant execution times and make rapid design tedious and time-consuming. In this work, we use DS3 [11], a Python-based system-level simulator that supports runtime scheduling algorithm development, and rapid design space exploration. DS3 is instrumented to generate AI models for task scheduling. The policies developed with DS3 are then implemented in user-space emulation framework [61] to schedule tasks in streaming applications. All the policies are evaluated on a quad-core Xilinx Ultrascale+ ZCU102 field-programmable gate array (FPGA) with two custom FFT/IFFT fixed-function hardware accelerators.

1.1 Motivation

1.1.1 Novelty

There are several publications regarding job scheduling designed and tested on simulators, with approximated values, but little work is performed on real hardware systems. Process scheduling using imitation learning techniques [82] is a new concept [53].

1.1.2 Implementing IL Policies on Hardware

Scheduling a new process on a monolithic kernel [68] is extremely costly compared to micro-kernels or hybrid-kernels. Each time a new process is created in the application layer, a system call to the operating system needs to be done. Switching from user-space to kernel back and forth is known as context switching. Unfortunately, context switching is extremely costly. We implement a machine learning framework that schedules wireless and radar applications using imitation learning. The machine learning-based scheduling framework [53] is implemented as an extension of the user-space emulation framework [61]. The scheduler schedules tasks to general-purpose cores and custom accelerators using the concept of worker threads. The scheduling is managed in user-space and thereby, helps in reducing the context switches.

1.2 Problem Statement

The performance of highly optimized schedulers may be satisfactory under idealized conditions, such as theoretical analysis and high-level simulators. However, implementation overheads can diminish the potential benefits and make them impractical to use in real applications. In contrast, commonly used low-overhead heuristics, like first-in-first-out (FIFO) [49], round-robin (RR) [78] schedulers, minimum execu-

tion time (MET) [27] and earliest task first (ETF) [44], give up optimality in exchange for lower overhead and broader applicability. The gap between these two solutions can be filled for domain-specific systems-on-chip (DSSoC) by exploiting the domain knowledge [95]. More specifically, the applications that belong to a particular domain, such as wireless communications and radar, are represented by directed acyclic graphs (DAGs), whose nodes represent the tasks that compromise the domain applications. One can identify the most commonly used set of tasks or kernels in the target domain through an ontological inference process [97, 105]. Finally, this knowledge is used to determine the optimal composition of general-purpose and specialized processors (a.k.a., hardware accelerators), as well as the affinity between the tasks and PEs [11, 97]. We note that these steps are necessary but not sufficient to maximize the performance and energy-efficiency. The designers must also ensure that the schedulers use the available hardware resources optimally while running the domain applications.

This thesis seeks to answer the following question: *Can we achieve a performance similar to optimal scheduling algorithms for a set of domain-specific applications with negligible overhead on hardware platforms?* While answering this question, we also consider two practical constraints. First, the proposed methodologies should easily generalize to new applications, which are not known at design-time, and to new SoC configurations with new PEs. Second, the proposed methodologies should not place all the optimization burden on the application developers. The application developer should not be required to know the intrinsic details of the hardware [95].

1.3 Main Contributions

This work implements new scheduling policies using a recent IL-based scheduling technique [53]. It evaluates them on a quad-core Xilinx Ultrascale+ ZCU102 FPGA

using applications from wireless communications and radar domains. The FPGA integrates four Arm Cortex-A53 cores, two custom FFT/IFFT custom hardware accelerators, direct memory access (DMA) interfaces, and fourth-generation advanced extensible interface (AXI). The SoC runs on a Linux operating system with kernel version 4.9.0-Xilinx-v2017.2.

This work proposes a machine learning scheduling framework using a system-level simulator DS3 [11] to create an Oracle and train imitation learning models that perform similar to an expert, with minimal error. The proposed approach overcomes the limitations of the Linux scheduler, which is not optimized for domain-specific, real-time applications, and lacks accelerator-awareness. This work proposes to evaluate the IL scheduling policies by extending the user-space emulation framework [61] that provides accelerator-awareness and user-space scheduling support.

Furthermore, creating new processes in Linux is an expensive task as it is memory- and latency-intensive, and it does not scale adequately with the number of processes (Chapter 3 discusses the limitations of the Linux real-time scheduler).

The scheduling policies are integrated with the extended emulation scheduling framework, as detailed in Chapter 6. Moreover, we customized the kernel such that specific kernel functionalities are exposed in the user-space through system calls and boot configurations.

The primary contributions of this thesis are:

1. Efficient integration of a recent IL-based scheduler [53] into a runtime environment [61],

2. Extensive evaluation of the proposed approach with domain-specific applications, showing that the IL policies incur an error of only 2.7% with respect to Oracle, and

3. Techniques to minimize the scheduling overheads, resulting in a speedup of 1.2× in execution time with respect to the Oracle.

1.4 Organization of the Thesis

The organization of this thesis is as follows. Chapter 2 introduces the related work, with projects and recent publications that inspired this work.

Chapter 3 provides a brief background on task scheduling and introduces the Linux scheduler, the structures used for process scheduling, and the limitations of the Linux scheduler.

Chapter 4 and Chapter 5 highlight different heuristic and machine learning algorithms used for task scheduling. In these chapters, we explain the functionality of the algorithms.

Chapter 6 introduces our proposed approach of the imitation learning framework in detail. We describe all the elements that form the proposed scheduling framework and present a summary of the machine learning training results.

Chapter 7, we evaluate the performance of the imitation learning-based scheduling policies on a commercial SoC. We also give a summary of this thesis, and we discuss future work. Last, Appendix A contains all the evaluations from Chapter 7, excluding the pulse-Doppler application.

Chapter 2

RELATED WORK

The scheduling problem in domain-specific SoCs, which are heterogeneous multi-core systems, is NP-complete. Conventional optimization-based approaches, such as integer linear programming (ILP), mixed-integer programming (MIP), provide optimal solutions to the scheduling problem. However, the overheads involved in such approaches are very high [53] and limit its practical use in runtime systems. Heuristics are commonly used as alternatives for scheduling and widely deployed in commercial systems. For example, the completely fair scheduler (CFS) is the default scheduler in Linux-based operating systems [103, 9]. It was introduced in late 2007 and evolved from the O(1) scheduler. The primary focus of CFS is to maximize CPU fairness among the running processes. This heuristic scheduler uses red-black trees and sorts the processes by a fairness metric, enabling O(1) to fetch a task to be scheduled. Although the time complexity to retrieve a task from the tree structure is O(1), the complexity to insert a task into the tree is significantly high. Hence, there is significant scope in improving the scheduling overhead in the Linux based operating systems.

CFS was originally developed for homogeneous architectures i.e., all cores on the computing platform are identical. Several works have focused on improving various aspects of CFS, such as support for heterogeneity, minimizing scheduling overheads, improving energy-efficiency, and load balancing. With the advent of heterogeneous architectures to improve performance, power and energy-efficiency [35, 69, 41, 65, 90], the following research has addressed the integration of heterogeneity in CFS [17, 16, 32]. The authors in [73] introduce a new load-aware metric to improve the performance

of CFS. A sensing-based load balancing technique for Linux to improve the energy efficiency of heterogeneous MPSoCs is described in [87]. Although several additions have been proposed to CFS, the primary goal is to provide CPU fairness to the different executing processes. Another class of scheduling techniques, called list scheduling techniques, focuses on different objectives, e.g., performance, power, energy, and security.

List scheduling approaches [85, 55] organize the running tasks (processes) into a list. Scheduling at design time is a function of the position of the task in the list and a target optimization metric. A widely popular heuristic, heterogeneous earliest finish time (HEFT) [96] defines a metric called the upward rank. The upward rank is a function of the communication and computation times of the tasks in an application, which is modeled as a directed acyclic graph (DAG). This metric is then used to assign the processing elements (PE) to the different tasks, at design time. The techniques in [21] and [10] refine the approach proposed in [96] to improve the execution time of applications. The lookahead based approach in [21] suffers from fourth-order complexity on the number of tasks, whereas [10] incurs second-order complexity. Several other heuristics approaches have focused on energy [15, 92, 110, 22, 112], security [106] and fairness [107]. However, the drawbacks of heuristic techniques are: (1) tailored to specific objectives, (2) lack of generalization for multiple optimization objectives and simultaneous applications, and (3) high scheduling overheads for runtime applications.

Recently, machine learning (ML) has emerged as an efficient technique for resource management in multi-core SoCs [24, 65, 50, 77]. More specifically, DeepRM [66], DECIMA [67] and [57] focus on the use of ML techniques for the scheduling problem. DeepRM [66] and DECIMA [67] use Reinforcement Learning (RL) to schedule DAGs. While RL is highly suitable to enable lifelong learning, it suffers from major draw-

backs. First, the convergence of the RL-based scheduling technique critically depends on the reward function. The design of the reward function is very complex [50]. Second, the complexity of the scheduling problem results in a high number of episodes for convergence [53].

Imitation Learning (IL), a sequential decision-making strategy, has emerged as a powerful approach that overcomes the drawbacks of RL [89]. The IL-based approach involves the construction of an Oracle (expert) and generation of supervised learning techniques to mimic the Oracle. IL has been successfully used for runtime resource management, both offline [65, 88] and online [64]. Moreover, IL has been successfully applied to the runtime task scheduling problem in [53]. In this work, we develop a highly efficient framework that integrated an IL-based scheduler for implementation in a real heterogeneous SoC. The implementation of custom schedulers in a heterogeneous environment typically requires a simulation framework for Oracle generation and a runtime framework for scheduler implementation. The authors in [105] propose a simulation framework that integrates four other tools for scheduling, power, energy, and interconnect modeling. We use DS3 [11], which is an integrated solution and is well validated against commercial heterogeneous SoCs. StarPU [13] is a prominent runtime framework used for desktop and HPC design space exploration. For the implementation of the proposed IL-based scheduling approach, we choose [61] as it integrates a compilation and a runtime framework for effective scheduling of streaming DAGs in heterogeneous many-core SoCs. To the best of our knowledge, this is the first work that implements an IL-based scheduler in heterogeneous SoC (Xilinx Zynq Ultrascale+ ZCU102) comprising general-purpose cores and fixed-function accelerators. We evaluate both regression tree (RT) and deep neural network (DNN) based machine learning classifiers.

Chapter 3

TASK-SCHEDULING BACKGROUND

In essence, task-scheduling is the process of fetching and mapping tasks to resources satisfying target metrics, e.g., execution time, power efficiency, utilization. Operating systems (OS) like Windows, Mac, or Linux manage the scheduler of many computers, including embedded systems. In specific, the Linux scheduler manages processes like opening a browser, editing a file, accessing some media, that the system tries to execute based on a priority value, within a rigid period or time-slice. Despite being broadly used, it has some limitations [80] discussed in this chapter. For example, processes are executed exclusively on the CPU. However, the scheduler must also consider fixed-function accelerators as processing elements to exploit the heterogeneity of domain-specific architectures.

If the reader wishes to dive deeper into the Linux kernel, or needs clarification with process scheduling, system calls, and related topics. We suggest Wolgangs Mauerer's book - Professional Linux Kernel Architecture [70], the 3rd edition of Daniel P. Bovet, Marco Cesati's book - Understanding the Linux Kernel [26], or Robert Love's 3rd edition of the book - Linux Kernel Development [60] since they all give an initial understanding of the extensive Linux system.

3.1 Linux Process Fundamentals

Linux is an example of a monolithic kernel, a system architecture that allows the operating system to run exclusively outside the user-space (i.e., the core of the operating system) [81]. The OS is composed of many blocks, one of which is the scheduler that runs entirely inside the kernel. The scheduler uses an algorithm that

must fulfill several objectives, such as fast response time, selecting the process with the highest priority, and avoiding starvation. The scheduler assigns tasks to resources following a set of rules. The set of rules used to determine where a new process executes is called a *scheduling policy.*

In Linux, processes are considered either a standard process or light-weight process (LWPs) [60]. The main difference between LWP and a standard process is that all LWPs share the same address space [2]. In computer science, threads are a small sequence of instructions that can be managed by a scheduler. A multi-threaded process is an application that contains multiple parallel execution flows. Threads represent processes in Linux [60, 2].

3.1.1 Linux Processes

In our study, we use graph theory to define a process. Graph theory describes graphs as G = (V, E). A graph is a structure representing the relationships of the objects [101]. Objects are called vertices, nodes, or points. The relationship between the pair of objects is called an edge, link, or line. Conventionally, direct acyclic graphs (DAGs) describe applications [63], where the nodes represent tasks, and the edges symbolize the communication cost. Provided a multi-thread environment, each node in the DAG gets created as a thread. The scheduler will treat the threads as individual processes [111].

The Linux process creation can be thought of as a computer program that utilizes numerous system resources to create a single process [3]. Unfortunately, every application represented through a DAG most likely will have system calls invoking threads that travel from the user-space to the kernel-space. System calls are a collection of application programming interface (API) calls that allow the user-space to request services from the kernel. Whenever a new process is invoked, a set of system calls

occur, introducing overhead. For these precise reasons, provided an N to N mapping between threads and the tasks composing a DAG, the scheduler will under-perform when the number of streamed applications is too large. In this study, given the nature of a DAG, every application that is represented through a DAG most likely will have system calls invoking threads that travel from the user-space to the kernel-space. System calls are a collection of application programming interfaces (APIs) that allow the user-space to request services from the kernel. Whenever a new process is invoked, a set of system calls occur, introducing overhead. Therefore, provided an N to N mapping between threads and the tasks composing a DAG, the scheduler will under-perform when the number of streamed applications is too large.

With these considerations in mind, we can now explain the limitations of the Linux scheduler in detail and our proposed approach in Section 3.2.

3.1.2 The Linux Scheduler: Data Structures

In this section, we describe the structures used by the real-time scheduler. The Linux scheduler uses a general run-queue that embeds other run-queues that manage jobs from policies other than CFS. Figure 1, captures this relationship. The primarily per-CPU run-queue data structure referred to as struct rq, has a structure struct rt_rq that embeds information about the real-time tasks placed on the per-CPU run-queue. Linux will organize processes based on their priority. A priority-indexed array of type struct rt_prio_array. Priorities range from 0-99, where priority levels "0" and "99" represent the lowest priority and the highest priorities a process can have, respectively.

An rt_prio_array consists of an array of sub-queues. There is one sub-queue per priority level for a total of one-hundred, and each queue contains the runnable real-time tasks. There is also a bitmask corresponding to the array. It is used to determine

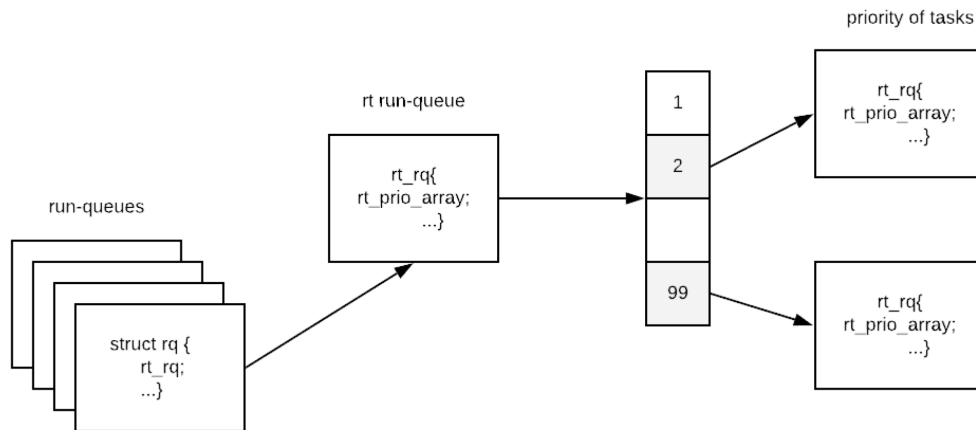


Figure 1: Scheduling Structures Used in the Real-Time Linux Kernel Scheduler

the highest-priority task effectively on the run-queue. The bitmask contains one bit per potential processor on the system. By default, all bits are set and, therefore, a process is potentially runnable on any processor. The priority array is mapped into a matrix structure, embedded into a scheduling entity; these are at a much lower level than task_struct. Linux uses this structure actually to schedule tasks. The mechanics for task allocation by SCHED_FIFO are not that complex. It will start from CPU0 and will try and find the first CPU that's idle. The RT load balancer ensures that the rt_rq is not saturated with jobs and will try and distribute the tasks with highest resources across all CPUs so that tasks are not hogging a CPU, and if necessary it will trigger the RT throttling, a safety mechanism that will try and avoid non-RT tasks from starving.

3.2 A New Paradigm for Scheduling

To bridge the gap between real-time scheduling theory and practice, we propose a scheduling framework that uses concepts from Mollison et al. [72] and Sanchez et

al. [86] where they propose a user-space application to manage scheduling. In other words, the task scheduling is decoupled from the Linux kernel, thereby providing a real-time runtime scheduler. In this study, we follow the implementation of the user-space emulation framework (USEF) [61] to overcome some of the limitations of the Linux scheduler. Using the user-space scheduling framework, we program the scheduling policies and stream domain-specific applications. The extended version of USEF manages, controls, and monitors the process scheduling on a target SoC. The emulation framework is capable of dynamically switching between different scheduling policies. It uses the real-time Linux thread policy [60] to guarantee the complete execution of a task from start to finish without preemption. The procedure limits the number of threads to be equal to the number of processing elements. The user-space scheduler has one dedicated processor, core 0 (main thread), that will submit tasks wrapped as function pointers to those worker threads corresponding to core 1, core 2, core 3, FFT 0, and FFT 1.

As stated before, the Linux RT scheduler has two policies first-in-first-out (FIFO) [56] and round robin (RR) [78]. The first policy chooses the task with the highest priority among all tasks in the priority array. It attempts to execute this task until completion without yielding the CPU voluntarily [1] . The second policy, RR, works similarly to the FIFO policy, but it voluntarily yields the CPU. After every quantum of time (time-slice), the operating system will execute the next RT task with the highest priority until that task completes or the quantum is expired.

To implement a real-time runtime scheduler free of preemption, we must allow the RT threads to run indefinitely. In the proposed framework, the total number of threads is equal to the number of processing elements. Executing one RT thread per processing element indefinitely will stall all other processes, including system

[1]Note that two RT tasks with equal priority do not preempt each other.

processes. This is unsafe, as system processes must be executed for safe operation of the operating system. The scheduler in Linux uses a safety mechanism to ensure that system processes are not permanently stalled. The scheduler parameter sched_rt_runtime_us defines the number of microseconds in a second that is allocated to execute RT threads. By default, Linux executes RT threads for 95% of the time in a second and system processes for the remaining 5% (the value of the parameter is 950000). We tweak this parameter to increase the priority of the user-space worker threads, by setting the value of sched_rt_runtime_us parameter to 999999. Consequently, Linux executes RT threads for 99.99% of the time and reserves 1μs to perform other system tasks.

By setting sched_rt_runtime_us parameter to 999999 (each digit has μs granularity), Linux executes the RT threads for 99.9% of the time and reserves 1μs to perform other system tasks that are not RT.

Assuming that the framework has RT threads that run without getting preempted, the approach suggested in this is work uses as many worker threads as resources are available (resources like CPU and accelerators) using the concept discussed by Sanchez et al. [86]. In our approach, one core is in charge of orchestrating the RT worker threads. DAGs will provide the inputs of tasks that an overlay processor will schedule (according to the scheduling policy) to an available resource. This approach is much more flexible than the existing kernel scheduler since we can easily use libraries that are capable of working outside the kernel. Moreover, our proposed framework can dynamically switch scheduling policies at runtime and enables user-space scheduling. This framework avoids using more and more memory for every newly forked process because instead, it will submit jobs to the worker threads already created awaiting for new jobs. Some security challenges are definitely prone to appear while selecting this hybrid-framework approach, but this is beyond the research of this master's thesis.

Chapter 4

HEURISTIC SCHEDULING ALGORITHMS

The meaning of heuristic derives from an ancient Greek word that means to discover. Heuristic-based schedulers can be used in scheduling to find a solution, it might not be optimal, but it is done in a simple and relatively short time. It is an approximation algorithm because it tends to find an optimal schedule, although not always succeed. Most heuristic schedulers are useful when having to reorder a large number of tasks.

This section introduces examples of the commonly known heuristics algorithms used in scheduling (Chapter 5 describes the machine learning algorithms), some of them being developed and used as part of this Master's thesis. To precisely explain the algorithms, and to enhance a better understanding, the algorithm pseudo-code is added to the description.

Some heuristic schedulers might be considered simple in concept. However, some of them have proven to perform better than complex algorithms, making them lighter and faster. The fact that simple schedulers sometimes perform better than sophisticated schedulers becomes a challenge for the proposed IL solution. Schedulers are organized in increasing order of complexity.

4.1 First-In First-Out (FIFO)

First-in-first-out [49] is a very simplistic algorithm, which is also known as first come first serve (FCFS). It is perhaps one of the most well-known algorithms. FIFO will take the first task that is ready and will assign it to the first available resource. This algorithm can be handy when (1) the tasks do not interleave with each other,

and (2) the number of simultaneous tasks does not exceed the maximum number of available PEs. Nonetheless, FIFO scheduling does not take task deadlines into account; hence, it fails at meeting time constraints.

Algorithm 1 FIFO Pseudo-Code Algorithm

```
1: procedure Compute FIFO
2:     for each resource do
3:         if resource = free then
4:             fifo_idx = index of resource
5:             break;
6:         end if
7:     end for
8:     map task to fifo_idx
9: end procedure
```

4.2 Earliest Finish Time (EFT)

Earliest finish time [91] is a heuristic algorithm that compares the execution time of the current tasks across all processing elements and tries to schedule the tasks to a free resource that will result in the earliest finish time. Hence, this scheduling algorithm will allocate tasks to available processing elements, but it will not re-order tasks.

The EFT algorithm needs to know the execution time of tasks, which might differ for different PEs. To guarantee an accurate estimation of task execution times, one has to run the tasks on each target PE multiple times and profile the average execution times. The EFT algorithm first traverses all the available resources to find the processing elements on which the finish time is the lowest. The algorithm will

assign the resource in round-robin fashion if multiple of them have the same finish time.

Algorithm 2 EFT Pseudo-Code Algorithm

```
1:  procedure COMPUTE EFT
2:      for each task in ready queue do
3:          for each resource do
4:              if task execution on resource(i) ≤ availability of resource(i) then
5:                  eft_idx = index of resource
6:                  update resource availability(i)
7:              end if
8:          end for
9:          map task to eft_idx
10:     end for
11: end procedure
```

4.3 Heterogeneous Earliest Finish Time (HEFT)

Heterogeneous earliest finish time is a list scheduling algorithm [96]. This algorithm is inspired by EFT, although it is far more complex. The goal of HEFT is to schedule a set of dependent tasks to a network of heterogeneous workers taking communication time into account. HEFT will use as inputs a set of tasks represented by a DAG, a collection of workers, the cost to execute tasks on different workers, and the times to transfer data from each job to each of its predecessors. It descends from list scheduling algorithms.

Many scheduling policies treat tasks in the natural spawning order as they arrive at the scheduler code. HEFT is capable of two things: 1) Determining from a set of

Algorithm 3 HEFT Pseudo-Code Algorithm

```
1: procedure CALCULATE THE RANK
2:     for each task in DAG calculate average exec.time on all PEs do
3:         if task is last task then
4:             rank of task = average of task
5:         else
6:             rank of task = average of task + max(rank(predecessors))
7:         end if
8:     end for
9: end procedure
10: procedure MAP TASK TO RESOURCES
11:     for each task in list of ranked tasks do
12:         if task is first task in the list then
13:             map task to processor with minimum execution time
14:         end if
15:         if  task and its predecessor on the same processor Pj then
16:             comm_time = 0
17:         else
18:             comm_time = communication time between two nodes
19:         end if
20:         for each processor do
21:             task_execution_time = execution_time of task
22:             on processor + comm_time + predecessor_execution time
23:         end for
24:     end for
25: end procedure
```

tasks T=T1, T2, T3, the order in which the whole scheduling will finish the earliest. 2) Allocating tasks to the resource on which the expected execution time is minimum. Keep in mind that HEFT is a greedy algorithm, and greedy algorithms are known to not converge for all cases. However, HEFT has a simple enough implementation in terms of memory and overhead, making this policy a conventional implementation for many scheduling policies.

The characterization of the HEFT algorithm has been adapted from the work [5] focused on dynamic task scheduling. The core concept is that all the nodes that represent tasks are scheduled based on a computed value called ranking. First, the algorithm tries to rank all the tasks respecting the DAG's dependencies, starting with the end-most task (the graph is traversed upwards). Secondly, it computes the dependencies and communication costs for each node. Since the first task will have the highest rank, and the last task that requires all the other nodes to be done will have the smallest rank. Next, the ranked tasks are arranged in descending order according to their rank value. Finally, tasks are scheduled for that processing element that guarantees the earliest finish time.

4.4 Earliest Task First (ETF)

The earliest task first algorithm [91] is a list scheduling algorithm because it compares the execution time of all tasks across all PEs mapping tasks to free resources resulting in the overall earliest finish time. Therefore, this algorithm can alter the order on which tasks are executed in pursuit of the combination that will satisfy minimum execution time.

ETF (not to confuse with EFT) has a higher computational cost than previously mentioned heuristic schedulers. By re-ordering tasks, this algorithm must traverse twice the pool of ready tasks. When all the resources are free, this algorithm behaves

Algorithm 4 ETF Pseudo-Code Algorithm

```
 1: procedure COMPUTE ETF
 2:     for each task in ready queue do
 3:         for each task in ready queue do
 4:             for each resource do
 5:                 if task execution on resource(i) ≤ availability of resource(i) then
 6:                     etf_idx = index of resource
 7:                     update resource availability(i)
 8:                 end if
 9:             end for
10:             map task to etf_idx
11:         end for
12:         set etf_idx to -1
13:     end for
14: end procedure
```

like EFT. Although this algorithm might not compute solutions as fast as other schedulers in real-world applications, in a simulated environment, ETF has shown similar reliability to more complex algorithms with less convergence time [11]. Thus, making this scheduler a great candidate to be used as the Oracle.

4.5 Completely Fair Scheduler (CFS)

Completely fair scheduler [51], the default Linux scheduler, is the primary desktop scheduler built in any Linux system. It tries to emulate the ideal behavior of a multi-tasking CPU by guaranteeing the fair execution of all tasks. To do so, it keeps track (using a virtual run-time) of tasks that are not running so that it estimates how much

should those tasks run for, to maintain a fair balance across the system.

The CFS is not based on queues; instead, it uses a red-black tree [70, 26, 60] structure acts as a timeline for future task execution. One of the structures is used for active processes, and the other one keeps track of expired processes. The expired red-black tree contains tasks that either have finished or got preempted. Hence, this scheduler can be preempted by any task in the system (tasks that are not I/O bound because those use a different scheduler). Besides, there are also priority levels that allow tasks to run for a more extended period.

To maintain fairness across the system, the Linux CFS utilizes expensive computational algorithms and re-balance functions that mainly migrate tasks from one queue (corresponding to a CPU) to another queue (corresponding to another CPU). In this way, it attempts to maintain fairness and avoid processes from starvation (Not enough CPU time). Unfortunately, the CFS is limited. This scheduling algorithm lacks support for accelerator awareness. Currently, it will consider only CPUs as processing elements, as well as the amount of storage involved required to run the completely fair scheduler.

Algorithm 5 CFS Pseudo-Code Algorithm

1: **procedure** COMPUTE CFS
2: *the left-most node of the scheduling tree is chosen*
3: *If the process completes execution, it is removed from the structure*
4: *If the process expires its time-slice, it is reinserted into the structure based on remaining execution time*
5: *The new left-most node is chosen*
6: **end procedure**

Chapter 5

DATA-DRIVEN SCHEDULING ALGORITHMS

Machine learning is useful in identifying patterns in scheduling. Moreover, scheduling can be considered a classification problem rather than an optimization problem [99, 53]. There are multiple techniques in machine learning that could be potentially applied to scheduling to solve classification problems. After designing and training a model using a simulator like DS3 [11], the model is exported and integrated with the proposed IL scheduler framework (it uses the model's weights after training to re-construct a model and run the inference). The models are compared with some of the conventional heuristic schedulers introduced in Chapter 4. This section introduces different ML-based techniques to solve scheduling, but the scope of this thesis focuses on imitation learning scheduling policies, not Reinforcement Learning (RL). Formulating the reward function for the reinforcement learning is not trivial [88, 38, 65, 64] and not practical. IL is a more straightforward approach. For an in-depth discussion of alternative data-driven algorithms, we suggest Peter's Flach book on machine learning [34].

Knowing that ML can potentially benefit scheduling, we need to answer the following question: How can machine learning be applied to optimize scheduling? Assuming that the domain applications are known at design-time, we make two observations. First, we build an expert, i.e., an Oracle that stored the optimal decisions, to train the ML model. Second, we ensure that the framework can imitate Oracle's behavior at run-time without adding too much overhead.

5.1 Genetic Algorithms

The process of natural selection inspires genetic algorithms (GA) [102]. GAs reflect the evolution of life, i.e., the survival of the fittest of beings. GAs use techniques and concepts extracted from the theory of evolutionary biology concepts like inheritance, mutation, crossover, and selection. In computer science, GAs are defined as a search technique to find a set of approximate solutions for optimization and classification problems.

The approximations are ranked according to the score of their fitness function. The best candidates are grouped and combined using genetic operators, creating new solutions. This process is also known as recombination; the rest of the solutions are removed. Each new solution contains traces and similarities to its parent. This process repeats until a solution exceeds a minimum threshold on the fitness value. The process is also known as evolution, and it usually starts from a population of randomly selected individuals, the groups or pairs are known as generations. In contrast to some machine learning techniques, GA will not iterate until convergence because the algorithm does not have a convergence rule. Instead, the algorithm will terminate when several pairs (or generations) of solutions are produced, or a satisfactory fitness level is reached.

Genetic algorithms are useful in solving NP-hard problems [8]. In particular, it offers a solution to solve one of the most notorious scheduling problems: the traveling salesman problem (TSP). The problem consists of a salesman that is given a set of cities. The salesman has to find the shortest route to visit each city exactly once and return to the starting city. The traveling salesman problem is described as follows:

TSP = (G, f, t):

G = (V, E) a complete graph,

G is a graph that contains a traveling salesman tour with cost not exceeding t,

f is a function of $V \times V \rightarrow Z$,

$t \in Z$

Though Genetic algorithms have proved to be a fast and powerful problem-solving approach, we distinguish some limitations. Repeating the fitness calculation is extremely costly; a single function evaluation might compel several hours or days of training. For this precise cause, although this algorithm is proven effective (It traverses almost all possible scenarios), other algorithms (decision trees, neural networks) might be more efficient, achieving similar results with faster convergence time. For this study, speed in convergence is mandatory, since we do not want to add an excessive overhead when computing a scheduling policy.

5.2 Decision Tree Algorithms

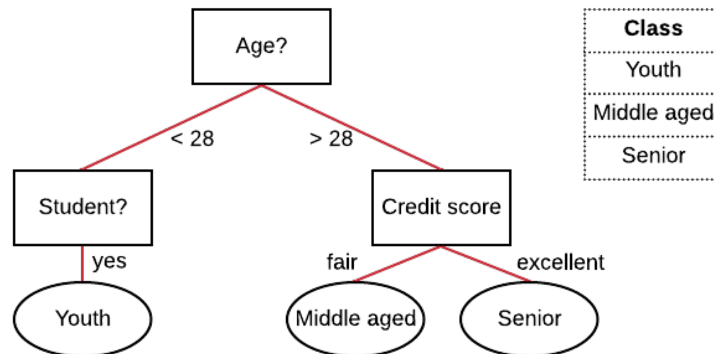


Figure 2: Example of Decision Trees Used for Classification

Decision trees [52] are another example of data-driven and structural algorithms used through supervised learning for regression and classification of single and multiple targets. Similarly to heuristics, they learn from the data and try and approximate

a solution to a threshold or a curve through an if-then-else set of rules. The complexity of decision trees is not measured in layers but in depth. The deeper the tree, the more complex.

Decision trees build their models in the form of a tree structure. A single decision tree is also known as a white box. Each level is associated with a hierarchy f.e the root node sits at the top of the tree where the depth is equal to zero. Every individual node from the decision tree corresponds to a question; the objective of this question is to divide the data into a smaller subset that corresponds to a class. The reader might notice the importance of selecting the question. To choose a question that splits the data, the model chooses an input feature and does the split. The Gini score represents the impurity of the class. Choosing a question and calculating the Gini score for all the features in the set. Finally, the root node is determined by the feature with the lowest Gini score.

Pruning is the process of reducing the size of the tree by turning not-relevant branch nodes into leaf nodes. Commonly the process of pruning is done to prevent data from overfitting. Overfitting is a problem of having excellent training accuracy but mispredictions upon new data. Overfitting will affect the performance of the model when fed new data to classify. Finally, fitting the tree is the process of finding the smallest tree that could fit the data. Typically, the dimensions of the tree are determined by choosing the tree with the lowest cross-validation error.

Decision Trees are an attractive candidate. At first glance, they seem to appear as a faster alternative to artificial neural networks [53]. Once the model is trained, through a set of simple rules (if-else statements), the policy predicts an output. However, in complex classification problems, the size of decision trees can be larger than small deep neural networks (less than 2000 weights).

5.3 Artificial Neural Networks

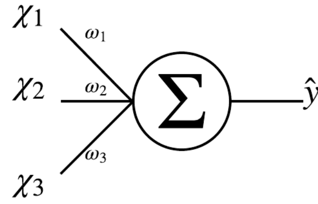


Figure 3: Perceptron Model Minski-Papert 1969 [71]

An artificial neural network (ANN) [36] is an algorithm designed to simulate how the human brain processes information, is inspired after the human brain and is proficient at recognizing patterns. The architecture uses multiple layers that extract features from the input. A typical example is image processing, where layers closer to the input represent edges. The layers closer to the output typically represent concepts more relevant to a human (f.e faces, shapes, and objects). Neural networks are considered a universal approximation theorem since the model tries to correlate and approximate inputs-to-outputs using continuous functions. Some examples for these continuous functions are TanH, Sigmoid, ReLU and Softmax [34, 7, 59]. Usually, these are called activation functions, and they determine the output a node will generate, based on some inputs.

Deep neural networks (DNN) [58] evolve from single neuron perceptron - the lowest unit belonging to a network. Figure 3 is an example of the original Minski-Paper model proposed in 1969. Figure 3 is described as follows:

$$\hat{y} = \Sigma_{i=1}^{n} \omega_i * \chi_i$$

multi-layer perceptron (MLP) [93] or deep neural networks are networks of perceptrons and linear classifiers [34] where the architecture of an MLP consists of an input

layer, some hidden layers, and an output layer. These layers are composed of single perceptrons. The depth of the network is determined by the number of stacked layers. According to the number of layers, some architectures will be more shallow (less number of stacked layers), and some others will go more in-depth (a higher number of stacked layers).

Neural networks are prevalent and well-known for classification problems, considering scheduling as a classification problem; it makes neural networks a candidate for this thesis. To train a model, the user should specify the target. Targets are usually identified by labeling data. The methodology is known as supervised learning, which differentiates from unsupervised learning precisely because of the labels. Examples of supervised learning are speech recognition, image recognition, and bioinformatics, to list a few. While scheduling is traditionally studied as an optimization problem, we consider scheduling as a classification problem by envisioning processing elements as classes in this work. Optimization problems have static solutions that can only be satisfied when evaluated under the same circumstances; in any other situation, it fails to be optimal. On the other hand, classification can be primarily trained offline under supervised learning to approximate the model to an optimal solution. Theoretically, the model can continue learning while deployed [104] by continually adjusting its weights (This is far beyond the scope of this thesis), making classification using DNN a much more generalizable approach to address the scheduling paradigm.

First, the system should be able to classify and map tasks-to-resources depending on real-time system alterations dynamically. The input parameters in the neural network represent these variables that affect the system state. Before training, the neural network with a customized architecture is generated. The neural network is initially naive, and its weights have default values. By training the neural network with sufficient imitation learning data and a large number of epochs, the model weight

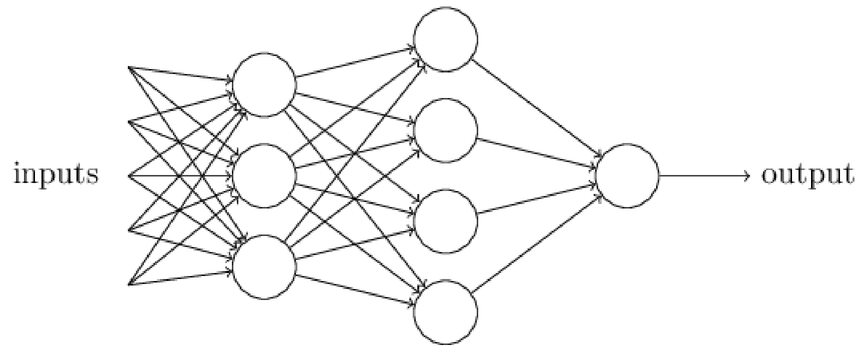


Figure 4: Example of Multiple Layers Stacked Forming a Neural Network

converge to stable values. Otherwise, a threshold is specified, and the training will complete once the training reaches that threshold. Depending on the dimensions of the imitation data file, the training process can last up to several hours. However, performance issues can arise with a poorly trained DNN. Similarly to decision trees, overfitting and underfitting are the two critical reasons why DNNs have poor performance. Typically, overfitting is related to deep architectures, and a weak or limited training data-set. It leads to situations where training will have a deficient error, but a high testing error. In contrast, underfitting is analogous to small architectures and a reduced training data-set. The model is unable to learn the relationship between x and y. High training and testing error is the synonym of underfitting.

Chapter 6

IL-BASED SCHEDULING FRAMEWORK

This Chapter presents an in-depth explanation of the IL-based scheduling policy used in this work. We cover its design, integration to the user-space scheduling framework, and evaluation on the Xilinx Zynq Ultrascale+ ZCU102. Starting from the highest level of abstraction, we introduce the Python/Weka [39] neural network/decision tree models. Then, we describe the conversion of the proposed IL models to C/C++. Finally, we present the framework used to deploy the ML policies and the deployment process. Figure 5 depicts a high-level representation of the IL scheduling framework's integration and its components.

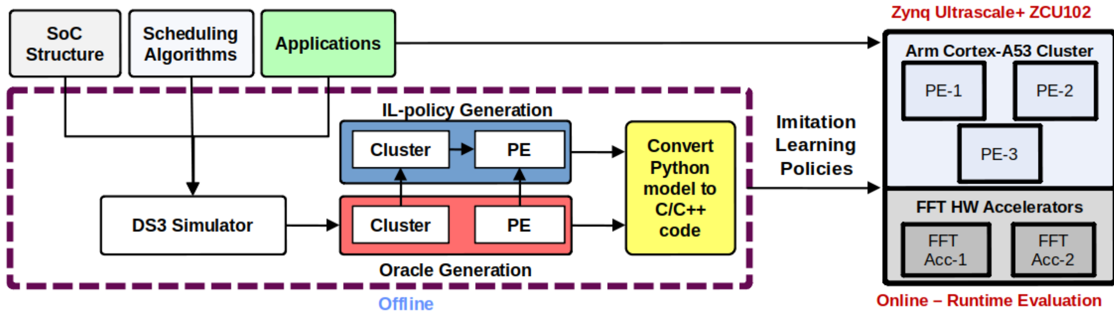


Figure 5: Overview of the Proposed Imitation Learning Scheduling Framework

The rest of this Chapter is organized as follows. First, Section 6.1 describes the Oracle, which acts as the expert to train the IL-based schedulers. Then, Section 6.2 details the architecture of the deep neural network and decision tree model. Section 6.3 summarizes the training results of the machine learning models. Section 6.4 describes the software AI Transformer [1] and Weka2c [47], conversion engines that will transform a high-level programmed ML model into a faster and more efficient

code. Next, we discuss the architecture of the user-space scheduler to adapt the IL policies using the pre-trained weights in Section 6.5. Finally, Section 6.6 describes the target SoC on which the framework with the new scheduling policy will be deployed.

This approach can be extended to other IL policies since the only difference is the estimation function computed to predict a resource. Furthermore, it can be generalized for different platforms by updating the features.

Algorithm 6 Algorithm for the proposed IL Models

```
 1: procedure COMPUTE IL MODEL
 2:     for each task in ready queue do
 3:         for each resource do
 4:             prepare the inputs for IL models
 5:             using features run prediction for any IL model
 6:             if task supports predicted resource then
 7:                 map task to resource
 8:             end if
 9:         end for
10:     end for
11: end procedure
```

6.1 Building the Oracle

An Oracle is a collection of all system states and the corresponding scheduling decisions [30]. A dataset contains the features and labels for corresponding states and decisions in the Oracle. Some examples of system variables are PE availability times, the situation of the scheduling queues, system utilization, and system load. The most relevant variables are often chosen as input features to the models.

The Oracle uses a scheduling policy π that makes optimal scheduling decisions. The policy that the Oracle uses makes optimal scheduling decisions by traversing the pool of ready tasks to compute the best execution placement. According to the experimental results carried through [11, 61], earliest task first (ETF) is a computational expensive heuristic scheduling policy that is proven to perform similarly to Integer Linear Programming [6], a mathematical approach for optimization. Generating data is accomplished using a domain-specific simulator DS3. The simulator allows for different SoC configurations for a variety of Arm processors (Cortex-A53, Cortex-A7, and Cortex-A15) and general-purpose hardware accelerators (FFT, IFFT, and scrambler). The simulator provides different scheduling algorithms. The user can select a scheduling algorithm for the expert through the configuration file, some of the currently available policies are ETF, minimum execution time (MET), shortest task first (STF) and other table-based schedulers.

ETF iterates over all ready-to-run tasks and finds the combination of location and order that satisfies the earliest possible finish time; this heuristic also, if necessary, can account for the communication overheads (inter-memory, PE to PE). However, the overhead (cost of time that the computation introduces) that ETF generates, provided that outside the simulator, we can not decouple the overhead from the execution time, EFT will not perform as simulated. Compared to other optimal schedulers, for instance, integer-linear-programming (ILP) and constraint programming (CP) [14], ETF achieves similar results and is a more straightforward approach to implement. For this study, ETF is the scheduling policy used to generate the Oracle. The Oracle generation is followed by the generation of a dataset which includes features for each system state and the corresponding label to train ML models.

6.2 Describing the Architecture of the ML Model

We introduce two ML scheduling models: IL-DNN and IL-DT. Both share the same architecture (features, attributes) and are trained using the same data previously generated by the Oracle; however, the approach to build and train each model is different.

Table I: Table of Static and Dynamic Features of the ML Models From Task T_i's Point of View

Feature Type	**Feature Description**	**Feature Category**	**Number of Features**
Static	Execution time of task T_i in the DAG	Task	2
	Predecessor ID(s) of task T_i	Application	5
Dynamic	Earliest time when PE is ready for task execution	PE	5
	Predecessor ID(s) assigned PE of task T_i	Application	5
	Total		**17**

The proposed ML models receive 17 inputs, a combination of static and dynamic system parameters and task attributes, as seen in Table I. Extracting attributes from applications is possible since this study considers applications that are well-defined and are characterized using DAGs. Therefore, expected execution time, predecessor(s), and communication costs are known parameters before scheduling. Table I below summarizes the input features organized by index:

Table II: Summary of the Features Inputs of the Proposed IL Policies

1	2	3	4	5	6	7	8	...	12	13	...	17
RA_1	RA_2	RA_3	RA_4	RA_5	ET_{core}	ET_{accel}	PID_1	PID_n	PID_5	$PAID_1$	$PAID_n$	$PAID_5$

The resource availability (RA_i) for each PE: core $C_i = PE_{1,2,3}$ accel $A_i = PE_{4,5}$ and the expected execution time (ET) of task T_i on core C_i and accelerator A_i. The predecessor(s) id (PID_i) of task T_i, up to a maximum of five predecessors, and the predecessor assigned id ($PAID_i$) of task T_i, up to a maximum of five predecessors.

6.3 Summary of Training Results

The dataset that contains the features and labels for corresponding states and decisions in the Oracle is used to train the IL models. Typically in machine learning, the dataset containing the features is split into the original dataset's subsets. The training is performed, utilizing 80% of the data for training and the remaining for testing. Furthermore, the larger dataset containing the training data is split into 80% of the data for training and 20% of the data for validation. The dataset is composed of approximately 220,000 entries of features and labels, and it is used throughout this study.

The model will train for several epochs in batches of data. Once the model has completed training, the model is tested using the validation dataset (a small portion of the data previously not exposed to the model) to reveal an estimate of the performance. Finally, the data is tested with the test dataset (larger than validation) to reveal the final accuracy. We calculate the final accuracy value is by averaging the results over multiple runs to achieve a reliable prediction score.

Table III: Accuracy Comparison of the ML Models Based on Test Data

Scheduling	**Accuracy (%)**	**Mean Absolute Error**
IL-DNN	97.31	0.02
IL-DT	89.92	0.05

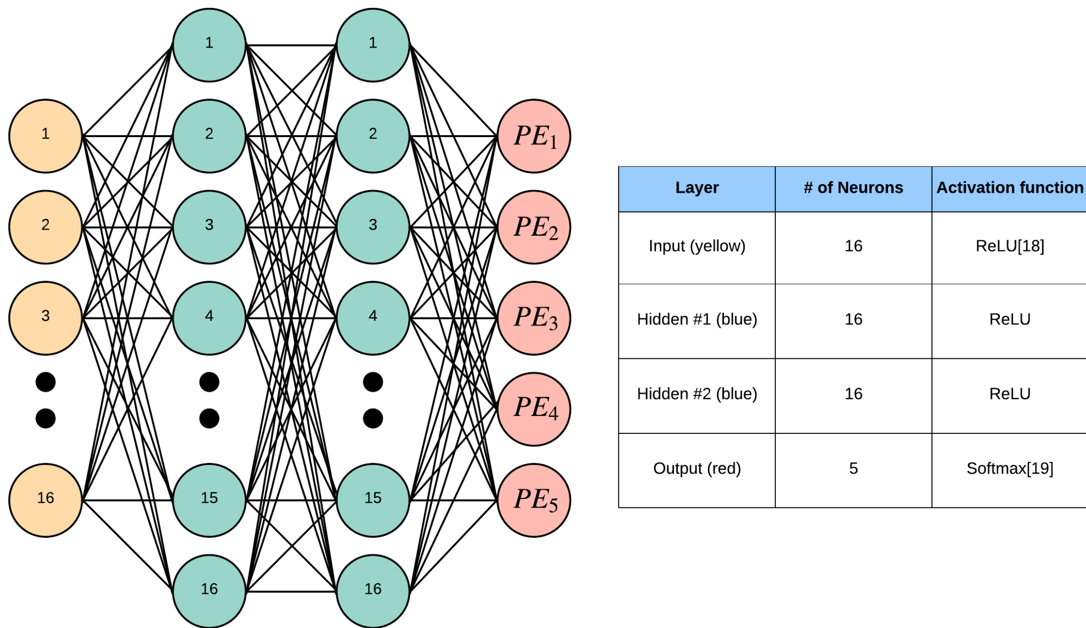
We use a 12-core Intel(R) Xeon(R) CPU E5-2630 server (@ 2.60GHz) for training. Table I compares the classification performance (accuracy and MAE) between the IL-DNN and the IL-DT. Although there is a significant drop in the prediction accuracy, IL-DT results in lower overhead and a much simpler model to export for hardware integration as compared to IL-DNN. Several metrics describe the statistical behavior, e.g., mean absolute error (MAE) measures the average magnitude of the errors in a set of predictions. In general, we summarize and assess the quality of a machine learning model by computing the MAE. When a model is trained, the MAE indicates the average error of our predictions over a set of data.

6.3.1 Training the Deep Neural Network

Neural networks are effective for predictive modeling, but many configuration parameters should be tuned thoroughly through experience and trial & error. Using Keras [36] public set of libraries engineered for Python, the DNN is built from scratch. Keras developed a set of high-level APIs for building and training deep learning models. Keras has a sequential model builder that allows the user to stack layers with different configurations (e.g., number of neurons, and activation function). Once the model is compiled, a naive, fully connected neural network model is constructed. Figure 6 shows the configuration of the neural network used for the IL-DNN policy.

The following equation: $\hat{y} = \Sigma_{i=1}^{n} \omega_i * \chi_i$, where $\hat{y}$ is the approximated output, χ_i are the inputs and ω_i the weights. During the training phase, the model will process inputs generated by the Oracle, and will adjust its weights ω_i in steps specified by the learning rate (lr), the lr of the compiled model used in the evaluations is $lr = 0.001$, the Keras compiled model iterated for 400 *epochs* and trained for over two hours.

ReLU [7] is the activation function for the neurons in the inner-layers. It has shown higher accuracy percentages during testing and validation than Sigmoid or



Layer	# of Neurons	Activation function
Input (yellow)	16	ReLU[18]
Hidden #1 (blue)	16	ReLU
Hidden #2 (blue)	16	ReLU
Output (red)	5	Softmax[19]

Figure 6: (Left) Representation of the Fully-Connected DNN. (Right) Table Summarizing Architecture Insight

Linear (Since the final implementation is going to be in C, we want a small and simple function). ReLU is linear for all positives values and zeroes for all negative values. There are many reasons to pick ReLU as the activation function. ReLU is cheap to compute, converges fast, and is sparsely activated. Sparsely activated meaning that, since negatives inputs are treated as zero, it is likely for some neurons to not activate at all, emulating the behavior of biological neural networks. Softmax [59] is used as the activation function for the output layer. Softmax is used to ensure that the neurons' output values are in the range of [0,1], representing the predicted probabilities. Therefore the neuron containing the highest probability will correspond to the predicted PE.

The IL-DNN machine learning model achieves a maximum of **97.7% accuracy.** Many multi-class classification problems use tools like confusion matrices and loss/ac-

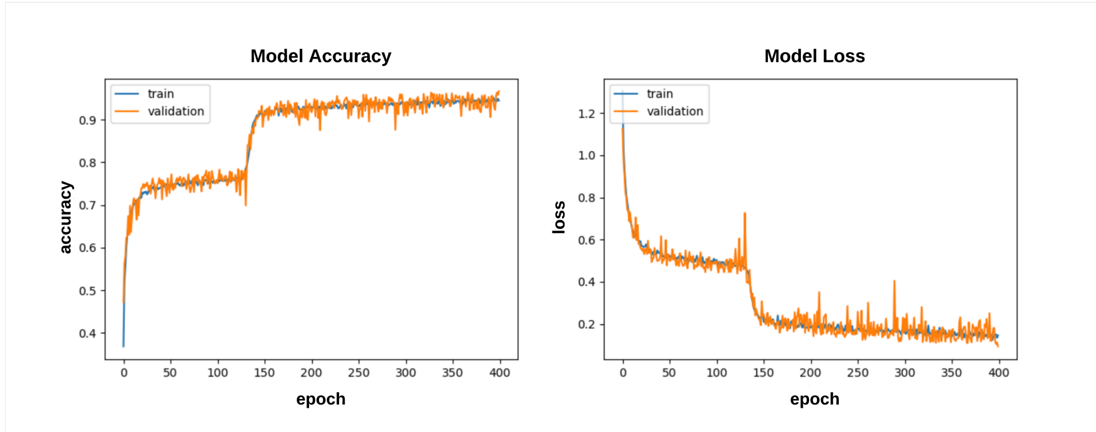


Figure 7: (Left) Plot of Model Accuracy Using the Training and Validation Datasets. (Right) Plot of Model Loss Function Using the Training and Validation Datasets

curacy function to illustrate the performance and have a better understanding of the fine-tuning. Figure 7 shows the accuracy and the loss function development over 400 epochs of the IL-DNN model. The figure in the left corresponds to the accuracy of the training (blue) and the validation (orange) dataset. Similarly, the rightmost figure corresponds to the cross-entropy loss function of the training (blue) and the validation (orange) dataset. Both cross-entropy and accuracy are showing good convergence behavior. Despite some irregularities, after 150 epochs, the neural network converges. Given the shoulder-shaped figure that appears on both graphs, it might suggest that the model is overfitted. To verify this, we introduced random tasks to the model and added noise to the feature values for the predictions on the real system. We conclude that tasks that can be executed on a fixed-function accelerator should always be predicted in such a resource. When performed the evaluations, we observed how the tasks that should be accelerated were getting assigned to such resources.

The classification model's performance over a set of known valid values is described using a confusion matrix. Figure 8 depicts the summary of the class prediction results

Predicted Label

True Label	CPU 0	CPU 1	CPU 2	FFT 0	FFT 1
CPU 0	0.97	0.00	0.01	0.01	0.01
CPU 1	0.02	0.93	0.02	0.03	0.01
CPU 2	0.01	0.01	0.96	0.02	0.01
FFT 0	0.01	0.01	0.01	0.97	0.00
FFT 1	0.01	0.02	0.01	0.03	0.93

Figure 8: DNN Confusion Matrix Showing Accuracy and Error per Class

for this particular classification problem. Confusion matrices are useful to determine the classification probabilities across the different classes, the accuracy per class, and the precision. Testing the model with unseen data illustrates how the model guarantees, at least, 93% accuracy for any class.

6.3.2 Training the Decision Tree

Decision trees have two key steps: Induction and Pruning. During the Induction step, the DT structure is built. That is, the set of hierarchical decisions are established at each node using the model's input features. Pruning is the process of removing unnecessary parts of the structure, making the model less prone to overfitting and simplifying its complexity. Decision trees can be easily ported to another platform. Once the model is trained, it is used in the emulation framework [61] by converting the hierarchical questions at each node into if-else statements.

C4.5 (J48) is an algorithm used to generate a decision tree developed by Ross Quinlan in his book [79]. C4.5 is an extension of Quinlan's earlier ID3 (Iterative

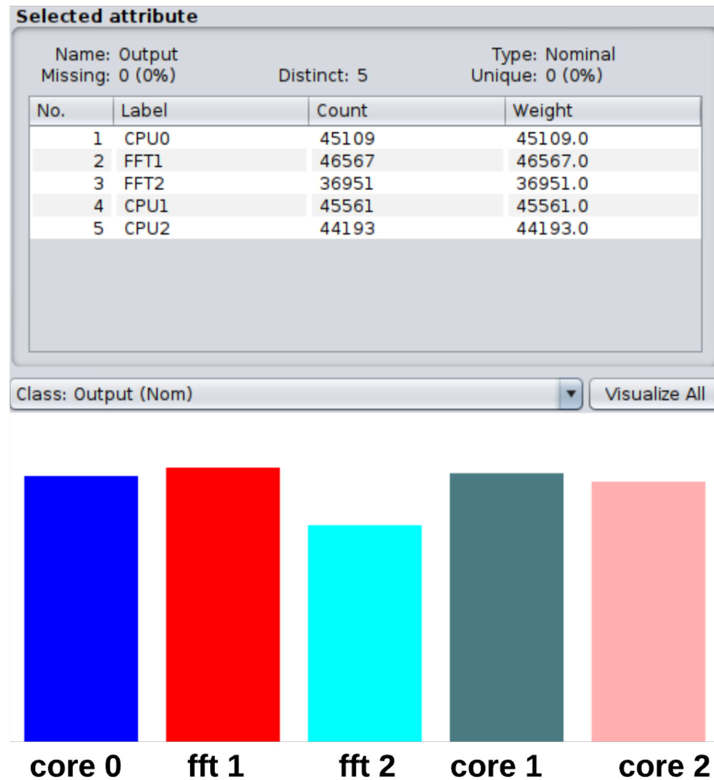


Figure 9: Weka Software GUI: Example of Training Data Used for the J48 Algorithm

Dichotomiser 3 [45]) algorithm. The decision trees generated by C4.5 is used for classification; for this reason, C4.5 is often referred to as a statistical classifier. At each decision of the tree, C4.5 chooses the attribute of the data that most effectively splits its set of samples into subsets corresponding to the different classes.

J48 is an open source java implementation of the decision tree algorithm C4.5 [79], developed by a research team at the University of Waikato, through their data mining software: Weka [39]. Weka provides a comprehensive collection of machine learning algorithms, data preprocessing tools, and a user interface to build and test ML models. Weka is recognized as a landmark system in the machine learning community. It can

be downloaded through GitHub [100].

Predicted Label (columns) / True Label (rows)

	CPU 0	CPU 1	CPU 2	FFT 0	FFT 1
CPU 0	0.89	0.05	0.04	0.01	0.00
CPU 1	0.05	0.89	0.04	0.00	0.01
CPU 2	0.05	0.05	0.87	0.02	0.01
FFT 0	0.02	0.01	0.00	0.89	0.09
FFT 1	0.01	0.02	0.01	0.08	0.88

Figure 10: DT Confusion Matrix Showing Accuracy and Error per Class

Similarly to the previous section, IL-DT uses the same dataset for its training. Figure 9 is an example of the interface that Weka provides, from left to right, each bar graph corresponds to a label: the blue graph represents core 0, the red is fft 1, the cyan is fft 2, the green is core 1 and the pink represents core 2. Note how the classes are equally distributed amongst all the different processing elements, meaning that the dataset is not heavily targeted to a specific PE. The IL-DT machine learning model achieves a maximum of **89.9% accuracy**, as shown in Figure 10.

6.4 Converting the ML Model to a Low Abstraction Language

At this point, the machine learning models are trained and have their weights tuned. New ML models are instantly deployed using the trained weights. The objective is to port the trained model from high-level abstraction language to C so that it is embedded in the emulation framework. Since Python and Java (used by Weka) are high abstraction programming languages, they are typically outperformed

by lower abstraction languages like C [28]. The drawback of a lower abstraction language is the increased complexity. For instance, coding neural networks in Python is straightforward in comparison with C. A code-generator transforms Python to C/C++ while maintaining the functional equivalence. For example, AI Transformer reads the weights previously-stored from Keras and creates a library with a set of API that communicates with the model. The software AI Transformer is a cloud-based code generator (under MIT License) for deep neural network models. Given a DNN model, it generates C code optimized for embedded systems and edge computing devices. Similarly, there is a software tool to convert decision trees into a hierarchical set of if-else statements. Given a model built through Weka, Weka2c [47] converts a decision tree, similar to AI Transformer (set of API), to communicate with the ML model.

6.5 Extending the User-Space Emulation Framework (USEF)

Chapter 3 introduced a discussion between kernel and user-space schedulers. The work published with the new generation emulation framework [61] has proven better performance than CFS for discretized applications. This framework enables the execution of applications modeled as DAGs. Version 1.0 of the emulation framework (the version is coded in C) offers a user-space emulation framework structured the following way: one processor, core 0, will be designated the overlay processor (OP), the key to this approach is that the processor has the particular task of fetching the next task, assign that task to a particular resource while the rest of the cores are waiting to receive jobs, the remaining cores act as worker threads (WT) and are responsible for executing a specific function of a job. To generalize this approach: it is possible to have as many as N-1 cores individually assigned as WT with one core as the OP. Worker threads are either CPUs or custom accelerators, ideally mapping one-

to-one processing elements with threads should produce better results. However, this approach is limited. To communicate with custom accelerators, we require a CPU. For example, Xilinx uses direct memory address with protocol AXI (DMA-AXI) to communicate with internal blocks or peripherals. Having accelerator awareness is of interest to this thesis. Hence it was decided to time-share a resource between a worker thread corresponding to a CPU, and a worker thread corresponding to an accelerator.

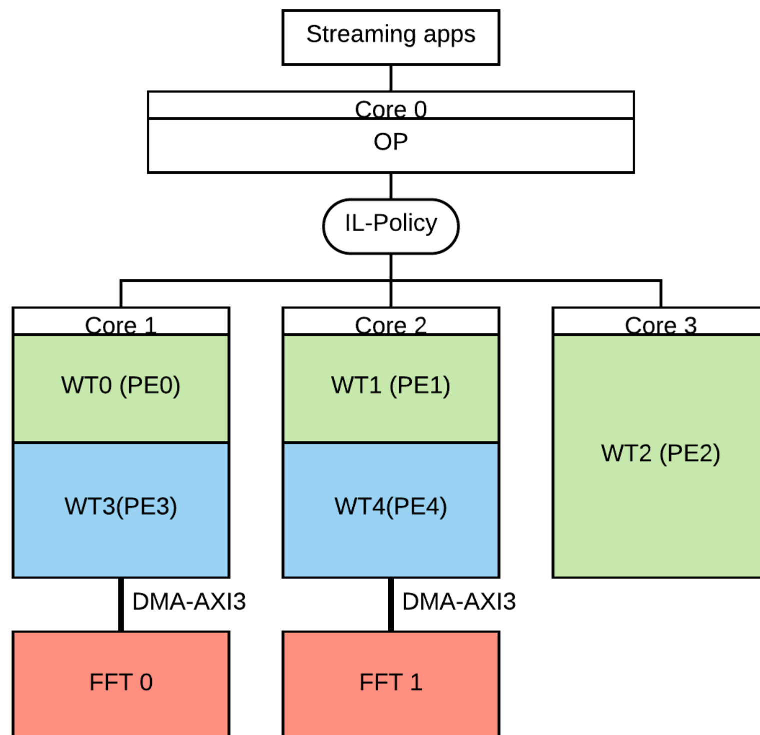


Figure 11: Extending the User-Space Emulation framework. The Proposed Approach has Five Worker Threads (WT), one Overlay Processor (OP) and Two FFT Accelerators

Figure 11 depicts how this customized version of the user-space emulation framework uses one Arm Cortex-A53 processor as the OP, three Arm Cortex-A53 processor

as WT (WT0, WT1, WT2, and WT3) and two FFT fixed-function accelerators also as WT (WT4 and WT5). WT are treated as individual Linux processes. Since the OS is Linux-based, we rely on its default scheduler to maximize the run-time of the USEF. The framework will hoard 99% of the CPU. Previously mentioned in Chapter 3, the scheduling policy FIFO will execute until completion or until the CPU is yield. Assigning the WT to be scheduled in the OS as RT-FIFO thread with priority equal to 99, ensures the total completion of the task.

Four wireless and radar streaming applications (Chapter 7 introduces the domain-specific applications) that are used to evaluate the model's performance. Naturally, the same applications (similar task structure and expected execution times) are used for the Oracle. These applications are coded in C and used in several studies [11, 53, 61, 88].

6.6 The System: Zynq UltraScale+ MPSoC ZCU102

The scheduling policies integrated with the extended USEF are evaluated on a popular commercial heterogeneous platform: Xilinx Zynq Ultrascale+ MPSoC ZCU102 [4]. This SoC features a quad-core Arm Cortex-A53 running up to 1.5GHz. Besides, this SoC has 16nm FInFET+ programmable logic. The device utilizes specialized processing elements to excel in wireless and communication applications.

The SoC is capable of running an entire Linux operating system. Using PetaLinux [109], the OS is customized so that the system is ready to execute programs and other functionalities. PetaLinux will generate a custom Board Support Package (BSP), including drivers and IP Cores [108], that allow for fast development. The hardware accelerators are implemented using programmable logic. Using memory-mapped streaming interfaces, the SoC communicates with the custom FFT accelerators. Direct Memory Address (DMA) enables the transmissions between the accel-

erator's buffers and the application user-space. Linux OS treats it as a module and will be able to perform read or write operations. Linux OS requires a driver to read-/write from the FFT accelerator, the user-space direct memory address driver [48], allocates contiguous memory blocks in the kernel space as DMA buffers and makes them available from the user-space.

Chapter 7

EXPERIMENTAL RESULTS

This Chapter presents the experimental setup and results of the proposed scheduling framework. First, we introduce the domain applications used for evaluation of the proposed approach in Section 7.1. Then, Section 7.2 and Section 7.3 evaluate the performance of the proposed IL-based scheduling approach. We compare deep neural network (IL-DNN) and decision tree (IL-DT, based on J48 [79] algorithm) with multiple heuristics and list-scheduling algorithms: MET (minimum execution time), EFT (earliest finish time) and the Oracle (ETF, earliest task first). Finally, Section 7.3 describes the study of the computational complexity of the proposed algorithms.

7.1 Domain-Specific Applications

This section presents the domain-specific applications that are used for the evaluation of the proposed IL-based scheduling approach. More specifically, we choose applications from the wireless communication and radar system domains. From wireless communications, we choose the popular WiFi protocol based transmitter and receiver applications [23, 11, 53]. Pulse Doppler and range detection are applications from radar systems chosen for evaluation [62, 29, 11, 53]. The pulse Doppler technique is used to calculate the speed of a moving object. In contrast, the range detection technique calculates the distance of a moving object with respect to a stationary object. The tasks in all these four applications, along with their execution times on Arm Cortex-A53 and hardware accelerator (wherever supported) are detailed in Table IV. The target applications are modeled as directed acyclic graphs (DAGs) in both DS3 and USEF.

Table IV: Execution Time Profiles of Domain-Specific Applications on Arm Cortex-A53 Core and FFT Accelerator [11]

Application	**Task**	**Execution time (μs)**	
		Arm Cortex-A53	**FFT Acc.**
WiFi-Tx	Scrambler	2	
	Encoder	20	
	Interleaver	8	
	QPSK Modulation	15	
	Pilot Insertion	4	
	Inverse-FFT	225	16
	CRC	5	
WiFi-Rx	Match Filter	15	
	Payload Extraction	5	
	FFT	218	12
	Pilot Extraction	5	
	QPSK Demodulation	79	
	Deinterleaver	10	
	Decoder	1983	
	Descrambler	2	
RangDet	FFT	68	30
	Vector Multiplication	52	
	Inverse-FFT	68	30
	Max Detection	10	
PD	FFT	30	6
	Vector Multiplication	30	
	Inverse-FFT	30	6
	Magnitude Computation	25	
	FFT Shift	6	

The four applications (WiFi-TX and WiFi-RX from wireless communications, and Pulse Doppler and Range Detection from radar systems) are used to construct workloads for evaluation of our approach. The target applications significantly differ from each other in terms of the number of tasks, types of kernels, individual task execution times, application execution time and number of tasks that are supported on hardware accelerators. For example, WiFi-TX and WiFi-RX applications contain 8, and 10 tasks respectively, with one task in each application supported on a hardware accelerator (Table V). Pulse Doppler and range detection applications are decomposed into 515 and 7 tasks, respectively (Table V). There are 256 tasks in pulse Doppler supported by hardware accelerators and one task in the range detection application can be accelerated in hardware

This rich mixture of applications with widely varying characteristics are useful in representing a large corpus of applications in the chosen domains. The proposed IL-based scheduler aims to schedule simultaneous streaming domain-specific applications. In this context, we construct workloads that are a random mixture of the four target applications, parameterized by different injection rates. The characteristics of one of the workloads used for evaluation is presented in Table V. The injection rates of the applications are varied, as indicated by the number of frames in the workload in the table.

7.2 Evaluation of the IL Policies on USEF

We evaluate the proposed IL scheduling policies using the user-space emulation framework [61]. The emulation framework is cross-compiled to work on a commercial SoC, the Xilinx Ultrascale+ ZCU102. To create a mix of workloads combining streaming applications, USEF provides a set of control knobs that inject jobs at a specified rate with a particular probability. The framework will generate logs of information,

Table V: Characteristics of the Target Domain-Specific Applications and the Workload Used for Evaluation

Application	**# of tasks in application**	**Appearances in workload**		**Avg. generation time of data frame (μs)**
		# frames	**# tasks**	
WiFi-Tx	8	199	1592	42.5
WiFi-Rx	10	199	1990	1230.0
RangeDet	7	99	693	542.0
PD	515	49	25235	2200.0

collecting execution time, and scheduling decisions for the analysis.

Table VI compares the overhead and storage of the proposed policies. IL-DNN utilizes lower storage memory, and IL-DT has a lower over-head.

Table VI: Scheduling Overhead Cost and Memory Comparison of The Proposed IL Policies on Single Pulse Doppler Frame

Policy	**Avg. Overhead(μs) on Arm Cortex-A53**	**Size (KB)**
IL-DNN	2.7	3.4
IL-DT	0.7	27.7

Computing the inference of a DNN on a single Arm Cortex-A53 is not the most efficient solution in terms of power and execution time. We have explored implementing the computation of the deep neural network on a custom accelerator. Preliminary studies indicate that the DNN computation is completed in 99 cycles on a custom accelerator. A single calculation could be completed in 100 ns, assuming a 1 GHz target frequency. Figure 12 shows how the IL-DNN(Acc) implementation decreases by 98% the overhead compared to the IL-DNN(Arm). Moreover, IL-DNN(Acc) computes

with the lowest overhead amongst the rest of the schedulers.

The workload used in Figure 12 contains a mix of 500+ frames of four interleaved wireless applications: WiFi-Tx, WiFi-Rx, RangeDet, and PD. The workload has an average injection rate of 5 $jobs/msec$. This workload is just a representation of a wireless scenario where different communication systems transmit and receive data simultaneously over time. Table VII contains a summary of the number of frames, the number of tasks, and the number of applications injected during this evaluation.

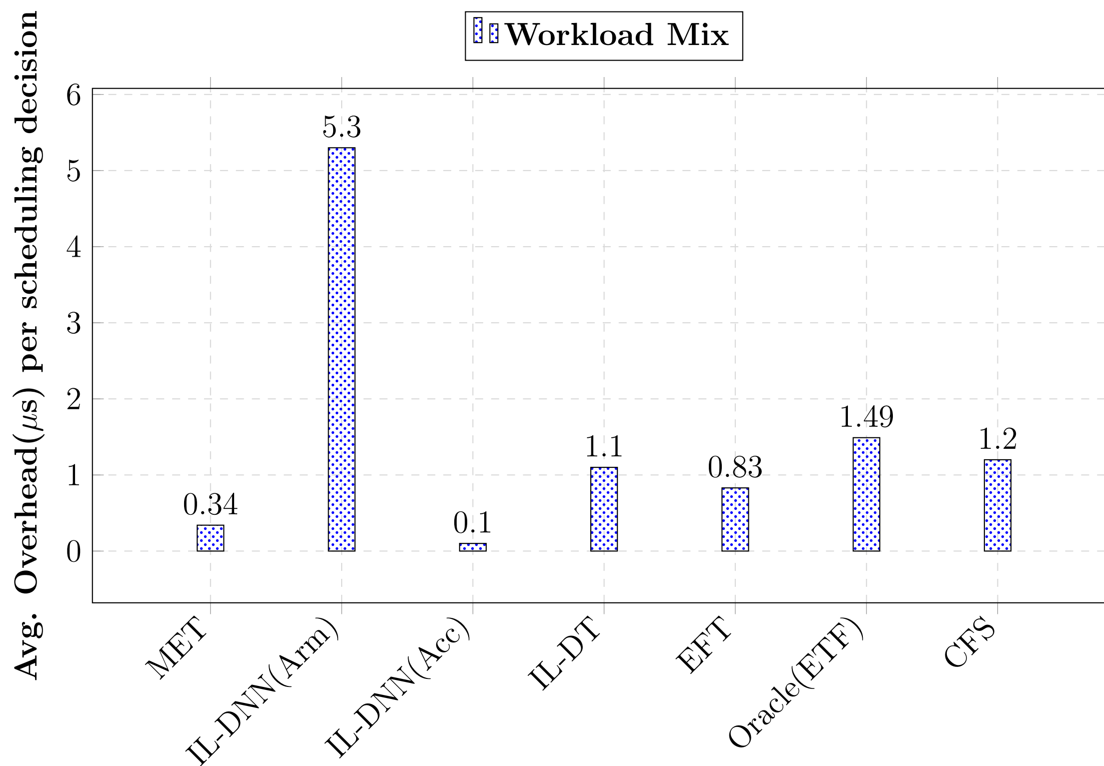


Figure 12: Overhead Study of the Different Scheduling Policies

7.3 Comparison of Scheduling Policies

Execution time is one of the most compelling features when evaluating the performance of a scheduler. Figure 13 represents the average throughput for several

scheduling policies. Each point of the graph corresponds to a different workload. Each workload varies in the number of frames injected from each application. The X-axis represents the normalized throughput that ranges from 1 job/ms to 15 $jobs/ms$. To keep a similar behavior when varying the throughput, we established an injection ratio. 4:2:1, for every four WiFi-Tx and WiFi-Rx applications, two range detection, and one pulse Doppler application are injected.

The proposed IL-DNN performs within 5% of the Oracle and the rest policies for lower throughput. In contrast, IL-DT performs better than the Oracle for all injection rates. However, these results are while computing the scheduling policy on Arm Cortex-A53. When the DNN is implemented on a custom accelerator, we estimate that IL-DNN achieves larger speed-ups than IL-DT. Therefore, although these are experimental results, the DNN running on a custom accelerator executes jobs faster than the rest of the scheduling policies.

Overall, the IL-DT and IL-DNN regularly assign accelerator-supported tasks to the accelerator when the resource is free. Pulse Doppler contains more than 200 tasks that can be accelerated. With jobs like pulse Doppler, the system must ensure accurate task-accelerated decisions. This SoC has two custom accelerators and four homogeneous cores. Figure 13 compares the execution performance of the scheduling policies to the ETF/Oracle. IL-DT performs better than the Oracle because IL-DT matches within 10% the accuracy of the Oracle, and computes a solution faster than ETF. The 10% error does not affect task-accelerated decisions. As the injection rate increases, IL-DT seems to perform better than the Oracle. IL-DT has an average job speed-up of 1.17× from the Oracle. IL-DNN(Arm) is estimated to have an average speed-up of 1.21× from the Oracle.

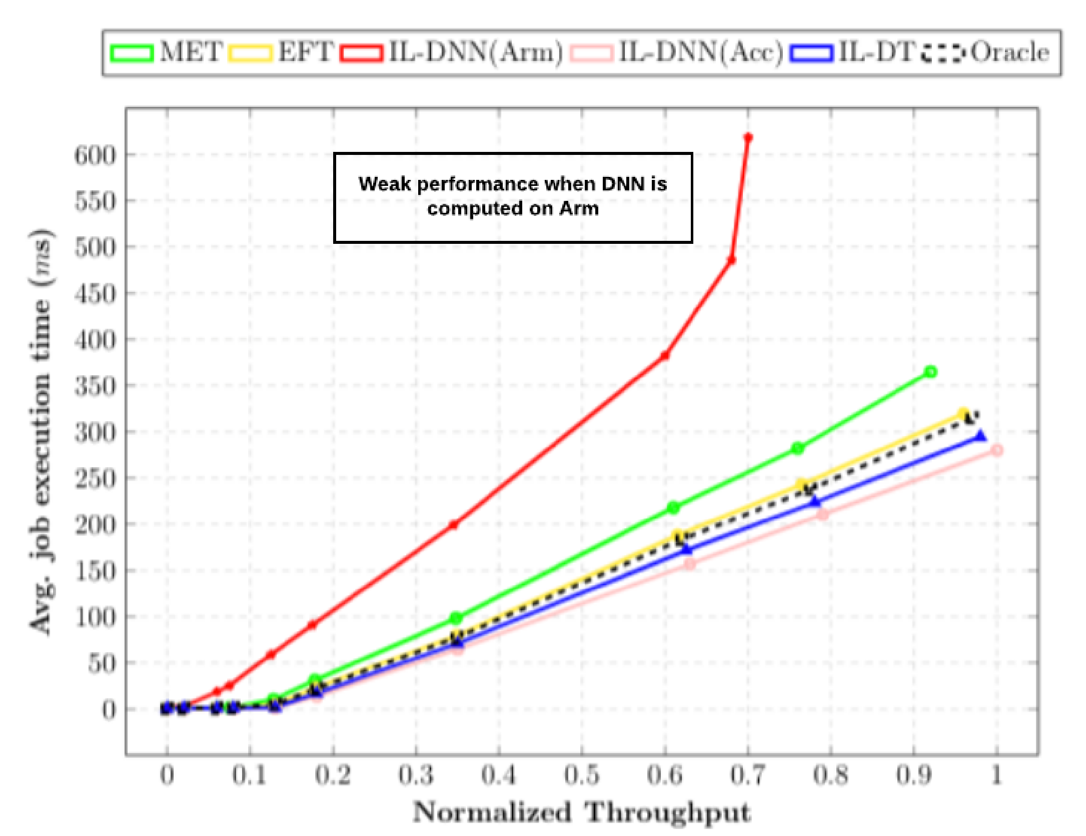


Figure 13: Comparison of the Average Job Execution Time for Varying Throughput Across Different Scheduling Policies

7.4 Algorithm Complexity Analysis

This section compares the computational complexity of the algorithm used for the Oracle/ETF with the complexity introduced by the proposed IL solutions. The function describing the performance represents an upper bound, determined by the algorithm's worst-case performance. As discussed in Chapter 3, ETF performs task re-ordering. The algorithm traverses the list of ready queues twice, and for each task, it compares the execution on each PEs. Considering n as the number of tasks inside the ready queue, and m, the number of processing elements in a platform. ETF has a

time complexity of $O(n^2m)$. On the other hand, the IL policies only need to compute - in constant time - a single output (for each of the tasks inside the ready queue) based on the feature inputs. Because of this algorithm's time complexity, the IL-DNN achieves higher speed-ups than Oracle, when the DNN inference is implemented on the accelerator. Both IL policies have a time complexity of $O(nm)$.

Chapter 8

CONCLUSION AND DIRECTION FOR FUTURE WORKS

This Chapter summarizes the work performed in this study and concludes the thesis. In this study, we explored the challenge of task scheduling and task acceleration in radio and wireless networks. Our proposed framework is capable of building, deploying, and evaluating machine learning models on embedded systems. We proposed two machine learning-based algorithms integrated with decision trees and deep neural network techniques to solve the problem. The results collected through this study show how machine learning benefits scheduling. IL policies are capable of replicating the decisions of an Oracle with more than 97% accuracy. Results demonstrate how the IL-DT performs faster than the Oracle when computed on a core. In contrast, IL-DNN occupies less memory space and has the potential to perform better than the other policies when computed on a custom accelerator [76], a trade-off between latency and memory. The proposed IL-based scheduling approach provides significant potential for future research in this space. For instance, as applications and computing platforms evolve rapidly, a methodology to update the weights of the machine learning model online is an interesting topic of future study. This methodology will enable the design of a lifelong-learning scheduler.

Several studies, tests, and evaluations can be addressed by future work. For example, scheduling policies could be dynamically switched, if we determine ranges of operations where other scheduling policies outperform the rest; math-intensive and floating-point operations are computed faster in customized resources rather than in basic processing units. We are also planning to integrate the hardware accelerators that executes the scheduling policy with the user-space scheduling framework. Finally,

The idea of a generic approach where custom fixed-point accelerators help improve the performance and reduce the scheduling policy overhead is perhaps the way to make machine learning in scheduling a reality.

REFERENCES

[1] AI Transformer. [Online] http://www.aitransformer.com/, accessed 8 Jul. 2020.

[2] Linux processes, threads and LWPs. [Online] https://www.thegeekstuff.com/2013/11/linux-process-and-threads/, accessed 7 Feb 2020 , author=Himanshu Arora, year=2013.

[3] Process Creation in Linux. [Online] https://www.tldp.org/LDP/tlk/kernel/processes.html, accessed 12 July 2020, author=David A Rusling, year=1999.

[4] ZCU102 Evaluation Board. https://www.xilinx.com/support/documentation/boards_and_kits/zcu102/ug1182-zcu102-eval-bd.pdf, Accessed 3 Mar. 2019.

[5] D. M. Abdelkader and F. Omara. Dynamic task scheduling algorithm with load balancing for heterogeneous computing system. *Egyptian Informatics Journal*, 13(2):135–145, 2012.

[6] I. Adler, M. G. Resende, G. Veiga, and N. Karmarkar. An implementation of karmarkar's algorithm for linear programming. *Mathematical programming*, 44(1-3):297–335, 1989.

[7] A. F. Agarap. Deep learning using rectified linear units (relu). *arXiv preprint arXiv:1803.08375*, 2018.

[8] Z. H. Ahmed. Genetic algorithm for the traveling salesman problem using sequential constructive crossover operator. *International Journal of Biometrics & Bioinformatics (IJBB)*, 3(6):96, 2010.

[9] Ankita Garg. Real Time Kernel Scheduler, 2009. [Online] https://www.linuxjournal.com/article/10165, accessed 15 Aug. 2019.

[10] H. Arabnejad and J. G. Barbosa. List Scheduling Algorithm for Heterogeneous Systems by an Optimistic Cost Table. *IEEE Trans. on Parallel and Distributed Systems*, 25(3):682–694, 2013.

[11] S. E. Arda, A. Krishnakumar, A. A. Goksoy, N. Kumbhare, J. Mack, A. L. Sartor, A. Akoglu, R. Marculescu, and U. Y. Ogras. Ds3: A system-level domain-specific system-on-chip simulation framework. *IEEE Transactions on Computers*, 69(8):1248–1262, 2020.

[12] O. Arnold and G. Fettweis. On the impact of dynamic task scheduling in heterogeneous mpsocs. In *2011 International Conference on Embedded Computer Systems: Architectures, Modeling and Simulation*, pages 17–24. IEEE, 2011.

[13] C. Augonnet, S. Thibault, R. Namyst, and P.-A. Wacrenier. Starpu: a unified platform for task scheduling on heterogeneous multicore architectures. *Concurrency and Computation: Practice and Experience*, 23(2):187–198, 2011.

[14] P. Baptiste, C. Le Pape, and W. Nuijten. *Constraint-based scheduling: applying constraint programming to scheduling problems*, volume 39. 2012.

[15] S. Baskiyar and K. K. Palli. Low Power Scheduling of DAGs to Minimize Finish Times. In Y. Robert, M. Parashar, R. Badrinath, and V. K. Prasanna, editors, *High Performance Computing - HiPC 2006*, Lecture Notes in Computer Science, pages 353–362. Springer Berlin Heidelberg, 2006.

[16] T. Beisel, T. Wiersema, C. Plessl, and A. Brinkmann. Cooperative multitasking for heterogeneous accelerators in the linux completely fair scheduler. In *ASAP 2011-22nd IEEE International Conference on Application-specific Systems, Architectures and Processors*, pages 223–226. IEEE, 2011.

[17] T. Beisel, T. Wiersema, C. Plessl, and A. Brinkmann. Programming and scheduling model for supporting heterogeneous accelerators in linux. In *Workshop on Computer Architecture and Operating System Co-design (CAOS)*, 2012.

[18] L. Benini and G. DeMicheli. *Dynamic power management: design techniques and CAD tools*. Springer Science & Business Media, 2012.

[19] G. Bhat, G. Singla, A. K. Unver, and U. Y. Ogras. Algorithmic optimization of thermal and power management for heterogeneous mobile platforms. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, 26(3):544–557, 2017.

[20] N. Binkert, B. Beckmann, G. Black, S. K. Reinhardt, A. Saidi, A. Basu, J. Hestness, D. R. Hower, T. Krishna, S. Sardashti, et al. The gem5 simulator. *ACM SIGARCH computer architecture news*, 39(2):1–7, 2011.

[21] L. F. Bittencourt, R. Sakellariou, and E. R. Madeira. DAG Scheduling Using a Lookahead Variant of the Heterogeneous Earliest Finish Time Algorithm. In *Euromicro Conference on Parallel, Distributed and Network-based Processing*, pages 27–34, 2010.

[22] L. F. Bittencourt, R. Sakellariou, and E. R. M. Madeira. DAG scheduling using a lookahead variant of the heterogeneous earliest finish time algorithm. In *2010 18th Euromicro Conference on Parallel, Distributed and Network-based Processing*, pages 27–34, Feb 2010.

[23] D. W. Bliss and S. Govindasamy. *Adaptive Wireless Communications: MIMO Channels and Networks*. Cambridge University Press, 2013.

[24] P. Bogdan, F. Chen, A. Deshwal, J. R. Doppa, B. K. Joardar, H. Li, S. Nazarian, L. Song, and Y. Xiao. Taming extreme heterogeneity via machine learning based design of autonomous manycore systems. In *Proceedings of the International Conference on Hardware/Software Codesign and System Synthesis Companion*, pages 1–10, 2019.

[25] P. Bogdan, R. Marculescu, and S. Jain. Dynamic power management for multidomain system-on-chip platforms: An optimal control approach. *ACM Transactions on Design Automation of Electronic Systems (TODAES)*, 18(4):1–20, 2013.

[26] D. P. Bovet and M. Cesati. *Understanding the Linux Kernel: from I/O ports to process management.* 2005.

[27] T. D. Braun, H. J. Siegel, N. Beck, L. L. Bölöni, M. Maheswaran, A. I. Reuther, J. P. Robertson, M. D. Theys, B. Yao, D. Hensgen, et al. A comparison of eleven static heuristics for mapping a class of independent tasks onto heterogeneous distributed computing systems. *Journal of Parallel and Distributed computing*, 61(6):810–837, 2001.

[28] J. M. Bull, L. A. Smith, C. Ball, L. Pottage, and R. Freeman. Benchmarking java against c and fortran for scientific applications. *Concurrency and Computation: Practice and Experience*, 15(3-5):417–430, 2003.

[29] A. R. Chiriyath, B. Paul, and D. W. Bliss. Radar-communications convergence: Coexistence, cooperation, and co-design. *IEEE Transactions on Cognitive Communications and Networking*, 3(1):1–12, 2017.

[30] S. Choudhury, A. Kapoor, G. Ranade, and D. Dey. Learning to gather information via imitation. In *2017 IEEE International Conference on Robotics and Automation (ICRA)*, pages 908–915. IEEE, 2017.

[31] A. K. Coskun, R. Strong, D. M. Tullsen, and T. Simunic Rosing. Evaluating the impact of job scheduling and power management on processor lifetime for chip multiprocessors. In *Proceedings of the eleventh international joint conference on Measurement and modeling of computer systems*, pages 169–180, 2009.

[32] B. Donyanavard, T. Mück, S. Sarma, and N. Dutt. SPARTA: Runtime task allocation for energy efficient heterogeneous manycores. In *2016 International Conference on Hardware/Software Codesign and System Synthesis (CODES+ISSS)*, pages 1–10. IEEE, 2016.

[33] J. Du and J. Y.-T. Leung. Complexity of scheduling parallel task systems. *SIAM Journal on Discrete Mathematics*, 2(4):473–487, 1989.

[34] P. Flach. *Machine learning: the art and science of algorithms that make sense of data.* 2012.

[35] D. Green et al. Heterogeneous Integration at DARPA: Pathfinding and Progress in Assembly Approaches. *ECTC, May*, 2018.

[36] A. Gulli and S. Pal. *Deep learning with Keras.* 2017.

[37] U. Gupta, R. Ayoub, M. Kishinevsky, D. Kadjo, N. Soundararajan, U. Tursun, and U. Y. Ogras. Dynamic power budgeting for mobile systems running graphics workloads. *IEEE Transactions on Multi-Scale Computing Systems*, 4(1):30–40, 2017.

[38] U. Gupta, S. K. Mandal, M. Mao, C. Chakrabarti, and U. Y. Ogras. A deep q-learning approach for dynamic management of heterogeneous processors. *IEEE Computer Architecture Letters*, 18(1):14–17, 2019.

[39] M. Hall, E. Frank, G. Holmes, B. Pfahringer, P. Reutemann, and I. H. Witten. The weka data mining software: an update. *ACM SIGKDD explorations newsletter*, 11(1):10–18, 2009.

[40] H. P. Hofstee. Power efficient processor architecture and the cell processor. In *11th International Symposium on High-Performance Computer Architecture*, pages 258–262. IEEE, 2005.

[41] C. Hsieh, A. A. Sani, and N. Dutt. SURF: Self-aware Unified Runtime Framework for Parallel Programs on Heterogeneous Mobile Architectures. In *2019 IFIP/IEEE 27th International Conference on Very Large Scale Integration (VLSI-SoC)*, pages 136–141, 2019.

[42] J. Hu and R. Marculescu. Communication and task scheduling of application-specific networks-on-chip. *IEE Proceedings-Computers and Digital Techniques*, 152(5):643–651, 2005.

[43] J. Hu and R. Marculescu. Energy-and performance-aware mapping for regular noc architectures. *IEEE Transactions on computer-aided design of integrated circuits and systems*, 24(4):551–562, 2005.

[44] J.-J. Hwang, Y.-C. Chow, F. D. Anger, and C.-Y. Lee. Scheduling precedence graphs in systems with interprocessor communication times. *SIAM Journal on Computing*, 18(2):244–257, 1989.

[45] Jack Park. Iterative Dichotomiser 3, 2017. [Online] `https://golden.com/wiki/Iterative_Dichotomiser_3`, accessed 5 Feb. 2020.

[46] A. Jantsch, N. Dutt, and A. M. Rahmani. Self-awareness in systems on chip—a survey. *IEEE Design & Test*, 34(6):8–26, 2017.

[47] Josh R. Weka to C, 2011. [Online] `https://github.com/raddy/weka2c`, accessed 7 Feb. 2020.

[48] Kawazome Ichiro. User DMA Buffer, 2016. [Online] `https://github.com/ikwzm/udmabuf`, accessed 5 Nov. 2019.

[49] A. S. Khot and R. K. Mishra. *Learning Functional Data Structures and Algorithms*. Packt Publishing Ltd, 2017.

[50] R. G. Kim et al. Imitation Learning for Dynamic VFI Control in Large-Scale Manycore Systems. *IEEE Trans. on Very Large Scale Integration (VLSI) Systems*, 25(9):2458–2471, 2017.

[51] J. Kobus and R. Szklarski. Completely fair scheduler and its tuning. *draft on Internet*, 2009.

[52] R. Kohavi and J. R. Quinlan. Data mining tasks and methods: Classification: decision-tree discovery. In *Handbook of data mining and knowledge discovery*, pages 267–276. 2002.

[53] A. Krishnakumar, S. E. Arda, A. A. Goksoy, S. K. Mandal, U. Y. Ogras, A. L. Sartor, and R. Marculescu. Runtime task scheduling using imitation learning for heterogeneous many-core systems. *arXiv preprint arXiv:2007.09361*, 2020.

[54] R. Kumar, K. I. Farkas, N. P. Jouppi, P. Ranganathan, and D. M. Tullsen. Single-isa heterogeneous multi-core architectures: The potential for processor power reduction. In *Proceedings. 36th Annual IEEE/ACM International Symposium on Microarchitecture, 2003. MICRO-36.*, pages 81–92. IEEE, 2003.

[55] Y.-K. Kwok and I. Ahmad. Dynamic critical-path scheduling: An effective technique for allocating task graphs to multiprocessors. *IEEE Trans. on Parallel and Distributed Systems*, 7(5):506–521, 1996.

[56] H. Leontyev and J. H. Anderson. Tardiness bounds for fifo scheduling on multiprocessors. In *19th Euromicro Conference on Real-Time Systems (ECRTS'07)*, pages 71–71. IEEE, 2007.

[57] J. Li, X. Ma, K. Singh, M. Schulz, B. R. de Supinski, and S. A. McKee. Machine learning based online performance prediction for runtime parallelization and task scheduling. In *2009 IEEE international symposium on performance analysis of systems and software*, pages 89–100. IEEE, 2009.

[58] W. Liu, Z. Wang, X. Liu, N. Zeng, Y. Liu, and F. E. Alsaadi. A survey of deep neural network architectures and their applications. *Neurocomputing*, 234:11–26, 2017.

[59] W. Liu, Y. Wen, Z. Yu, and M. Yang. Large-margin softmax loss for convolutional neural networks. In *ICML*, volume 2, page 7, 2016.

[60] R. Love. *Linux kernel development.* 2010.

[61] J. Mack, N. Kumbhare, U. Y. Ogras, and A. Akoglu. User-space emulation framework for domain-specific soc design. *arXiv preprint arXiv:2004.01636*, 2020.

[62] P. R. Mahapatra and D. S. Zrnic. Practical algorithms for mean velocity estimation in pulse doppler weather radars using a small number of samples. *IEEE transactions on geoscience and remote sensing*, (4):491–501, 1983.

[63] G. Malewicz, A. L. Rosenberg, and M. Yurkewych. Toward a theory for scheduling dags in internet-based computing. *IEEE Transactions on Computers*, 55(6):757–768, 2006.

[64] S. K. Mandal, G. Bhat, J. R. Doppa, P. P. Pande, and U. Y. Ogras. An energy-aware online learning framework for resource management in heterogeneous platforms. *ACM Transactions on Design Automation of Electronic Systems (TODAES)*, 25(3):1–26, 2020.

[65] S. K. Mandal, G. Bhat, C. A. Patil, J. R. Doppa, P. P. Pande, and U. Y. Ogras. Dynamic resource management of heterogeneous mobile platforms via imitation learning. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, 27(12):2842–2854, 2019.

[66] H. Mao, M. Alizadeh, I. Menache, and S. Kandula. Resource Management with Deep Reinforcement Learning. In *ACM Workshop on Hot Topics in Networks*, pages 50–56, 2016.

[67] H. Mao, M. Schwarzkopf, S. B. Venkatakrishnan, Z. Meng, and M. Alizadeh. Learning Scheduling Algorithms for Data Processing Clusters. In *ACM Special Interest Group on Data Communication*, pages 270–288. 2019.

[68] Mark Twain. Monolithic Kernels, 2020. [Online] `https://hackinbits.com/articles/linux-kernel-explained-i`, accessed 2 Feb. 2020.

[69] Matthew Reilly. Trends in Multicore, 2020. [Online] `https://archive.ll.mit.edu/HPEC/agendas/proc08/Day1/16-Day1-Session2-Reilly-abstract.pdf`, accessed 10 Jan. 2020.

[70] W. Mauerer. *Linux® Kernel Architecture*. 2008.

[71] M. Minsky and S. A. Papert. *Perceptrons: An introduction to computational geometry*. MIT press, 2017.

[72] M. S. Mollison and J. H. Anderson. Bringing theory into practice: A userspace library for multicore real-time scheduling. In *2013 IEEE 19th Real-Time and Embedded Technology and Applications Symposium (RTAS)*, pages 283–292. IEEE, 2013.

[73] M. Nabelsee, A. Busse, H. Parzyjegla, and G. Mühl. Load-aware scheduling for heterogeneous multi-core systems. In *Proceedings of the 31st Annual ACM Symposium on Applied Computing*, pages 1844–1851, 2016.

[74] U. Y. Ogras, R. Marculescu, and D. Marculescu. Variation-adaptive feedback control for networks-on-chip with multiple clock domains. In *2008 45th ACM/IEEE Design Automation Conference*, pages 614–619, 2008.

[75] U. Y. Ogras, R. Marculescu, D. Marculescu, and E. G. Jung. Design and management of voltage-frequency island partitioned networks-on-chip. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, 17(3):330–341, 2009.

[76] A. Parashar, P. Raina, Y. S. Shao, Y.-H. Chen, V. A. Ying, A. Mukkara, R. Venkatesan, B. Khailany, S. W. Keckler, and J. Emer. Timeloop: A systematic approach to dnn accelerator evaluation. In *2019 IEEE international symposium on performance analysis of systems and software (ISPASS)*, pages 304–315. IEEE, 2019.

[77] S. Pasricha, R. Ayoub, M. Kishinevsky, S. K. Mandal, and U. Y. Ogras. A survey on energy management for mobile and iot devices. *IEEE Design & Test*, 2020.

[78] M. Pinedo. *Scheduling*, volume 29. 2012.

[79] J. R. Quinlan. *C4. 5: programs for machine learning*. Elsevier, 2014.

[80] A. Roca, S. Rodriguez, A. Segura, K. Marquet, and V. Beltran. A linux kernel scheduler extension for multi-core systems. In *2019 IEEE 26th International Conference on High Performance Computing, Data, and Analytics (HiPC)*, pages 353–362. IEEE, 2019.

[81] B. Roch. Monolithic kernel vs. microkernel. *TU Wien*, 2004.

[82] S. Ross, G. Gordon, and D. Bagnell. A Reduction of Imitation Learning and Structured Prediction To No-Regret Online Learning. In *Proc. of the Int. Conf. on Art. Intel. and Stat.*, pages 627–635, 2011.

[83] K. Roy, B. Jung, D. Peroulis, and A. Raghunathan. Integrated systems in the more-than-moore era: designing low-cost energy-efficient systems using heterogeneous components. *IEEE Design & Test*, 33(3):56–65, 2013.

[84] O. Sahin, L. Thiele, and A. K. Coskun. Maestro: Autonomous qos management for mobile applications under thermal constraints. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 38(8):1557–1570, 2018.

[85] R. Sakellariou and H. Zhao. A hybrid heuristic for dag scheduling on heterogeneous systems. In *Int. Parallel and Distributed Processing Symposium*, page 111. IEEE, 2004.

[86] D. Sanchez, D. Lo, R. M. Yoo, J. Sugerman, and C. Kozyrakis. Dynamic fine-grain scheduling of pipeline parallelism. In *2011 International Conference on Parallel Architectures and Compilation Techniques*, pages 22–32. IEEE, 2011.

[87] S. Sarma et al. Smartbalance: Sensing-driven linux load balancer for energy efficiency of heterogeneous mpsocs. In *2015 52nd ACM/EDAC/IEEE Design Automation Conference (DAC)*, pages 1–6. IEEE, 2015.

[88] A. L. Sartor, A. Krishnakumar, S. E. Arda, U. Y. Ogras, and R. Marculescu. Hilite: Hierarchical and lightweight imitation learning for power management of embedded socs. *IEEE Computer Architecture Letters*, 19(1):63–67, 2020.

[89] S. Schaal. Is Imitation Learning the Route To Humanoid Robots? *Trends in Cognitive Sciences*, 3(6):233–242, 1999.

[90] A. K. Singh, A. Kumar, and T. Srikanthan. Accelerating throughput-aware runtime mapping for heterogeneous mpsocs. *ACM Transactions on Design Automation of Electronic Systems (TODAES)*, 18(1):1–29, 2013.

[91] S. S. Skiena. *The algorithm design manual: Text*, volume 1. Springer Science & Business Media, 1998.

[92] J. Song, G. Xie, R. Li, and X. Chen. An Efficient Scheduling Algorithm for Energy Consumption Constrained Parallel Applications on Heterogeneous Distributed Systems. In *2017 IEEE International Symposium on Parallel and Distributed Processing with Applications and 2017 IEEE International Conference on Ubiquitous Computing and Communications (ISPA/IUCC)*, pages 32–39, Dec. 2017.

[93] H. Taud and J. Mas. Multilayer perceptron (mlp). In *Geomatic Approaches for Modeling Land Change Scenarios*, pages 451–455. Springer, 2018.

[94] R. Teodorescu and J. Torrellas. Variation-aware application scheduling and power management for chip multiprocessors. *ACM SIGARCH computer architecture news*, 36(3):363–374, 2008.

[95] Tom Rondeau. Domain-Specific System on Chip (DSSoC), 2020. [Online] `https://www.darpa.mil/program/domain-specific-system-on-chip`, `https://eri-summit.darpa.mil/docs/20180725_1100_DSSoC.pdf`, accessed 25 July 2020.

[96] H. Topcuoglu, S. Hariri, and M.-Y. Wu. Performance-Effective and Low-Complexity Task Scheduling for Heterogeneous Computing. *IEEE Trans. on Parallel and Distrib. Syst.*, 13(3):260–274, 2002.

[97] R. Uhrie, D. W. Bliss, C. Chakrabarti, U. Y. Ogras, and J. Brunhaver. Machine Understanding of Domain Computation for Domain-Specific System-on-Chips (DSSoC). In *Open Architecture/Open Business Model Net-Centric Systems and Defense Transformation 2018*, volume 11015, page 110150O. International Society for Optics and Photonics, 2019.

[98] G. Von Laszewski, L. Wang, A. J. Younge, and X. He. Power-aware scheduling of virtual machines in dvfs-enabled clusters. In *2009 IEEE International Conference on Cluster Computing and Workshops*, pages 1–10. IEEE, 2009.

[99] L. Wang and H. Sahbi. Directed Acyclic Graph Kernels for Action Recognition. In *Proceedings of the IEEE International Conference on Computer Vision*, pages 3168–3175, 2013.

[100] Weka. Weka Software, 2009. [Online] `https://waikato.github.io/weka-wiki/downloading_weka/`, accessed 4 Feb. 2020.

[101] D. B. West et al. *Introduction to graph theory*, volume 2. 1996.

[102] D. Whitley. A genetic algorithm tutorial. *Statistics and computing*, 4(2):65–85, 1994.

[103] C. Wong, I. Tan, R. Kumari, J. Lam, and W. Fun. Fairness and interactive performance of O (1) and CFS Linux kernel schedulers. In *2008 International Symposium on Information Technology*, volume 4, pages 1–8, 2008.

[104] L. Wu, G. Kaiser, D. Solomon, R. Winter, A. Boulanger, and R. Anderson. Improving efficiency and reliability of building systems using machine learning and automated online evaluation. In *2012 IEEE Long Island Systems, Applications and Technology Conference (LISAT)*, pages 1–6. IEEE, 2012.

[105] Y. Xiao, S. Nazarian, and P. Bogdan. Self-Optimizing and Self-Programming Computing Systems: A Combined Compiler, Complex Networks, and Machine Learning Approach. *IEEE Trans. on Very Large Scale Integration (VLSI) Systems*, pages 1–12, 2019.

[106] T. Xiaoyong, K. Li, Z. Zeng, and B. Veeravalli. A Novel Security-Driven Scheduling Algorithm for Precedence-Constrained Tasks in Heterogeneous Distributed Systems. *IEEE Transactions on Computers*, 60(7):1017–1029, 2010.

[107] G. Xie, G. Zeng, L. Liu, R. Li, and K. Li. Mixed Real-Time Scheduling of Multiple DAGs-based Applications on Heterogeneous Multi-core Processors. *Microprocessors and Microsystems*, 47:93–103, 2016.

[108] Xilinx. Intellectual property and cores, 2009. [Online] `https://www.xilinx.com/support/documentation/sw_manuals/xilinx11/ise_c_intellectual_property_cores.htm`, accessed 11 Dec. 2019.

[109] Xilinx. PetaLinux Software, 2020. [Online] `https://www.xilinx.com/support/download/index.html/content/xilinx/en/downloadNav/embedded-design-tools.html`, accessed 10 Dec. 2019.

[110] L. Zhang, K. Li, C. Li, and K. Li. Bi-objective workflow scheduling of the energy consumption and reliability in heterogeneous computing systems. *Information Sciences*, 379:241–256, 2017.

[111] Y. Zhang, K. Ootsu, T. Yokota, and T. Baba. Automatic thread decomposition for pipelined multithreading. In *2010 IEEE 16th International Conference on Parallel and Distributed Systems*, pages 91–98. IEEE, 2010.

[112] N. Zhou, D. Qi, X. Wang, Z. Zheng, and W. Lin. A list scheduling algorithm for heterogeneous systems based on a critical node cost table and pessimistic cost table. *Concurrency and Computation: Practice and Experience*, 29(5):e3944, 2017.

APPENDIX A

EXCLUDING PULSE DOPPLER FROM EVALUATIONS

This section excludes the application pulse Doppler from the evaluations reducing the complexity of the benchmark. Figure 14 shows how by excluding pulse Doppler from evaluations, there is a reduction in the policy overhead. Due to the amount of FFT-alike tasks that the PD contains, we developed an optimization that reduced the IL policies overhead. This optimization was based on change detection. Hence the array of inputs from the current scheduling decision is compared with the previous one. If there is no change, we will use the same predicted PE, avoiding running the deep neural network again and thus shortening the latency. Because we are excluding PD from the evaluations, there are fewer changes detected, leading to higher latency.

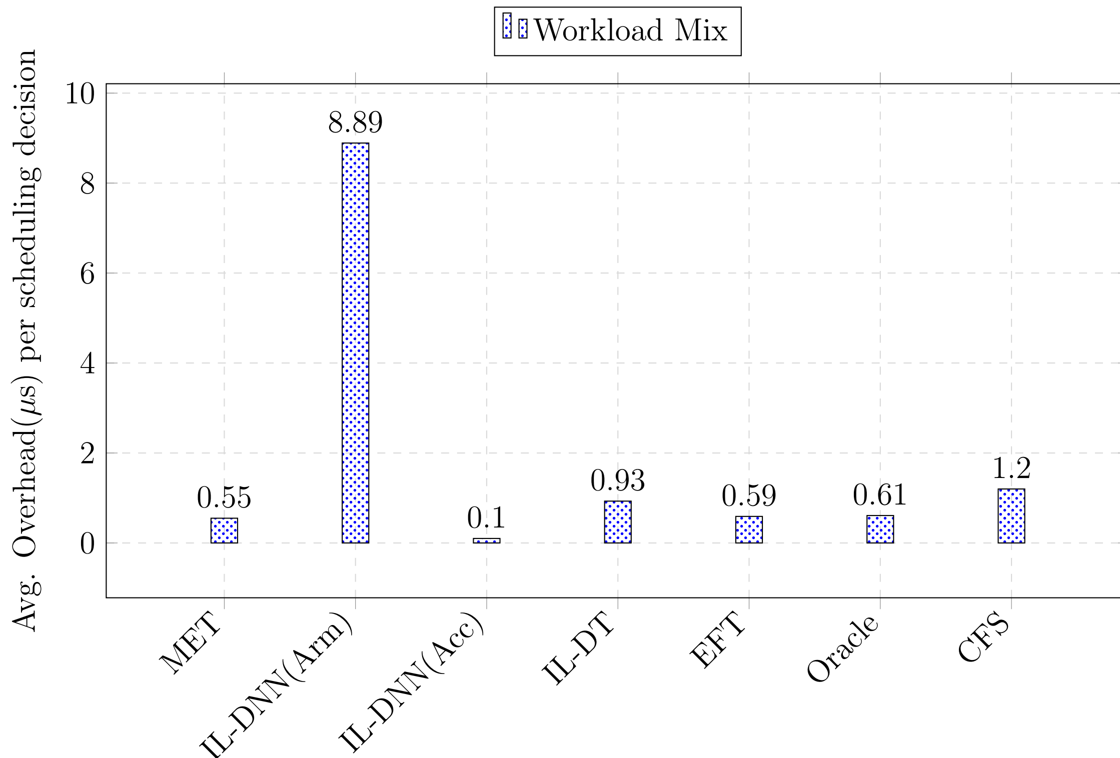


Figure 14: Overhead Study of the Different Scheduling Policies Excluding Pulse Doppler Application

In Figure 15, we observe how IL-DNN(Arm) still performs poorly compared to the rest of the scheduling policies. However, MET, IL-DT, and EFT seem to perform close to the Oracle, although IL-DT, like previously, outperforms the rest of the scheduling policies across different injection rates achieving an average job speedup of 1.07× from the Oracle. The IL-DNN(Acc) achieves a speed up of 1.11× from the Oracle.

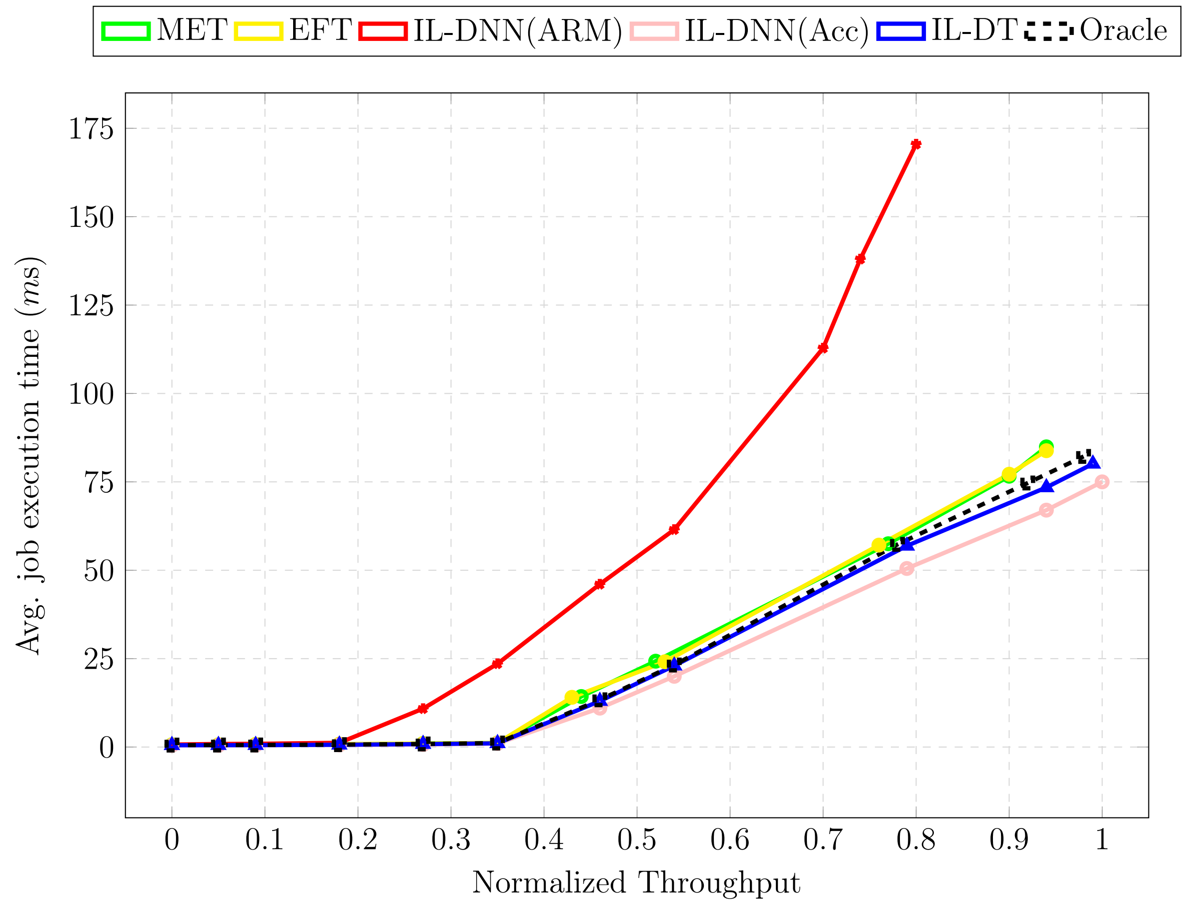


Figure 15: Comparison of the Average Job Execution Time for Varying Throughput Across Different Scheduling Policies