Design and Run-Time Resource Management of Domain-Specific Systems-on-Chip (DSSoCs)

by

Anish Nallamur Krishnakumar

A preliminary report for the degree of

Doctor of Philosophy

(Department of Electrical and Computer Engineering)

at the

UNIVERSITY OF WISCONSIN–MADISON

01/20/2022

Preliminary Examination Date: 01/20/2022

Preliminary Exam Committee:

Umit Y. Ogras, Associate Professor, Electrical and Computer Engineering, University of Wisconsin-Madison
Yu Hen Hu, Professor, Electrical and Computer Engineering, University of Wisconsin-Madison
Younghyun Kim, Professor, Computer Sciences Engineering, University of Wisconsin-Madison
Chaitali Chakrabarti, Professor, Electrical Engineering, Arizona State University

CONTENTS

Contents

ABSTRACT

The saturation of Moore's Law has stalled the improvement in performance and energy efficiency obtained with conventional homogeneous processors over technology nodes. Homogeneous processors are also not able to cater to the contrasting performance and energy requirements from different applications, leading to the rise of heterogeneous computing architectures. While heterogeneous processors provide programming flexibility, there is still a steep performance and energy-efficiency gap when compared to special-purpose solutions (for example: GPUs, DSPs and hardware accelerators). However, combining all kinds of processing elements in a single chip leads to a severe penalty in design cost, chip area and poor utilization at run-time. To address all the above challenges, domain-specific judiciously combine processing elements such general-purpose cores, special-purpose cores and hardware accelerators to maximize the energy efficiency of applications in a particular domain, and also provide programming flexibility to execute applications from other domains. The major challenge in DSSoCs is to optimally utilize these processing elements at run-time to exploit the potential of the diverse and energy-efficient compute elements on chip. Mapping tasks to the processing elements (task scheduling) and controlling their voltage and frequencies form two key aspects of resource management in DSSoCs.

To this end, we propose an imitation learning scheduler that approximates the performance of optimal/near-optimal schedulers with negligible run-time overheads, and an imitation learning based power management policy to determine the optimal voltage and frequency levels at run-time while also satisfying soft deadline requirements. Our imitation learning based scheduling policy achieves performance that is within 1% of an Oracle for multiple optimization objectives using a decision tree classifier. Furthermore, we optimize the execution of a decision tree classifier in both

software and hardware to achieve latencies of less than 50 nanoseconds for decision trees of up to depth 12. Finally, evaluating the performance and functional correctness of a complex DSSoC with such diverse processing elements is critical to avoid the exorbitant costs of post-silicon failures and bugs. Therefore, we design a FPGA based emulation framework that integrates Arm cores, hardware accelerators, caches and interconnects to validate the functionality and performance of the DSSoC, perform rapid and realistic design space exploration, evaluate scheduling algorithms and enable early software, firmware and driver development.

1 INTRODUCTION

The improvements in integrated circuit performance, power and energy-efficiency over process nodes have significantly slowed down, thereby indicating the end of Moore's Law [95, 36]. In addition, the end of Dennard scaling has failed our estimates of power and performance from the next generation circuits [31]. While these two aspects have been instrumental in improving energy efficiency over the years, they are no more sufficient to obtain the expected gains. Although circuits experienced an increase in the operating frequencies to compensate for the gap in performance, the cubic dependency of power with the frequency increased the power consumption to an extent that heat dissipation became a critical challenge, a phenomenon popularly referred to as the "Power Wall" [37, 103]. Chip architects then turned towards instruction-level parallelism techniques – such as processor pipelining, prefetching, branch prediction, superscalar execution, out-of-order execution – in the microarchitecture of general-purpose processors under power and temperature constraints [44]. However, all these techniques saturated and provided only marginal benefits in performance, thereby leaving a substantial scope for improvement.

The rise of multicore architectures effectively circumvented the power wall by integrating multiple identical cores onto the same die to provide higher computational power under similar area budgets [33, 43]. While marginal increase in frequencies led to significant increase in area and power, multiple cores on chip provided ideally double the computation power at moderate frequencies, thereby allowing for drastic improvements in energy-efficiency [34]. Processors began to experience contrasting application requirements as they were exploited to execute multiple applications simultaneously. For instance, on the one hand, internet browsing and e-mail applications require low computational power. On the other hand,

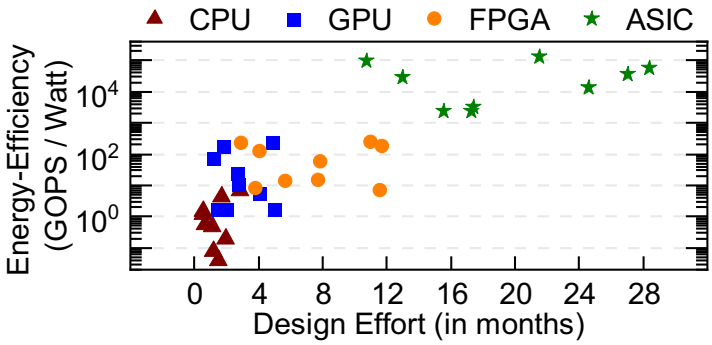


Figure 1.1: Trends in energy-efficiency and design effort of CPU, GPU, FPGA and special-purpose ASIC implementations in literature.

video games, augmented/virtual reality applications and multimedia applications demand tremendous computational power to perform wireless communications, video and audio processing, and other computations simultaneously. Homogeneous multicore processors tradeoff performance with power consumption, and vice-versa, and hence cannot satisfy both requirements together. To this end, heterogeneous multiprocessor architectures addressed this problem by integrating low-power energy-efficient cores and high-performance cores [73, 39]. The heterogeneous cores cater to the contrasting requirements of applications, i.e., the energy-efficient cores service the simple applications such as email and internet browsing, and the high-performance cores provide the computational power required for multimedia-like applications. While heterogeneous architectures were primarily introduced in embedded system processors, they are extensively used in most processing systems such as mobile phones, laptops, desktops and even servers[12, 39, 78, 84].

Heterogeneous multiprocessor systems-on-chip (MPSoCs) solved several challenges to improve performance and energy-efficiency over prior techniques and computing architectures. However, they still suffered a substantial gap with respect to special-purpose solutions. For example,

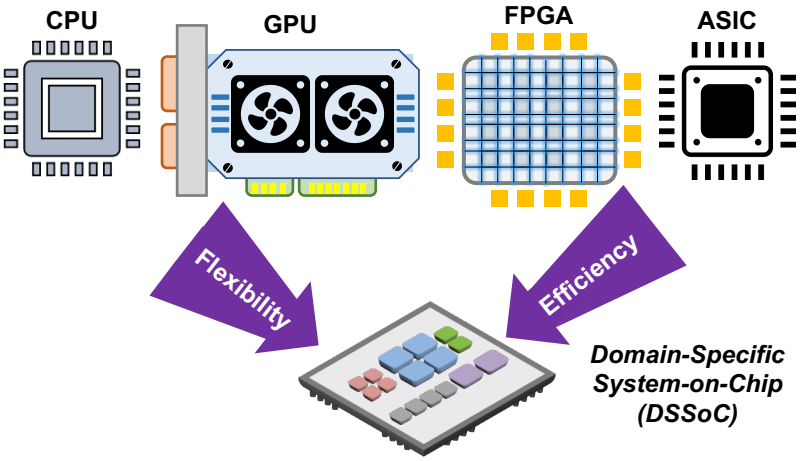


Figure 1.2: DSSoCs combine the programmability and flexibility benefits of CPUs and GPUs, and the energy-efficiency of FPGAs and special-purpose ASICs.

a special-purpose ASIC for software-defined radio applications such as 3GPP-LTE achieves 216× better area-time-energy product when compared to an equivalent RISC-V processor implementation [70]. However, the energy-efficiency of specialized solutions is achieved at the cost of high design time. Indeed, Figure 1.2(a) shows the energy-efficiency of CPU, GPU, FPGA and ASIC implementations of several applications in literature. CPU implementations requiring the least design effort, also provide the lowest energy efficiency. GPUs and FPGAs improve the energy-efficiency by exploiting the benefits of single-instruction multiple data (SIMD) execution and parallelism benefits, respectively. Design effort for GPUs comprise converting the applications into GPU-compatible code, and FPGAs involve hardware descriptions (and/or) high-level synthesis. ASICs provide the highest energy efficiency since they are specifically designed for the target application and enjoy the energy efficiency benefits that hardware implementations provide. However, the effort for ASICs that include design, development, fabrication, software development and bringup could run

into several months–years worth of time.

General-purpose multiprocessor systems provide ease of flexibility and programmability at the cost of energy-efficiency, and special-purpose ASICs provide superior energy-efficiency with substantially higher design effort. Domain-specific systems-on-chip (DSSoCs), an instance of heterogeneous computing architectures, exploit the best of both worlds by integrating general-purpose, special-purpose and hardware accelerator cores on a single die, thereby providing programmability through the general-purpose cores and energy-efficiency through the special purpose cores and hardware accelerators (shown in Figure 1.2). The architecture is domain-specific because the processing elements are judiciously selected in such a way that frequently occurring kernels in a particular domain can be accelerated using the specialized cores. On the one hand, integrating too many accelerators increases the design time, design cost, chip complexity, die area, power and energy consumption. On the other hand, including too few accelerators may force a majority of the kernels to still be executed in the general-purpose cores, thereby degrading the energy-efficiency and the purpose of DSSoCs. For a given target domain, DSSoCs can provide three orders of magnitude higher energy-efficiency in comparison to general-purpose processors [25].

Harvesting the full potential of DSSoCs depends critically on the integration of optimal combination of computing resources and their effective utilization and management at runtime. Hence, the first step in the design flow includes analysis of the domain applications to identify the commonly used kernels [98]. This analysis aids in determining the set of specialized hardware accelerators for the target applications. For example, DSSoCs targeting wireless communication applications obtain better performance with the inclusion of Fast-Fourier Transform (FFT) accelerators. Similarly, SoCs optimized for autonomous driving applications integrate deep neural network (DNN) accelerators [55]. Then, a wide range of design-

and run-time algorithms are employed to schedule the applications to the processing elements (PEs) in the DSSoC [24, 23, 90, 26]. Finally, dynamic power and thermal management (DTPM) techniques optimize the SoC for energy efficient operations at runtime. In this report, we introduce novel task scheduling algorithms to maximize the energy efficiency of DSSoCs and tools that help in both fast and effective validation of DSSoCs.

Section 1.1 presents the novel imitation learning based task scheduling algorithm for DSSoCs. Section 1.2 describes the high-level simulation framework for rapid design space exploration and evaluation of scheduling algorithms and dynamic voltage-frequency governors. The proposed work on an incremental learning algorithm for decision trees is discussed in Section 2. The need and ideas for an integrated SoC and network-on-chip voltage-frequency scaling technique is presented in Section 3. Section 4 concludes the report.

1.1 Imitation Learning based Task Scheduling for DSSoCs

1.1.1 Introduction

Homogeneous multi-core architectures have successfully exploited thread- and data-level parallelism to achieve performance and energy efficiency beyond the limits of single-core processors. While general-purpose computing achieves programming flexibility, it suffers from significant performance and energy efficiency gap when compared to special-purpose solutions. Domain-specific architectures, such as graphics processing units (GPUs) and neural network processors, are recognized as some of the most promising solutions to reduce this gap [40]. Domain-specific systems-on-chip (DSSoCs), a concrete instance of this new architecture, judiciously combine general-purpose cores, special-purpose processors, and hardware accelerators. DSSoCs approach the efficacy of fixed-function solutions for a specific domain while maintaining programming flexibility for other domains [36].

The success of DSSoCs depends critically on satisfying two intertwined requirements. First, the available processing elements (PEs) must be utilized optimally, at runtime, to execute the incoming tasks. For instance, scheduling all tasks to general-purpose cores may work, but diminishes the benefits of the special-purpose PEs. Likewise, a static task-to-PE mapping could unnecessarily stall the parallel instances of the same task. Second, acceleration of the domain-specific applications needs to be oblivious to the application developers to make DSSoCs practical. This work addresses these two requirements simultaneously.

The task scheduling problem involves assigning tasks to processing elements and ordering their execution to achieve the optimization goals, e.g., minimizing execution time, power dissipation, or energy consumption. To

this end, applications are abstracted using mathematical models, such as directed acyclic graph (DAG) and synchronous data graphs (SDG) that capture both the attributes of individual tasks (e.g., expected execution time) and the dependencies among the tasks [96, 17, 85]. Scheduling these tasks to the available PEs is a well-known NP-complete problem [32, 99]. An optimal *static schedule* can be found for small problem sizes using optimization techniques, such as mixed-integer programming (MIP) [35] and constraint programming (CP) [83]. These approaches are not applicable to runtime scheduling for two fundamental reasons. First, statically computed schedules lose relevance in a dynamic environment where tasks from multiple applications stream in parallel, and PE utilizations change dynamically. Second, the execution time of these algorithms, hence their overhead, can be prohibitive even for small problem sizes with few tens of tasks. Therefore, a variety of heuristic schedulers, such as shortest job first (SJF) [101] and complete fair schedulers (CFS) [72], are used in practice for homogeneous systems. These algorithms trade off the quality of scheduling decisions and computational overhead.

To improve this state of affairs, this work addresses the following challenging proposition: *Can we achieve a scheduler performance close to that of optimal MIP and CP schedulers, while using minimal runtime overhead compared to commonly used heuristics?* Furthermore, we investigate this problem in the context of heterogeneous PEs. We note that much of the scheduling in heterogeneous many-core systems is tuned manually, even to date [11]. For example, OpenCL, a widely-used programming model for heterogeneous cores, leaves the scheduling problem to the programmers. Experts manually optimize the task to resource mapping based on their knowledge of application(s), characteristics of the heterogeneous clusters, data transfer costs, and platform architecture. However, manual optimization suffers from scalability for two reasons. First, optimizations do not scale well for all applications. Second, extensive engineering efforts are required

to adapt the solutions to different platform architectures and varying levels of concurrency in applications. Hence, there is a critical need for a methodology to provide optimized scheduling solutions applicable to a variety of applications at runtime in heterogeneous many-core systems.

Scheduling has traditionally been considered as an optimization problem [35]. We change this perspective by formulating runtime scheduling for heterogeneous many-core platforms as a classification problem. This perspective and the following *key insights* enable us to employ machine learning (ML) techniques to solve this problem:

Key insight 1: One can use an optimal (or near-optimal) scheduling algorithm offline without being limited by computational time and other runtime overheads. Then, the inputs to this scheduler and its decisions can be recorded along with relevant features to construct an Oracle.

Key insight 2: One can design a policy that approximates the Oracle with minimum overhead and use this policy at runtime.

Key insight 3: One can exploit the effectiveness of ML to learn from Oracle with different objectives, which includes minimizing execution time, energy consumption, etc.

Realizing this vision requires addressing several challenges. First, we need to construct an Oracle in a dynamic environment where tasks from multiple applications can overlap arbitrarily, and each incoming application instance observes a different system state. Finding optimal schedules is challenging even offline, since the underlying problem is NP-complete. We address this challenge by constructing Oracles using both CP and a computationally expensive heuristic, called earliest task first (ETF) [42]. ML uses informative properties of the system (*features*) to predict the category in a classification problem. The second challenge is identifying the minimal set of relevant features that can lead to high accuracy with minimal overhead. We store a small set of 45 relevant features for a many-core platform with 16 processing elements along with the Oracle to minimize

the runtime overhead. This enables us to represent a complex scheduling decision as a set of features and then predict the best processing element for task execution. The final challenge is approximating the Oracle accurately with a minimum implementation overhead. Since runtime task scheduling is a sequential decision-making problem, supervised learning methodologies, such as linear regression and decision tree, may not generalize for unseen states at runtime. Reinforcement learning (RL) and imitation learning (IL) are more effective for sequential decision-making problems [92, 59, 87]. Indeed, RL has shown promise when applied to the scheduling problem [63, 64, 104], but it suffers from slow convergence and sensitivity of the reward function [49, 58]. In contrast, IL takes advantage of the expert's inherent knowledge and produces policies that imitate the expert decisions [88]. Hence, we propose an IL-based framework that schedules incoming applications to heterogeneous multi-core systems.

The proposed IL framework is formulated to facilitate generalization, i.e. it can be adapted to learn from any Oracle that optimizes a specific objective, such as performance and energy efficiency, of an arbitrary DSSoC. We evaluate the proposed framework with six domain-specific applications from wireless communications and radar systems. The proposed IL policies successfully approximate the Oracle with more than 99% accuracy, achieving fast convergence and generalizing to unseen applications. In addition, the scheduling decisions are made within 1.1µs (on an Arm A53 core), which is better than CFS performance (1.2µs). To the best of our knowledge, this is the first imitation learning-based scheduling framework for heterogeneous many-core systems capable of handling multiple applications exhibiting streaming behavior. The main contributions of this section are as follows:

- An imitation learning framework to construct policies for task scheduling in heterogeneous many-core platforms;
- Oracle design using both optimal and heuristic schedulers for performance-

and energy- based optimization objectives;

- Extensive experimental evaluation of the proposed IL policies along with latency and storage overhead analysis;
- Performance comparison of IL policies against reinforcement learning and optimal schedules obtained by constraint programming.

The rest of this work is organized as follows. We review the related work in Section 1.1.2. Section 1.1.3 provides background information on DAG scheduling and imitation learning. In Section 1.1.4, we discuss the proposed methodology, followed by relevant experimental results in Section 1.1.6. Section 1.1.7 presents the conclusions and possible future research for this work.

1.1.2 Related Work and Novel Contributions

Current many-core systems use runtime heuristics to enable scheduling with low overheads. For example, the completely fair scheduler (CFS) [72], widely used in Linux systems, aims to provide fairness for all processes in the system. CFS maintains two queues (active and expired) to manage task scheduling. In addition, CFS gives a fixed time quantum for each process. Tasks are swapped between active and expired queues based on activation and expiration of the time quantum. However, complex heuristics are required to manage such queues. CFS also does not generalize to optimization objectives apart from performance and fairness. More importantly, CFS scheduling is limited to general-purpose cores and lacks support for specialized cores and hardware accelerators [19]. With the same limitations, shortest job first (SJF) [101] scheduler estimates the task's CPU processing time and assigns the first available resource to the task with the shortest execution time.

List scheduling techniques [85, 52] for DAGs [96, 22, 13] prioritize various objectives, such as energy [17, 93], fairness [109], security [108].

In general, this technique places the nodes (tasks) of a DAG in a list and provides a PE assignment and order at design time. Heterogeneous earliest finish time (HEFT) [96] is one example, in which an upward rank is computed to perform the scheduling decisions. The authors in [22] use a lookahead algorithm as an enhancement to the HEFT scheduler to improve the execution time, but suffers from fourth order complexity $O(n^4)$ on the number of tasks (n). Another recent technique shows improvement in performance with quadratic complexity [13]. However, these algorithms suffer from the time complexity problem and are tailored to particular objectives and fail to generalize to a combination of objectives and choice of applications.

Machine learning (ML)-based schedulers show promise in eliminating the drawbacks of list scheduling and runtime heuristic techniques. ML-based schedulers possess the capabilities to be further tuned at runtime [63]. A recent support vector machine (SVM)-based scheduler for OpenCL kernels assigns kernels (tasks) between CPUs and GPUs [105]. In contrast to schedulers that use supervised learning, authors in [65] uses reinforcement learning (RL) to schedule Tensorflow device placement, but lacks the ability of scheduling streaming jobs. DeepRM [63] uses deep neural networks with RL for scheduling at an application granularity as opposed to using the notion of DAGs. On the other hand, Decima [64] uses a combination of graph neural networks and RL to perform coarse-grained processing-cluster level scheduling for streaming DAGs.

RL-based scheduling techniques have two major drawbacks. *First*, they require a significant number of episodes to converge. For example, the technique proposed in [64] takes 50k episodes, with 1.5 seconds each, to converge to a solution that is equivalent to 21 hours of simulation in Nvidia Tesla P100 GPU. *Second*, the efficiency of an RL-based technique predominantly depends on the choice of the reward function. Usually, the reward function is hand-tuned, depending on the problem under

consideration.

To overcome these difficulties, we propose an IL-based scheduling methodology. Since IL uses an Oracle to construct a policy, it does not suffer from slow convergence, as seen in RL. IL-based policies were initially used in robotics to show their fast convergence property [88]. Recently, the use of imitation learning to intelligently manage power and energy consumption in SoCs has been demonstrated [49, 58]. To the best of our knowledge, *this is the first approach that applies IL for multi-application streaming task scheduling in heterogeneous many-core platforms*.

1.1.3 Background and Overview

The runtime scheduling problem addressed in this work is illustrated in Fig. 1.3. We consider streaming applications that can be modeled using directed acyclic graphs, such as the one shown in Fig. 1.3(a). These applications process data frames that arrive at a varying rate over time. For example, a WiFi-transmitter, one of our domain applications, receives and encodes raw data frames before they are transmitted over the air. Data frames from a single application or multiple simultaneous applications can overlap in time as they go through the tasks that compose the application. For instance, Task-1 in Fig. 1.3(a) can start processing a new frame, while other tasks continue processing earlier frames. Processing of a frame is said to be completed after the terminal task without any successor (Task-7 in Fig. 1.3(a)) is executed. We define the application formally to facilitate description of the schedulers.

Definition 1: An *application graph* $G_{App}(\mathcal{T}, \mathcal{E})$ is a directed acyclic graph, where each node $T_i \in \mathcal{T}$ represents the tasks that compose the application. Directed edge $e_{ij} \in \mathcal{E}$ from task T_i to T_j shows that T_j cannot start processing a new frame before the output of T_i reaches T_j for all $T_i, T_j \in \mathcal{T}$. v_{ij} for each edge $e_{ij} \in \mathcal{E}$ denotes the communication volume over this edge. It is used to account for the communication latency.

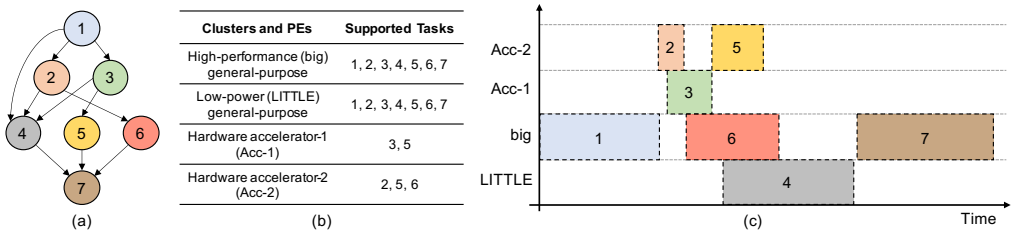


Figure 1.3: (a) An example DAG consisting of 7 tasks (b) A heterogeneous computing platform with 4 processing elements and list of tasks in DAG supported by each PE (c) A sample schedule of the DAG on the heterogeneous many-core system.

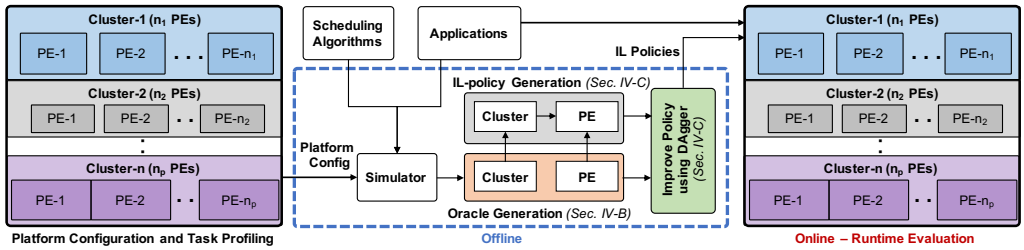


Figure 1.4: An overview of the proposed imitation learning framework for task scheduling in heterogeneous many-core systems. The framework integrates the system configuration, profiling information, scheduling algorithms and applications to construct Oracle, and train IL policies for task scheduling. The IL policies, that are improved using DAgger, are then evaluated on the heterogeneous many-core system at runtime.

Each task in a given application graph G_{App} can execute on different processing elements in the target DSSoC. We formally define the target DSSoC as follows:

Definition 2: An *architecture graph* $G_{Arch}(\mathcal{P}, \mathcal{L})$ is a directed graph, where each node $P_i \in \mathcal{P}$ represents processing elements, and $L_{ij} \in \mathcal{L}$ represents the communication links between P_i and P_j in the target SoC. The nodes and links have the following quantities associated with them:

- $t_{exe}(P_i, T_j)$ is the execution time of task T_j on PE $P_i \in \mathcal{P}$, if P_i can

execute (i.e., it supports) T_j.

- $t_{comm}(L_{ij})$ is the communication latency from P_i to P_j for all $P_i, P_j \in \mathcal{P}$.

- $C(P_i) \in \mathcal{C}$ is the PE cluster $P_i \in \mathcal{P}$ belongs to.

The DSSoC example in Fig. 1.3(b) assumes one big core cluster, one LITTLE core cluster, and two hardware accelerators each with a single PE in them for simplicity. The low-power (LITTLE) and high-performance (big) general-purpose clusters can support the execution of all tasks, as shown in the *supported tasks* column in Fig. 1.3(b). In contrast, hardware accelerators (Acc-1 and Acc-2) support only a subset of tasks.

A particular instance of the scheduling problem is illustrated in Fig. 1.3(c). Task-6 is scheduled to big core (*although it executes faster on Acc-2*) since Acc-2 is not available at the time of decision making. Similarly, Task-4 is scheduled to the LITTLE core (*even if it executes faster on big*) because the big core is utilized when Task-4 is ready to execute. In general, scheduling complex DAGs in heterogeneous many-core platforms present a multitude of choices making the runtime scheduling problem highly complex. The complexity increases further with: (1) overlapping DAGs at runtime, (2) executing multiple applications simultaneously, and (3) optimizing for objectives such as performance, energy, etc.

The proposed solution leverages imitation learning, and is outlined in Fig. 1.4. It is also referred to as learning by demonstration, and is an adaption of supervised learning for sequential decision making problems. The decision-making space is segmented into distinct decision epochs, called *states* ($\mathcal{S}$). There exists a finite set of actions $\mathcal{A}$ for every state $s \in \mathcal{S}$. IL uses policies that map each state (s) to a corresponding action.

Definition 3: Oracle Policy (expert) $\pi^*(s) : \mathcal{S} \rightarrow \mathcal{A}$ maps a given system state to the optimal action. In our runtime scheduling problem, the state includes the set of ready tasks and actions that correspond to assignment

Table 1.1: Summary of the notations used in this work

T_j	Task-j	$\mathcal{T}$	Set of Tasks
P_i	PE-i	$\mathcal{P}$	Set of PEs
c	Cluster-c	$\mathcal{C}$	Set of clusters
L_{ij}	Communication links between P_i to P_j	$\mathcal{L}$	Set of communication links
$t_{exe}(P_i, T_j)$	Execution time of task T_j on PE P_i	$t_{comm}(L_{ij})$	Communication latency from P_i to P_j
s	State-s	$\mathcal{S}$	Set of states
v_{jk}	Communication volume from task T_j to T_k	$\mathcal{A}$	Set of actions
$\mathcal{F}_S$	Static features	$\mathcal{F}_D$	Dynamic features
$\pi_C(s)$	Apply cluster policy for state s	$\pi_{P,c}(s)$	Apply PE policy in cluster-c for state s
π	Policy	π^*	Oracle policy
π^G	Policy for many-core platform configuration G	π^{*G}	Oracle for many-core platform configuration G

of tasks $\mathcal{T}$ to processing elements $\mathcal{P}$. Given the Oracle π^*, the goal with imitation learning is to learn a runtime policy that can approximate it. We construct an Oracle offline and approximate it using a hierarchical policy with two levels. Consider a generic heterogeneous many-core platform with a set of clusters $\mathcal{C}$, as illustrated in Fig. 1.4. At the first level, an IL policy chooses one cluster (among n clusters) for a task to be executed in.

The first-level policy assigns the ready tasks to one of the clusters in $\mathcal{C}$, since each PE within the same cluster has the same static parameters. Then, a cluster-level policy assigns the tasks to a specific PE within that cluster. The details of state representation, Oracle generation, and hierarchical policy design are presented in the next section.

1.1.4 Proposed Methodology and Approach

This section first introduces the system state representation, including the features used by the IL policies. Then, it presents the Oracle generation process, and the design of the hierarchical IL policies. Table 1.1 details the notations that will be used hereafter.

1.1.5 System State Representation

Offline scheduling algorithms are NP-complete even though they rely on static features, such as average execution times. The complexity of runtime decisions is further exacerbated as the system schedules multiple applications that exhibit streaming behavior. In the streaming scenario, incoming frames do not observe an empty system with idle processors. In strong contrast, PEs have different utilization, and there may be an arbitrary number of partially processed frames in the wait queues of the PEs. Since our goal is to learn a set of policies that generalize to all applications and all streaming intensities, the ability to learn the scheduling decisions critically depends on the effectiveness of state representation. The system state should encompass both static and dynamic aspects of the set of tasks, applications, and the target platform. Naive representations of DAGs include adjacency matrix and adjacency list. However, these representations suffer from drawbacks such as large storage requirements, highly sparse matrices which complicates the training of supervised learning techniques, and scalability for multiple streaming applications. In contrast, we carefully study the factors that influence task scheduling in a streaming scenario and construct features that accurately represent the system state. We broadly categorize the features that make up the state as follows:

- *Task features:* This set includes the attributes of individual tasks. They can be both static, such as average execution time of a task on a given

PE ($t_{exe}(P_i, T_j)$), and dynamic, such as the relative order of a task in the queue.

- *Application features:* This set describes the characteristics of the entire application. They are static features, such as the number of tasks in the application and the precedence constraints between them.

- *PE features:* This set describes the dynamic state of the processing elements. Examples include the earliest available times (readiness) of the PEs to execute tasks.

The static features are determined at the design time, whereas the dynamic features can only be computed at runtime. The static features aid in exploiting design time behavior. For example, $t_{exe}(P_i, T_j)$ helps the scheduler compare the expected performance of different PEs. Dynamic features, on the other hand, present the runtime dependencies between tasks and jobs and also the busy states of the processing elements. For example, the expected time when cluster c becomes available for processing adds invaluable information, which is only available at runtime.

In summary, the features of a task comprehensively represent the task itself and the state of the processing elements in the system to effectively learn the decisions from the Oracle policy. The specific types of features used in this work to represent the state and their categories are listed in Table 1.2. The static and dynamic features are denoted as $\mathcal{F}_S$ and $\mathcal{F}_D$, respectively. Then, we define the systems state at a given time instant k using the features in Table 1.2 as:

$$s_k = \mathcal{F}_{S,k} \cup \mathcal{F}_{D,k} \tag{1.1}$$

where $\mathcal{F}_{S,k}$ and $\mathcal{F}_{D,k}$ denote the static and dynamic features respectively at a given time instant k. For an SoC with 16 processing elements grouped as 5 clusters, we obtain a set of 45 features for the proposed IL technique.

Table 1.2: Types of features employed for state representation from point of view of task T_j

Feature Type	Feature Description	Feature Categories
Static ($\mathcal{F}_S$)	ID of task-j in the DAG	Task
	Execution time of a task T_j in PE P_i ($t_{exe}(P_i, T_j)$)	Task PE
	Downward depth of task T_j in the DAG	Task Application
	IDs of predecessor tasks of task T_j	Task Application
	Application ID	Application
	Power consumption of task T_j in PE P_i	Task PE
Dynamic ($\mathcal{F}_D$)	Relative order of task T_j in the ready queue	Task
	Earliest time when PEs in a cluster-c are ready for task execution	PE
	Clusters in which predecessor tasks of task T_j executed	Task
	Communication volume from task T_j to task T_k(v_{jk})	Task

1.1.5.1 Oracle Generation

The goal of this work is to develop generalized scheduling models for streaming applications of multiple types to be executed on heterogeneous many-core systems. The generality of the IL-based scheduling framework enables using IL with any Oracle. The Oracle can be any scheduling algorithm that optimizes an arbitrary metric, such as execution time, power consumption, and total SoC energy.

Algorithm 1: Hierarchical imitation learning Framework

```
for task T ∈ 𝒯 do
    s = Get current state for task T
    /* Level-1 IL policy to assign cluster */
    c = π_C(s)
    /* Level-2 IL policy to assign PE */
    p = π_{P,c}(s)
    /* Assign T to the predicted PE */
end
```

To generate the training dataset, we implemented both optimal schedulers using CP and heuristics. These schedulers are integrated into a SoC simulation framework, as explained under experimental results. Suppose a new task T_j becomes ready at time k. The Oracle is called to schedule the task to a PE. The Oracle policy for this action task with system state s_k can be expressed as:

$$\pi^*(s_k) = P_i, \tag{1.2}$$

where $P_i \in \mathcal{P}$ is the PE T_j scheduled to and s_k is the system state defined in Equation 1.1. After each scheduling action, the particular task that is scheduled (T_j), the system state $s_k \in \mathcal{S}$, and the scheduling decision are added to the training data. To enable the Oracle policies to generalize for different workload conditions, we constructed workload mixes using the target applications at different data rates, as detailed in Section 1.1.6.1.

1.1.5.2 IL-based Scheduling Framework

This section presents the hierarchical IL-based scheduler for runtime task scheduling in heterogeneous many-core platforms. A hierarchical structure is more scalable since it breaks a complex scheduling problem down into simpler problems. Furthermore, it achieves a significantly higher classification accuracy compared to a flat classifier (>93% versus 55%), as detailed in Section 1.1.6.4.

Our hierarchical IL-based scheduler policies approximate the Oracle with two levels, as outlined in Algorithm 1. The first level policy $\pi_C(s) : \mathcal{S} \rightarrow \mathcal{C}$ is a coarse-grained scheduler that assigns tasks into clusters. This is a natural choice since individual PEs within a cluster have identical static parameters, i.e., they differ only in terms of their dynamic states. The second level (i.e., fine-grained scheduling) consists of *one dedicated policy* $\pi_{P,c}(s) : \mathcal{S} \rightarrow \mathcal{P}$ for each cluster $c \in \mathcal{C}$. These policies assign the input task to a PE within its own cluster, i.e., $\pi_{P,c}(s) \in \mathcal{P}^c$, $\forall c \in \mathcal{C}$. We leverage off-the-shelf machine learning techniques, such as decision trees and neural networks, to construct the IL policies. The application of these policies approximates the corresponding Oracle policies constructed offline.

IL policies suffer from error propagation as the state-action pairs in the Oracle are not necessarily i.i.d. (independent and identically distributed). Specifically, if the decision taken by the IL policies at a particular decision epoch is different from the Oracle, then the resultant state for the next epoch is also different with respect to the Oracle. Therefore, the error further accumulates at each decision epoch. This can occur during runtime task scheduling when the policies are applied to applications that the policies did not train with. This problem is addressed by the data aggregation algorithm (DAgger), proposed to improve IL policies [82]. DAgger adds the system state and the Oracle decision to the training data whenever the IL policy makes a wrong decision. Then, the policies are retrained after the execution of the workload.

DAgger is not readily applicable to the runtime scheduling problem since the number of states is unbounded as a scheduling decision at time t for state s (s_t) can result in any possible resultant state, s_{t+1}. In other words, the feature space is continuous, and hence, it is infeasible to generate an exhaustive Oracle offline. We overcome this challenge by generating an Oracle on-the-fly. More specifically, we incorporate the proposed framework into a simulator. The offline scheduler used as the Oracle is called

Algorithm 2: Methodology to aggregate data in a hierarchical imitation learning framework

```
for task T ∈ 𝒯 do
    s = Get current state for task T
    if π_C(s) == π*_C(s) then
        if π_{P,c}(s) != π*_{P,c}(s) then
            Aggregate state s and label π*_{P,c}(s) to the dataset
        end
    end
    else
        Aggregate state s and label π*_C(s) to the dataset
        c* = π*_C(s)
        if π_{P,c*}(s) != π*_{P,c*}(s) then
            Aggregate state s and label π*_{P,c}(s) to the dataset
        end
    end
    /* Assign T to the predicted PE */
end
```

dynamically for each new task. Then, we augment the training data with all the features, Oracle actions, as well as the results of the IL policy under construction. Hence, the data aggregation process is performed as part of the dynamic simulation.

The hierarchical nature of the proposed IL framework introduces one more complexity to data aggregation. The cluster policy's output may be correct, while the PE cluster reaches a wrong decision (or vice versa). If the cluster prediction is correct, we use this prediction to select the PE policy of that cluster, as outlined in Algorithm 2. Then, if the PE prediction is also correct, the execution continues; otherwise, the PE data is aggregated in the dataset. However, if the cluster prediction does not align with the Oracle, in addition to aggregating the cluster data, an on-the-fly Oracle is invoked to select the PE policy, then the PE prediction is compared to the Oracle, and the PE data is aggregated in case of a wrong prediction.

1.1.6 Experimental Results

Section 1.1.6.1 presents the experimental methodology and setup. Section 1.1.6.2 explores different machine learning classifiers for IL. The significance of the proposed features is studied using a decision tree classifier in Section 1.1.6.3. Section 1.1.6.4 presents the evaluation of the proposed IL-scheduler. Section 1.1.6.5 analyzes the generalization capabilities of IL-scheduler. The performance analysis with multiple workloads is presented in Section 1.1.6.6. We demonstrate the evaluation of the proposed IL technique to energy-based optimization objectives in Section 1.1.6.7. Section 1.1.6.8 presents comparisons with RL-based scheduler and Section 1.1.6.9 analyzes the complexity of the proposed approach.

1.1.6.1 Experimental Methodology and Setup

Table 1.3: Characteristics of applications used in this study and the number of frames of each application in the workload

App	# of Tasks	Execution Time (μs)	Supported Clusters	Representation in workload	
				#frames	#tasks
WiFi-TX	27	301	big, LITTLE, FFT	69	1863
WiFi-RX	34	71	big, LITTLE, FFT, Viterbi	111	3774
RangeDet	7	177	big, LITTLE, FFT	64	448
SC-TX	8	56	big, LITTLE	64	512
SC-RX	8	154	big, LITTLE, Viterbi	91	728
TempMit	10	81	big, LITTLE, Matrix mult.	101	1010
TOTAL				**500**	**8335**

Domain Applications: The proposed IL scheduling methodology is evaluated using applications from wireless communication and radar processing domains. We employ WiFi-transmitter (*WiFi-TX*), WiFi-receiver (*WiFi-RX*), range detection (*RangeDet*), single-carrier transmitter (*SC-TX*), single-carrier receiver (*SC-RX*) and temporal mitigation (*TempMit*) applications, as summarized in Table 1.3. We construct workload mixes using these applications and run them in parallel.

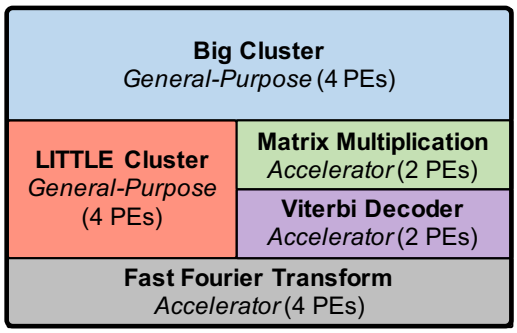


Figure 1.5: Configuration of the heterogeneous many-core platform comprising 16 processing elements, used for scheduler evaluations.

Heterogeneous DSSoC Configuration: Considering the nature of applications, we employ a DSSoC with 16 PEs, including accelerators for the most computationally intensive tasks; they are divided into five clusters with multiple homogeneous PEs, as illustrated in Fig. 1.5. To enable power-performance trade-off while using general-purpose cores, we include a big cluster with four Arm A57 cores and a LITTLE cluster with four Arm A53 cores. In addition, the DSSoC integrates accelerator clusters for matrix multiplication, FFT, and Viterbi decoder to address the computing requirements of the target domain applications summarized in Table 1.3. The accelerator interfaces are adapted from [57]. The number of accelerator instances in each cluster is selected based on how much the target applications use them. For example, three out of the six reference applications involve FFT, while range detection application alone has three FFT opera-

tions. Therefore, we employ four instances of FFT hardware accelerators and two instances of Viterbi and matrix multiplication accelerators, as shown in Fig. 1.5.

Simulation Framework: We evaluate the proposed IL scheduler using the discrete event-based simulation framework [14], *which is validated against two commercial SoCs*: Odroid-XU3 [39] and Zynq Ultrascale+ ZCU102 [7]. This framework enables simulations of the target applications modeled as DAGs under different scheduling algorithms. More specifically, a new instance of a DAG arrives following a specified inter-arrival time rate and distribution, such as an exponential distribution. After the arrival of each DAG instance, called a frame, the simulator calls the scheduler under study. Then, the scheduler uses the information in the DAG and the current system state to assign the ready tasks to the waiting queues of the PEs. The simulator facilitates storing this information and the scheduling decision to construct the Oracle, as described in Section 1.1.5.1.

The execution times and power consumption for the tasks in our domain applications are profiled on Odroid-XU3 and Zynq ZCU102 SoCs. The simulator uses these profiling results to determine the execution time and power consumption of each task. After all the tasks that belong to the same frame are executed, the processing of the corresponding frame completes. The simulator keeps track of the execution time and energy consumed for each frame. These end-to-end values are within 3%, on average, of the measurements on Odroid-XU3 and Zynq ZCU102 SoCs.

Scheduling Algorithms used for Oracle and Comparisons: We developed a CP formulation using IBM ILOG CPLEX Optimization Studio [8] to obtain the optimal schedules whenever the problem size allows. After the arrival of each frame, the simulator calls the CP solver to find the schedule dynamically as a function of the current system state. Since the CP solver takes hours for large inputs (~100 tasks), we implemented two versions with one minute ($\mathrm{CP}_{1-\mathrm{min}}$) and five minutes ($\mathrm{CP}_{5-\mathrm{min}}$) time-out per

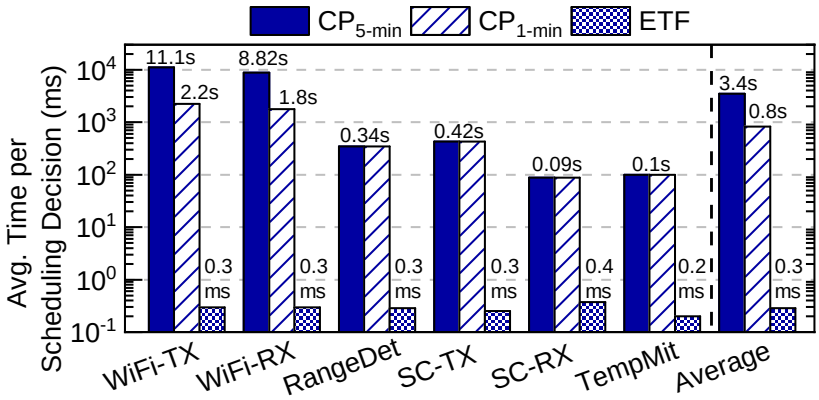


Figure 1.6: A comparison of average runtime per scheduling decision for each application with $\mathrm{CP}_{5-\mathrm{min}}$, $\mathrm{CP}_{1-\mathrm{min}}$ and ETF schedulers.

scheduling decision. When the model fails to find an optimal schedule, we use the best solution found within the time limit. Fig. 1.6 shows that the average time of the CP solver per scheduling decision for the benchmark applications is about 0.8 seconds and 3.5 seconds, respectively, based on the time limit. Consequently, one entire simulation can take up to 2 days, even with a time-out.

We also implemented the ETF heuristic scheduler, which goes over all tasks and possible assignments to find the earliest finish time considering communication overheads. Its average execution time is close to 0.3 ms, which is still prohibitive for a runtime scheduler, as shown in Fig. 1.6. However, we observed that it performs better than $\mathrm{CP}_{1-\mathrm{min}}$ and marginally worse than $\mathrm{CP}_{5-\mathrm{min}}$, as we detail in Section 1.1.6.4.

Oracle generation with the CP formulation is not practical for two reasons. First, it is possible that for small input sizes (e.g., less than ten tasks), there might be multiple (incumbent) optimal solutions, and CP would choose one of them randomly. The other reason is that for large input sizes, CP terminates at the time limit providing the best solution found so far, which is sub-optimal. The sub-optimal solutions produced by

Table 1.4: Classification accuracies of trained IL policies with different machine learning classifiers.

Classifier	Cluster Policy	LITTLE Policy	big Policy	MatMult Policy	FFT Policy	Viterbi Policy
DT	99.6	**93.8**	**95.1**	99.9	99.5	100
SVC	95.0	**85.4**	**89.9**	97.8	97.5	98.0
LR	89.9	**79.1**	**72.0**	98.7	98.2	98.0
NN	97.7	**93.3**	**93.6**	99.3	98.9	98.1

Table 1.5: Execution time and storage overheads per IL policy for decision tree and neural network classifiers.

Classifier	Latency (μs)		Storage (KB)
	Odroid-XU3 (Arm A15)	Zynq Ultrascale+ ZCU102 (Arm A53)	
DT	1.1	1.1	19.3
NN	14.4	37	16.9

CP vary based on the problem size and the limit. In contrast, ETF is easier to imitate at runtime and its results are within 8.2% of CP_{5-min} results. Therefore, we use ETF as the Oracle policy in our experiments and use the results of CP schedulers as reference points. We train IL policies for this Oracle in Section 1.1.6.2 and evaluate their performance in Section 1.1.6.4.

1.1.6.2 Exploring Different Machine Learning Classifiers for IL

We explore various ML classifiers within the IL methodology to approximate the Oracle policy. One of the key metrics that drive the choice of machine learning techniques is the classification accuracy of the IL policies. At the same time, the policy should also have a low storage and execution time overheads. We evaluate the following algorithms for classification accuracy and implementation efficiency: decision tree (DT), support vector classifier (SVC), logistic regression (LR), and a multi-layer perceptron

Table 1.6: Training accuracy of IL policies with subsets of the proposed feature set

Features Excluded from Training	Cluster Policy	LITTLE Policy	big Policy	MatMul Policy	FFT Policy	Viterbi Policy
None	**99.6**	**93.8**	**95.1**	**99.9**	**99.5**	**100**
Static features	87.3	93.8	92.7	99.9	99.5	100
Dynamic features	88.7	52.1	57.6	94.2	70.5	98
PE availability times	92.2	51.1	61.5	94.1	66.7	98.1
Task ID, depth, app. ID	90.9	93.6	95.3	99.9	99.5	100

neural network (NN) with 4 hidden layers and 32 neurons in each hidden layer.

The classification accuracy of ML algorithms under study are listed in Table 1.4. In general, all classifiers achieve a high accuracy to choose the cluster (the first column). At the second level, they choose the correct PE with high accuracy (>97%) within the hardware accelerator clusters. However, they have lower accuracy and larger variation for the LITTLE and big clusters (highlighted columns). This is intuitive as the LITTLE and big clusters can execute all types of tasks in the applications, whereas accelerators execute fewer tasks. In strong contrast, a flat policy, which directly predicts the PE, results in training accuracy with 55% at best. Therefore, we focus on the proposed hierarchical IL methodology.

Decision trees (DT) trained with a maximum depth of 12 produce the best accuracy for the cluster and PE policies, with more than 99.5% accuracy for the cluster and hardware acceleration policies. DT also produces an accuracy of 93.8% and 95.1% to predict PEs within the LITTLE and big clusters, respectively, which is the highest among all the evaluated classifiers. The classification accuracy of NN policies are comparable to DT, with a slightly lower cluster prediction accuracy of 97.7%. In contrast, support vector classifiers (SVC) and logistic regression (LR) are not preferred due to lower accuracy of less than 90% and 80%, respectively, to

predict PEs within LITTLE and big clusters.

We choose decision trees and neural networks to analyze the latency and storage overheads (due to their superior performance). The latency of DT is 1.1µs on Arm Cortex-A15 in Odroid-XU3 and on Arm Cortex-A53 in Zynq ZCU102, as shown Table 1.5. In comparison, the scheduling overhead of CFS, the default Linux scheduler, on Zynq ZCU102 running Linux Kernel 4.9 is 1.2µs, which is slightly larger than our solution. The storage overhead of an DT policy is 19.33 KB. The NN policies incur an overhead of 14.4µs on the Arm Cortex-A15 cluster in Odroid-XU3 and 37µs on Arm Cortex-A53 in Zynq, with a storage overhead of 16.89 KB. NNs are preferable for use in an online environment as their weights can be incrementally updated using the back-propagation algorithm. However, due to competitive classification accuracy and lower latency overheads of DT over NNs, we choose DT for the rest of the experiments.

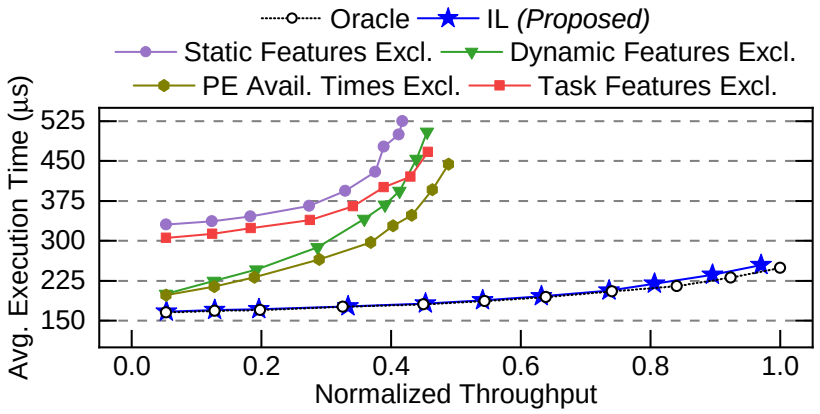


Figure 1.7: Average execution time comparison of the applications with Oracle, IL (*Proposed*) and IL policies with subsets of features. As shown, the average execution time with IL closely follows the Oracle.

1.1.6.3 Feature Space Exploration with Decision Tree Classifier

This section explores the significance of the features chosen to represent the state. For this analysis, we assess the impact of the input features on the training accuracy with DT classifier and average execution time following a systematic approach.

The training accuracy with subsets of features and the corresponding scheduler performance is shown in Table 1.6 and Fig. 1.7 respectively. *First*, we exclude all static features from the training dataset. The training accuracy for the prediction of the cluster significantly drops by 10%. Since we use hierarchical IL policies, an incorrect first-level decision results in a significant penalty for the decisions at the next level. *Second*, we exclude all dynamic features from training. This results in a similar impact for the cluster policy (10%) but significantly affects the policies constructed for the LITTLE, big, and FFT. *Next*, a similar trend is observed when PE availability times are excluded from the feature set. The accuracy is marginally higher since the other dynamic features contribute to learning the scheduling decisions. *Finally*, we remove a few task related features such as the downward depth, task, and application identifier. In this case, the impact is to the cluster policy accuracy since these features describe the node in the DAG and influence the cluster mapping.

As observed in Fig. 1.7, the average execution time of the workload significantly degrades when all features are not included. Hence, the chosen features help to construct effective IL policies, approximating the Oracle with over 99% accuracy in execution time.

1.1.6.4 IL-Scheduler Performance Evaluation

This section compares the performance of the proposed policy to the ETF Oracle, $\mathrm{CP}_{1-\mathrm{min}}$, and $\mathrm{CP}_{5-\mathrm{min}}$. Since heterogeneous many-core systems are capable of running multiple applications simultaneously, we stream the frames in our application mix (see Table 1.3) with increasing injection

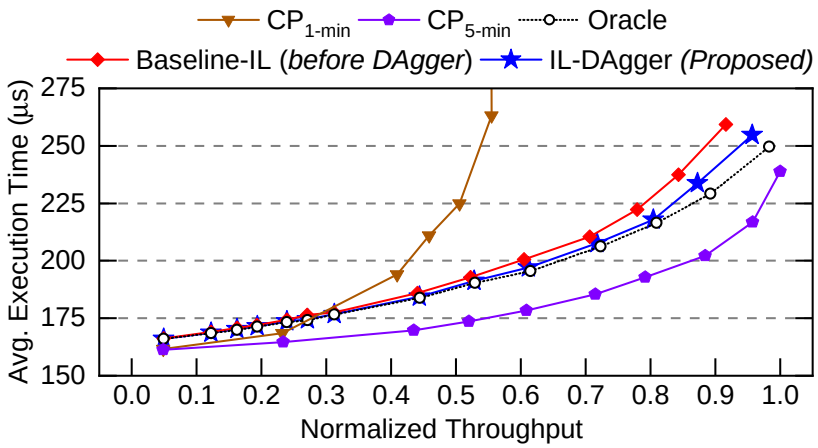


Figure 1.8: Comparison of average job execution time between Oracle, CP solutions, and imitation learning policies to schedule a workload comprising a mix of six streaming applications. IL scheduler policies with baseline-IL (*before DAgger*) and with IL-DAgger (*Proposed*) are shown in the comparison.

rates. For example, a normalized throughput of 1.0 in Fig. 1.8 corresponds to 19.78 frames/ms. Since the frames are injected faster than they can be processed, there are many overlapping frames at any given time.

First, we train the IL policies with all six reference applications and refer to this as the baseline-IL scheduler. IL policies suffer from error propagation due to the non i.i.d. nature of training data. To overcome this limitation, we use a data aggregation technique adapted for a hierarchical IL framework (IL-DAgger), as discussed in Section 1.1.5.2. A DAgger iteration involves executing the entire workload. We execute ten DAgger iterations and choose the best iteration with performance within 2% of the Oracle. If we fail to achieve the target, we continue to perform more iterations.

Fig. 1.8 shows that the proposed IL-DAgger scheduler performs almost identical to the Oracle; the mean average percentage difference between them is 1%. More notably, the gap between the proposed IL-DAgger pol-

icy and the optimal CP_{5-min} solution is only 9.22%. We emphasize that CP_{5-min} is included only as a reference point, but it has six orders of magnitude larger execution time overhead and cannot be used at runtime. Furthermore, the proposed approach performs better than CP_{1-min}, which is not able to find a good schedule within the one-minute time limit per decision. Finally, we note that the baseline IL can approach the performance of the proposed policy. This is intuitive since both policies are tested on known applications in this experiment. This is in contrast to the leave one out experiments presented in Section 1.1.6.5.

Pulse Doppler Application Case Study: We demonstrate the applicability of the proposed IL-scheduling technique in complex scenarios using a pulse Doppler application. It is a real-world radar application, which computes the velocity of a moving target object. This application is significantly more complex, with 13-64$\times$ more tasks than the other applications. Specifically, it consists of 449 tasks comprising 192 FFT tasks, 128 inverse-FFT tasks, and 129 other computations. The FFT and inverse-FFT operations can execute on the general-purpose cores and hardware accelerators. In contrast, the other tasks can execute only on the general-purpose cores.

The proposed IL policies achieve an average execution time within 2% of the Oracle. The 2% error is acceptable, considering that the application saturates the computing platform quickly due to its high complexity. Moreover, the CP-based approach does not produce a viable solution either with 1-minute or 5-minute time limits due to the large problem size. For this reason, this application is not included in our workload mixes and the rest of the comparisons.

1.1.6.5 Illustration of Generalization with IL for Unseen Applications, Runtime Variations and Platforms

This section analyzes the generalization of the proposed IL-based scheduling approach to unseen applications, runtime variations, and many-core

platform configurations.

IL-Scheduler Generalization to Unseen Applications using Leave-one-out Experiments: IL, being an adaptation of supervised learning for sequential decision making, suffers from lack of generalization to unseen applications. To analyze the effects of unseen applications, we train IL policies, excluding applications one each at a time from the training dataset [102].

To compare the performances of two schedulers S_1 and S_2, we use the job slowdown metric $slowdown_{S_1,S_2} = T_{S_1}/T_{S_2}$. $Slowdown_{S_1,S_2} > 1$ when $T_{S_1} > T_{S_2}$ [63]. The average slowdown of scheduler S_1 with respect to scheduler S_2 is computed as the average slowdown for all jobs at all injection rates. The results present an interesting and intuitive explanation of the average job slowdown in execution times for each of the leave-one-out experiments.

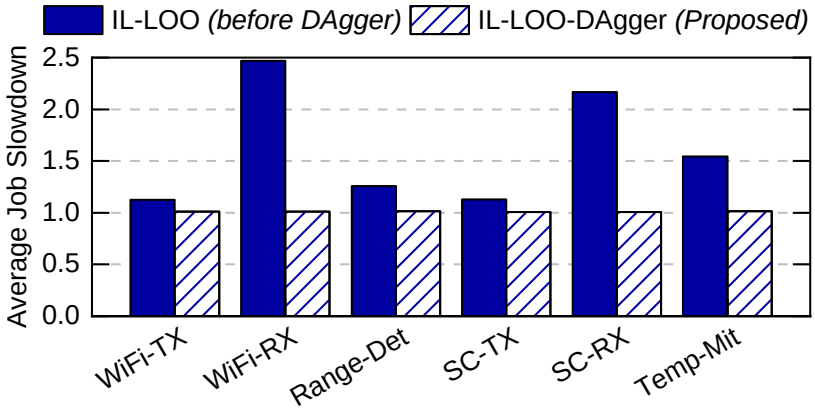


Figure 1.9: Average slowdown of IL policies in comparison with Oracle for leave-one-out (LOO) experiments before and after DAgger (*Proposed*).

Fig. 1.9 shows the average slowdown of the baseline IL (IL-LOO) and proposed policy with DAgger iterations (IL-LOO-DAgger) with respect to the Oracle. We observe that the proposed policy outperforms the baseline IL for all applications, with the most significant gains obtained for WiFi-RX

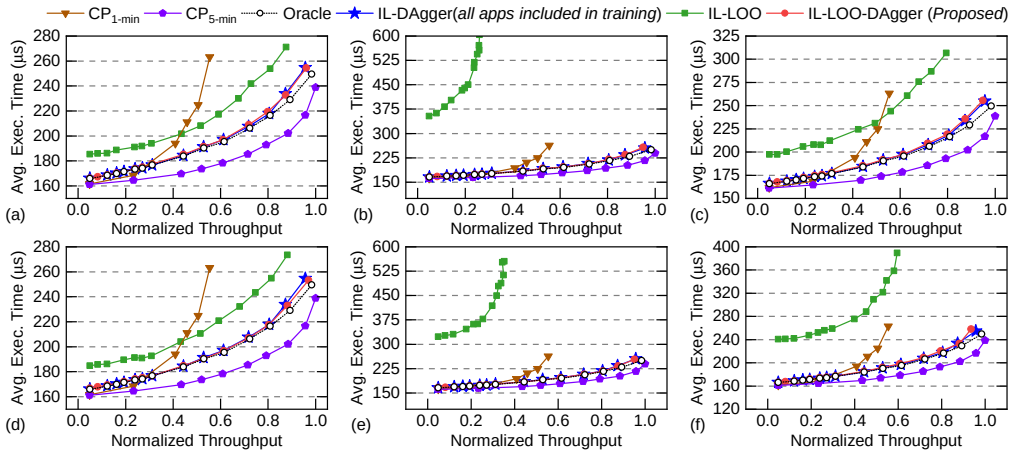


Figure 1.10: Average execution time with Oracle, IL-DAgger (*all applications are included for training*), IL with one application excluded from training (IL-LOO) and finally, the leave-one-out policy improved with DAgger (*Proposed* IL-LOO-DAgger) technique. The *excluded* applications are: (a) WiFi-TX, (b) WiFi-RX, (c) Range Detection (d) Single-Carrier TX (e) Single-Carrier RX and (f) Temporal Mitigation.

and SC-RX applications. These two applications consist of a Viterbi decoder operation, which is very expensive to compute on general-purpose cores and highly efficient to compute on hardware accelerators. When these applications are excluded, the IL policies are not exposed to the corresponding states in the training dataset and make incorrect decisions. The erroneous PE assignments lead to an average slowdown of more than 2× for the receiver applications. The slowdown when the transmitter applications (WiFi-TX and SC-TX) are excluded from training is approximately 1.13×. Range detection and temporal mitigation applications experience a slowdown of 1.25× and 1.54×, respectively, for leave-one-out experiments. The extent of the slowdown in each scenario depends on the application excluded from training and its execution time profile in the different processing clusters. In summary, the average slowdown of all leave-one-out IL policies after DAgger (IL-LOO-DAgger) improves to

Table 1.7: Standard deviation (in percentage of execution time) profiling of applications in Odroid-XU3 and Zynq ZCU-102.

Application	WiFi-TX	WiFi-RX	RangeDet	SC-TX	SC-RX	TempMit
Zynq ZCU-102	0.34	0.56	0.66	1.15	1.80	0.63
Odroid-XU3	6.43	5.04	5.43	6.76	7.14	3.14

~1.01× in comparison with the Oracle, as shown in Fig. 1.9.

Fig. 1.10(a)-(f) show the average job execution times for the Oracle (ETF), baseline-IL, IL with leave-one-out and DAgger for IL with leave-one-out policies for each of the applications. The highest number of DAgger iterations needed was 8 for SC-RX application, and the lowest was 2 for range detection application. If the DAgger criterion is relaxed to achieving a slowdown of 1.02×, all applications achieve the same in less than 5 iterations. A drastic improvement in the accuracy of the IL policies with few iterations shows that the policies generalize quickly and well to unseen applications, thus making them suitable for applicability at runtime.

IL-Scheduler Generalization with Runtime Variations: Tasks experience runtime variations due to variations in system workload, memory, and congestion. Hence, it is crucial to analyze the performance of the proposed approach when tasks experience such variations, rather than observing only their static profiles. Our simulator accounts for variations by using a Gaussian distribution to generate variations in execution time [106]. To allow evaluation in a realistic scenario, all tasks in every application are profiled on big and LITTLE cores of Odroid-XU3, and, on Cortex-A53 cores and hardware accelerators on Zynq for variations in execution time.

We present the average standard deviation as a ratio of execution time for the tasks in Table 1.7. The maximum standard deviation is less than 2% of the execution time for the Zynq platform, and less than 8% on the Odroid-XU3. To account for variations in runtime, we add a noise of 1%, 5%, 10%, and 15% in task execution time during simulation. The IL policies achieve

average slowdowns of less than 1.01× in all cases of runtime variations. Although IL policies are trained with static execution time profiles, the aforementioned results demonstrate that the IL policies adapt well to execution time variations at runtime. Similarly, the policies also generalize to variations in communication time and power consumption.

IL-Scheduler Generalization with Platform Configuration: This section presents a detailed analysis of the IL policies by varying the configuration i.e., the number of clusters, general-purpose cores, and hardware accelerators. To this end, we choose five different SoC configurations presented in Table 1.8. The Oracle policy for a configuration G1 is denoted by π^{*G1}. An IL policy evaluated on configuration G1 is denoted as π^{G1}. G1 is the baseline configuration that is used for extensive evaluation. Between configurations G1–G4, we vary the number of PEs within each cluster. We also consider a degenerate case that comprises only LITTLE and big clusters (configuration G5). We train IL policies with only configuration G1. The average execution times of π^{G1}, π^{G2}, and π^{G3} are within 1%, π^{G4} performs within 2%, and π^{G5} performs within 3%, of their respective Oracles. The accuracy of π^{G5} with respect to the corresponding Oracle (π^{*G5}) is slightly lower (97%) as the platform saturates the computing resources very quickly, as shown in Fig. 1.11.

Table 1.8: Configuration of many-core platforms.

Platform Config.	LITTLE PEs	big PEs	MatMul Acc. PEs	FFT Acc. PEs	Decoder Acc. PEs
G1 (Baseline)	4	4	2	4	2
G2	2	2	2	2	2
G3	1	1	1	1	1
G4	4	4	1	1	1
G5	4	4	0	0	0

Based on these experiments, we observe that the IL policies generalize well for the different many-core platform configurations. The change in

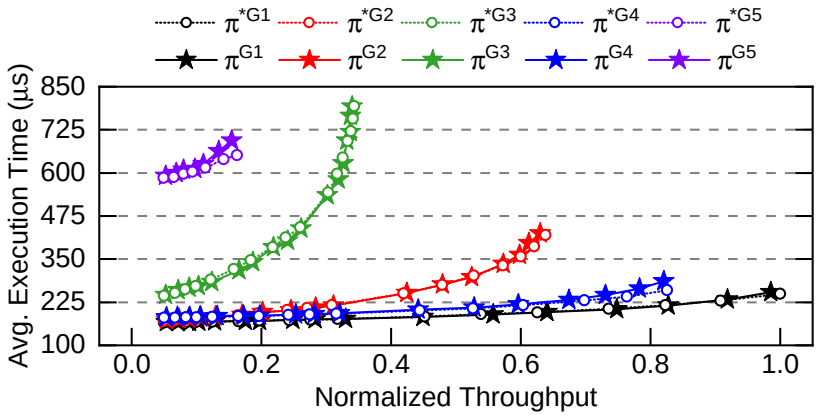


Figure 1.11: IL policy evaluation with multiple many-core platform configurations. IL policies are trained with only configuration G1.

system configuration is accurately captured in the features (in execution times, PE availability times, etc.), which enables us to generalize well to new platform configurations. When the cluster configuration in the many-core platform changes, the IL policies generalize well (within 3%) but can also be improved by using DAgger to obtain improved performance (within 1% of the Oracle).

1.1.6.6 Performance Analysis with Multiple Workloads

To demonstrate the generalization capability of the IL policies trained and aggregated on one workload (*IL-DAgger*), we evaluate the performance of the same policies on *50* different workloads consisting of different combinations of application mixes at varying injection rates, and each of these workloads contains 500 frames. For this extensive evaluation, we consider workloads each of which are intensive on one of WiFi-TX, WiFi-RX, range detection, SC-TX, SC-RX, and temporal mitigation. Finally, we also consider workloads in which all applications are distributed similarly.

Fig. 1.12 presents the average slowdown for each of the *50* different workloads (represented as *W-1*, *W-2* and so on). While W-22 observes a

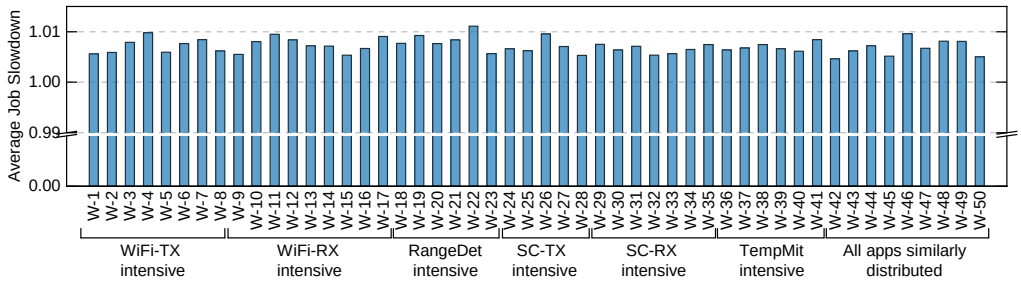


Figure 1.12: Comparison of average job slowdown normalized with IL-DAgger (*Proposed*) policies against the Oracle for *50* different workloads. The slowdown of IL-DAgger policies are shown for workloads with different *intensities* of each application in the benchmark suite.

slowdown of 1.01× against the Oracle, all other workloads experience an average slowdown of less than 1.01× (within 1% of Oracle). Independent of the distribution of the applications in the workloads, the IL policies approximate the Oracle well. On average, the slowdown is less than 1.01×, demonstrating the IL policies generalize to different workloads and streaming intensities.

1.1.6.7 Evaluation with Energy and Energy-Delay Objectives

Average execution time is crucial in configuring computing systems for meeting application latency requirements and user experience. Another critical metric in modern computing systems, especially battery-powered platforms, is energy consumption [67, 80]. Hence, this section presents the proposed IL-based approach with the following objectives: performance, energy, energy-delay product (EDP), and energy-delay^2 product (ED^{2}P). We adapt ETF to generate Oracles for each objective. Then, the different Oracles are used to train IL policies for the corresponding objectives. The scheduling decisions are significantly more complex for these Oracles. Hence, we use an DT of depth 16 (execution time uses DT of depth 12) to learn the decisions accurately. The average latency per scheduling decision

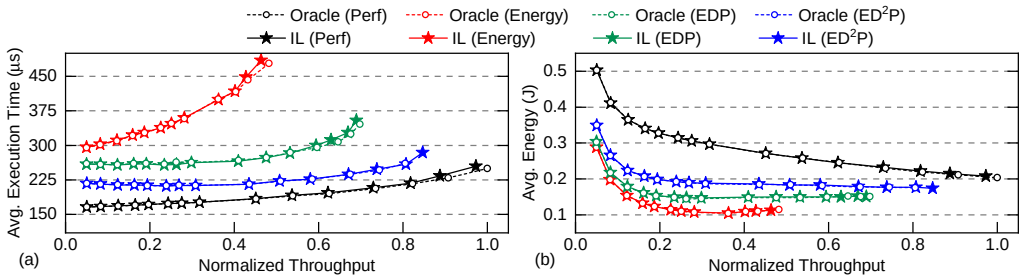


Figure 1.13: (a) Average execution time and (b) average energy consumption of the workload with Oracles and IL policies for performance, energy, energy-delay product (EDP) and energy-delay^2 product (ED^{2}P) objectives.

remains similar for DT of depth 16 (~1.1µs) on Cortex-A53.

Fig. 1.13(a) and Fig. 1.13(b) present the average execution time and average energy consumption, respectively, for IL policies with different objectives. The lowest energy is achieved by the energy Oracle, while it increases as more emphasis is added to performance (EDP $\rightarrow$ ED^{2}P $\rightarrow$ performance), as expected. The average execution time and energy consumption in all cases are within 1% of the corresponding Oracles. This demonstrates the proposed IL scheduling approach is powerful as it learns from Oracles that optimize for any objective.

1.1.6.8 Comparison with Reinforcement Learning

Since the state-of-the-art machine learning techniques [63, 64] do not target streaming DAG scheduling in heterogeneous many-core platforms, we implemented a policy-gradient based reinforcement learning technique using a deep neural network (multi-layer perceptron with 4 hidden layers with 32 neurons in each hidden layer) to compare with the proposed IL-based task scheduling technique. For the RL implementation, we vary the exploration rate between 0.01 to 0.99 and learning rate from 0.001 to 0.01. The reward function is adapted from [64]. RL starts with random weights

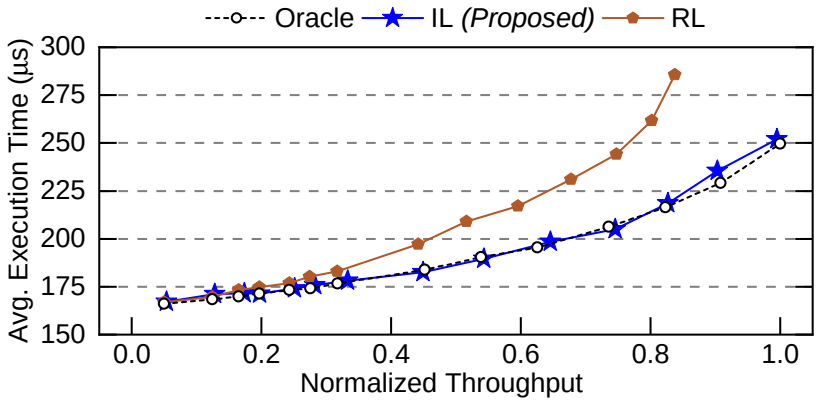


Figure 1.14: Comparison of average execution time between Oracle, IL, and RL policies to schedule a workload comprising a mix of six streaming real-world applications.

and then updates them based on the extent of exploration, exploitation, learning rate, and reward function. These factors affect convergence and quality of the learned RL models.

Fewer than 20% of the experiments with RL converge to a stable policy and less than 10% of them provide competitive performance compared to the proposed IL-scheduler. We choose the RL solution that performs best to compare with the IL-scheduler. The Oracle generation and training parts of the proposed technique take 5.6 minutes and 4.5 minutes, respectively, when running on an Intel Xeon E5-2680 processor at 2.40 GHz. In contrast, an RL-based scheduling policy that uses the policy gradient method converges in 300 minutes on the same machine. Hence, the proposed technique is 30× faster than RL. As shown in Fig. 1.14, the RL scheduler performs within 11% of the Oracle, whereas the IL scheduler presents average execution time that is within 1% of the Oracle.

In general, RL-based schedulers suffer from the following drawbacks: (1) need for excessive fine-tuning of the parameters (learning rate, exploration rate, and NN structure), (2) reward function design, and (3)

slow convergence for complex problems. In strong contrast, IL policies are guided by strong supervision eliminating the slow convergence problem and the need for a reward function.

1.1.6.9 Complexity Analysis of the Proposed Approach

In this section, we compare the complexity of our proposed IL-based task scheduling approach with ETF, which is used to construct the Oracle policies. The complexity of ETF is $O(n^2m)$ [42], where n is the number of tasks and m is the number of PEs in the system. While ETF is suitable for use in Oracle generation (offline), it is not efficient for online use due to the quadratic complexity on the number of tasks. However, the proposed IL-policy which uses decision tree has the complexity of $O(n)$. Since the complexity of the proposed IL-based policies is linear, it is practical to implement in heterogeneous many-core systems.

1.1.7 Conclusions

Efficient task scheduling in heterogeneous many-core platforms is crucial to improve the system performance, but is very challenging due to its NP-hardness. In this work, we have presented an imitation learning based approach for task scheduling in many-core platforms executing streaming applications from wireless communications and radar systems. We have presented a hierarchical imitation learning framework that learns from an Oracle to develop task scheduling policies to minimize the execution time of applications. The framework has been evaluated comprehensively with six domain-specific applications and analyzed the storage and latency overheads of the IL policies. We have shown that the IL policies approximate the Oracle better than 99%. The overhead of the policies are significantly low at 1.1µs latency per scheduling decision and lower than the completely fair scheduler (1.2µs). Our IL policies achieve application

execution times within 9.3% of optimal schedules obtained offline using constraint programming. Preliminary experiments in which we have used IL to bootstrap RL for task scheduling in heterogeneous many-core platforms, show much faster convergence to optimal policies. We envision this work to initiate a new direction in scheduling studies with optimal Oracle generation and evaluation with applications from various domains.

1.2 Evaluation Frameworks for DSSoCs

1.2.1 Introduction

With the saturation of Moore's Law and emergence of new workloads, the ability of traditional homogeneous processors and single-ISA heterogeneous multicore architectures to satisfy the power and performance requirements of applications has saturated. Specialized and targeted implementations on graphical processing units (GPUs) and digital signal processors (DSPs) provide significantly improve the efficiency metrics. Homogeneous and single-ISA systems provide programming flexibility at the cost of energy efficiency, while special-purpose solutions trade-off computational efficiency for flexibility. Domain-specific system-on-chip (DSSoC) architectures, which are a realization of an advanced heterogeneous architectures, bridge the gap between programming flexibility and energy efficiency by smartly combining general-purpose, special-purpose and hardware accelerator cores. The special-purpose and hardware accelerator cores in DSSoCs strive to maximize the energy efficiency of applications in a targeted domain, and the general-purpose cores provide the programming flexibility of applications from unknown domains.

SoC architectures, and in particular DSSoCs, face monumental design and verification efforts due to rapidly increasing design sizes and complexity. For instance, Intel 4004 (the world's first microprocessor) uses merely 2300 transistors, and this number grew to 7.5 million transistors in Intel Pentium 2, and to 57 billion transistors in the Apple M1 Pro Max SoC [71, 30]. Designs also include multiple power domains to reduce the power consumption by utilizing techniques such as voltage scaling and power gating. For example, the Arm Cortex-A72 processor uses 4–8 power domains, and the Xilinx Zynq Ultrascale+ ZCU102 uses ~10 power rails [1, 6]. Furthermore, to facilitate clock isolation, gating and different frequencies, SoC designs use several clock domains thereby increasing

design complexity due to clock synchronization and domain-crossing protocols. DSSoCs also integrate several tens of IPs, and honoring the power, performance, interface and design constraints further convolute the design cycle [38]. Therefore, the design complexity and size of DSSoCs pose critical threats to design and verification life cycle, planning, cost, man effort, tools and time to market.

Functional and performance bugs and failures in these complex chips post fabrication result in unprecedented costs. The infamous Intel floating point divider failure (Pentium FDIV bug) forced Intel to recall the defective chips and resulted in a cost of $475 million [77]. More recently, two security vulnerabilities namely Meltdown and Spectre have initiated many lawsuits against major technology giants [2]. To circumvent the post-silicon bug and failure challenges, stringent pre-silicon verification techniques such as RTL simulation, gate-level simulation, formal verification, FPGA emulation and prototyping frameworks are used to identify and rectify bugs in the early design stages.

Simulation frameworks are extensively used for design validation and span across multiple stages of the design cycle. *First*, performance estimating simulation frameworks employ analytical models, transaction level modeling and bus functional models at the early design stages when microarchitects and hardware designers iterate to establish a baseline architecture. The accuracy of performance estimation critically relies on the accuracy of the analytical and functional models, which are typically improved over multiple iterations of the design. *Second*, hardware designers use RTL simulations for functional validation of the design. RTL simulations are highly suitable for design sizes that simulation and verification tools comfortably handle today. However, the tools demand tremendously high computing capacity for large designs (give example) while resulting in unreasonable simulation times (several days for few milliseconds of application simulation). *Finally*, gate-level simulations, which simulate

post-synthesis or post automatic, place and route (APR) netlists, are several orders of magnitude slower (give number from reference). Therefore, simulation techniques do not scale well with larger designs due to the challenges described above, and are utilized only for partitions of SoC designs.

Prototyping and emulation on FPGAs is highly promising as it allows orders of magnitude (four orders of magnitude) faster run-times for validation of the entire system at marginally higher development efforts [56, 41, 9]. A majority of works in literature utilize FPGA emulation to validate SoCs and evaluate key performance indicators [15, 86, 27]. Apart from better validation run-times, FPGA prototyping offers the following advantages: (1) Enables execution of real-world workload scenarios that otherwise consume unrealistic time and computational power for simulation, (2) enables full-system validation, (3) allows for early pre-fabrication firmware and software development, (4) improves the confidence of functional validation and performance indicators, and (5) improved time-to-market cycles. The use of rich and diverse processing elements in DSSoCs significantly increases the firmware and software development, apart from the design complexity.

1.2.2 Related Work

There are a large number of works on design space exploration for embedded systems, but they are found to be lacking in support for rich scheduling, thermal, and power optimization algorithms. Khalilzad et al. [45] consider a heterogeneous multiprocessor platform along with applications modeled as synchronous dataflow graphs and periodic tasks. The design space exploration problem is solved using a constraint programming solver for different objectives such as deadline, throughput, and energy consumption. ASpmT [69] proposes a multi-objective tool using Answer Set Programming (ASP) for heterogeneous platforms with

a grid-like network template and applications specified as directed acyclic graphs (DAGs). Trčka et al. [97] utilize the Y-chart [48] philosophy for design space exploration and introduces an integrated framework using the Octopus toolset [18] as its kernel module. Then, for different steps in the exploration process (i.e., modeling, analysis, search, and diagnostics), different languages and tools such as Ptolemy, Uppaal, and OPT4J are employed. Target platforms and applications are modeled in the form of an intermediate representation to support translation from different languages and to different analysis tools. Artemis [76] aims to evaluate embedded-systems architecture instantiations at multiple abstraction levels. Later, authors extend the work and introduce the Sesame framework [75] in which target multimedia applications are modeled as Kahn Process Network (KPN) written in C/C++. Architecture models, on the other hand, include components such as processor, buffers, and buses and are implemented in SystemC. The framework supports different schedulers such as first in, first-out (FIFO), round-robin, or customized. A trace-driven simulation is applied for cosimulation of application and architecture models.

Finally, ReSP [20] is a virtual platform targeting multiprocessor SoCs focusing on a component-based design methodology utilizing SystemC and transaction-level modeling libraries. ReSP adopts lower-level instruction set based simulation approach and is restricted to applications implemented in SystemC. All aforementioned frameworks or tools lack accurate power and thermal models, and do not support for exploration of scheduling algorithms and power-thermal management techniques. Several FPGA based emulation frameworks have also been designed to accelerate complex simulations, and also to prototype heterogeneous SoCs [56, 89, 86].

Outside of embedded systems, there has also been a large body of work on design space exploration via heterogeneous runtimes at the desktop or HPC scale, with StarPU [16] being one of the most prominent examples

of such a runtime. StarPU is a comprehensive framework that provides the ability to perform run-time scheduling and execution management for DAG based programs on heterogeneous architectures. Although, the framework allows users to develop new scheduling algorithms, StarPU lacks power-thermal models and DVFS techniques to optimize power and energy consumption. A recent work [107] targets domain-specific programmability of heterogeneous architectures through intelligent compile-time and run-time mapping of tasks across CPUs, GPUs, and hardware accelerators. In the proposed approach, the authors employ four different simulators, more specifically, Contech to generate traces, MacSim to model CPU/GPU architectures, BookSim2 to model the networks-on-chip, and McPat to predict energy consumption. The proposed DS3 simulator integrates the above features in a unified framework to benefit similar studies in the future. The FPGA framework is a full system solution to prototype general-purpose cores, accelerators, software, drivers and firmware.

1.2.3 DS3: Domain-Specific System-on-Chip Simulation Framework

In this section, we present DS3, a system-level domain-specific system-on-chip simulation framework. DS3 framework enables (1) run-time scheduling algorithm development, (2) DTPM policy design, and (3) rapid design space exploration. To this end, DS3 facilitates plug-and-play simulation of scheduling algorithms; it also incorporates built-in heuristic and table-based schedulers to aid developers and provide a baseline for users. DS3 also includes power dissipation and thermal models that enable users to design and evaluate new DTPM policies [87]. Furthermore, it features built-in dynamic voltage and frequency scaling (DVFS) governors, which are deployed on commercial SoCs. Besides providing representative baselines, this capability enables users to perform extensive studies to characterize a variety of metrics, PPW and EDP for a given SoC and set of

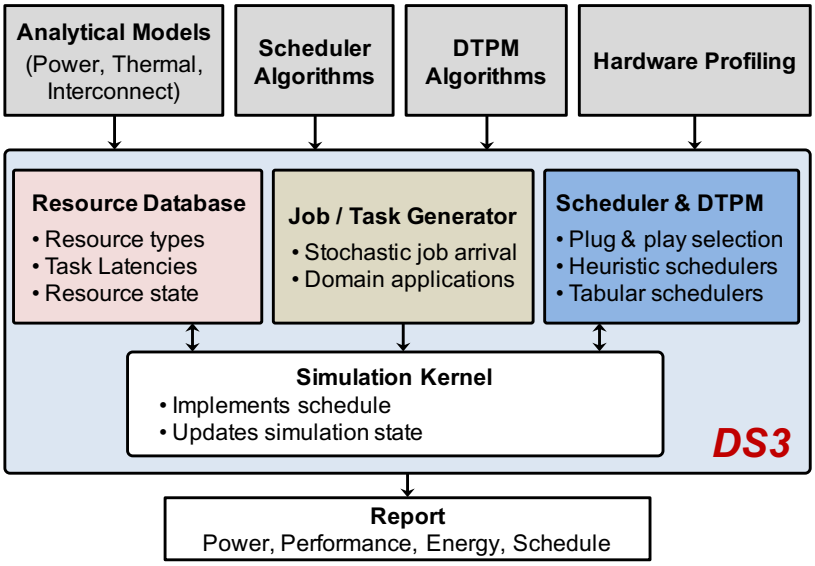


Figure 1.15: Organization of DS3 framework describing the inputs and key functional components to perform rapid design space exploration and validation.

applications. Finally, DS3 comes with *six reference applications* from wireless communications and radar processing domain. These applications are profiled on heterogeneous SoC platforms, such as Xilinx ZCU102 [7] and Odroid-XU3 [4], and included as a benchmark suite in DS3 distribution.

The goal of DS3 is to enable rapid development of scheduling algorithms and DTPM policies, while enabling extensive design space exploration. To achieve these goals, it provides:

- Scalability: Provide the ability to simulate instances of multiple applications simultaneously by streaming multiple jobs from a pool of active domain applications.
- Flexibility: Enable the end-users to specify the SoC configuration, target applications, and the resource database swiftly (e.g., in minutes) using simple interfaces.

- Modularity: Enable algorithm developers to modify the existing scheduling and DTPM algorithms, and add new algorithms with minimal effort.
- User-friendly Productivity Tools: Provide built-in capabilities to collect, report and plot key statistics, including power dissipation, execution time, throughput, energy consumption, and temperature.

Overview of the DS3 Framework: The organization of the DS3 framework designed to accomplish these objectives is shown in Figure 1.15. The resource database contains the list of PEs, including the type of each PE, capacity, operating performance points (OPP), among other configurations. By exploiting the deterministic nature of domain applications, the profiled latencies of the tasks are also included in the resource database. The simulation is initiated by the job generator, which generates application representative task graphs. The injection of applications in the framework is controlled by a random exponential distribution. The DS3 framework invokes the scheduler at every scheduling decision epoch with the list of tasks ready for execution. Then, the simulation kernel simulates task execution on the corresponding PE using execution time profiles based on reference hardware implementations. Similarly, DS3 employs analytical latency models to estimate interconnect delays on the SoC [61]. After each scheduling decision, the simulation kernel updates the state of the simulation, which is used in subsequent decision epochs. In parallel, DS3 estimates power, temperature and energy of each schedule using power models [21]. The framework aids the design space exploration of dynamic power and thermal management techniques by utilizing these power models and commercially used DVFS policies. DS3 also provides plots and reports of schedule, performance, throughput and energy consumption to help analyze the performance of various algorithms.

The life cycle of a task in DS3 is shown in Figure 1.16. The job generator constructs a task graph for each application. The tasks that are ready to

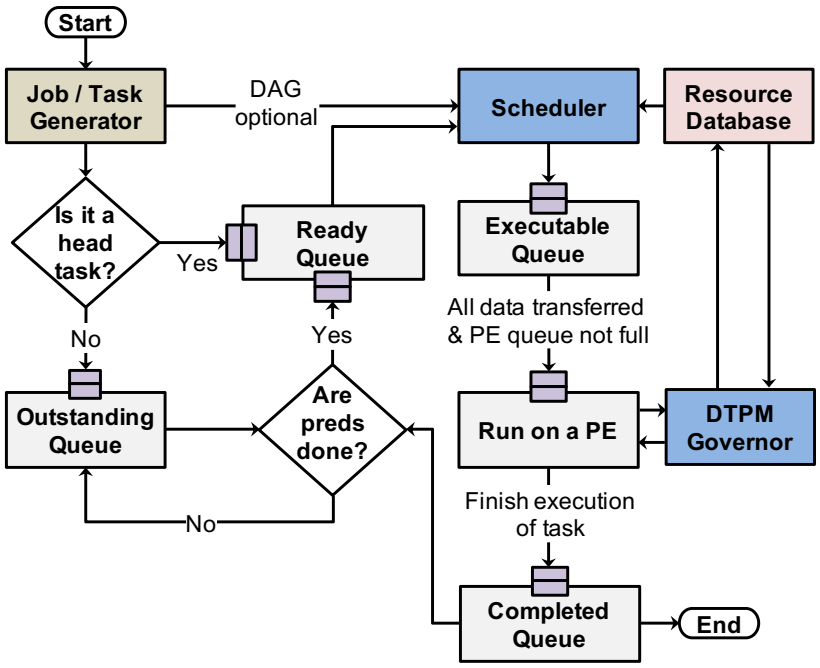


Figure 1.16: Life-cycle of a task in DS3 queues.

execute (i.e. free of dependencies) are moved to a *Ready Queue*. The other tasks that are waiting for predecessors to complete execution are held in the *Outstanding Queue* before being moved to the *Ready Queue*. The scheduler, an algorithm either built-in or user-defined, uses the resource database and produces PE assignments for ready tasks. Then, the simulation kernel migrates the tasks to the *Executable Queue* until communication requirements from predecessors are met. Finally, the task is simulated on the PE and retired after execution. The simulation kernel clears the dependencies imposed by these tasks and removes them from the system. If all the predecessors of a task waiting in the *Outstanding Queue* retire, the kernel moves them to the *Ready Queue*. This triggers a new scheduling decision and the tasks experience a similar life cycle in the framework, as described above. Memory and network are shared resources in an SoC.

The communication fabric performing high-speed data transfers among the various resources of the platform is assumed to be a mesh-based network-on-chip (NoC). We integrate analytical models to compute the latency at a given traffic load in a priority-aware mesh-based industrial NoC [60]. Executing multiple applications simultaneously leads to higher traffic in the network, as compared to the standalone execution. Hence, we account for the effect of a congestion in the network on execution time of applications. To model memory communication in the SoC, we include a bandwidth-latency model for memory latency modeling based on DRAMSim2 [81]. DRAMSim2 is used to obtain memory latencies at varying bandwidth requirements. DS3 models the transactions between the various communicating elements and keeps track of outstanding memory requests in a sliding window. We compute the memory bandwidth based on outstanding requests and then utilize the bandwidth-latency curve as a look-up table to obtain the average latency for the current memory bandwidth and add it to the execution time of the application(s). Hence, we account for contention of shared resources using the described network and memory models.

DS3 comes with *six reference applications* from wireless communications and radar processing domain: WiFi Transmitter (TX), WiFi Receiver (RX), low-power single-carrier TX, single-carrier RX, radar and pulse doppler. The WiFi protocol consists of compute-intensive blocks, such as FFT, modulation, demodulation, and Viterbi decoder (see Table 1.9), which require a significant amount of system resources. When the bandwidth and latency requirements are small, one can use a simpler single carrier protocol to achieve lower power consumption. Finally, we include two applications from the radar domain as part of the benchmark application suite - (1) range detection and (2) pulse Doppler (see Table 1.9).

The benchmark applications enable various algorithmic optimizations and realistic design space exploration studies, as we demonstrate in this

section. We will continuously include applications from other domains to the benchmarks. To demonstrate the capabilities of DS3, we use a typical heterogeneous SoC with a total of 16 general-purpose cores and hardware accelerators: 4 big Arm Cortex-A15 cores, 4 LITTLE Arm Cortex-A7, and 2 scrambler, 4 FFT, and 2 Viterbi decoder accelerators. We schedule and execute the WiFi TX/RX, range detection and pulse Doppler task flow graphs using DS3 and plot the average job execution time trend with respect to the job injection rate, as shown in Figure 1.17. We use the parameters p_{RX}, p_{TX}, p_{range}, and p_{pulse} representing the probabilities for the new job being WiFi-RX, WiFi-TX, range detection and pulse Doppler, respectively.

Figures 1.17(a) and (b) depict the results with WiFi applications for a download and upload intensive workload, independently. To understand the performance of scheduling algorithms, we analyze the average execution time at varying rates of job injection. We use three different schedulers for evaluation: (1) Minimum execution time (MET), (2) Integer linear programming (ILP), and (3) earliest task first (ETF). The minimum execution time is a low-overhead list-based scheduler which simply schedules a task to a resource that can execute it in the lowest amount of time. The ETF scheduler executes a task on a resource that also executes the task in the shortest amount of time, but it also takes into account the communication overheads from the predecessor tasks, and the availability of the resources. An ILP scheduler solves an optimization problem for one application with average execution time as the objective and takes into account all the constraints in the system such as the directed acyclic graph precedence constraints and heterogeneity constraints. It uses a static table-based schedule which is optimal for one job instance. At low injection rates (less than 1 job/ms), ILP is suitable as jobs do not interleave. However, as the injection rate increases, the ILP schedule is not optimal. ETF scheduler is superior in comparison to the others, as

observed in Figures 1.17(a),(b).

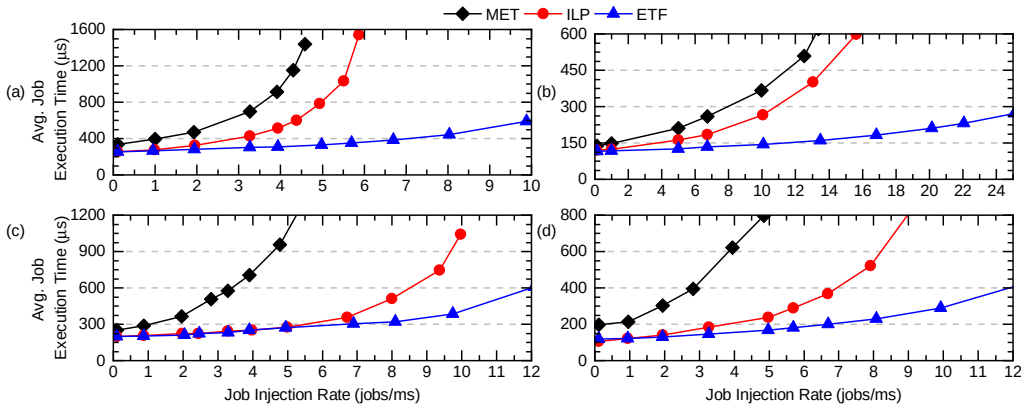


Figure 1.17: Results from different schedulers with a workload consisting of (a) WiFi-TX ($\mathbf{p_{TX}}$=0.2) and WiFi-RX ($\mathbf{p_{RX}}$=0.8), (b) WiFi-TX ($\mathbf{p_{TX}}$=0.8) and WiFi-RX ($\mathbf{p_{RX}}$=0.2), (c) range detection ($\mathbf{p_{range}}$=0.8) and pulse Doppler ($\mathbf{p_{pulse}}$=0.2), (d) WiFi-TX ($\mathbf{p_{TX}}$=0.3), WiFi-RX ($\mathbf{p_{RX}}$=0.3), range detection ($\mathbf{p_{range}}$=0.3), and pulse Doppler ($\mathbf{p_{pulse}}$=0.1).

Figure 1.17(c) demonstrates the results for a workload comprising radar benchmarks. This workload uses $p_{range} = 0.8$ and $p_{pulse} = 0.2$, owing to the difference in execution times of the two applications. The performance of ETF and ILP schedulers are similar until 5 jobs/ms, following which performance of ETF is superior in comparison to ILP. Although the trend in execution time for radar benchmarks is similar to WiFi, the job injection rate at which ETF and ILP diverge is different because of the differences in execution times of these applications. At an injection rate lower than 5 jobs/ms, the level of interleaving of jobs is low which aligns with the ILP solution.

Finally, we construct a workload comprising of four applications and Figure 1.17(d) shows the corresponding results. The performance trend of the schedulers with all applications is similar to WiFi and radar workloads. MET considers only the best performing PEs for mapping and ILP is sub-optimal at high injection rates whereas ETF utilizes the state information

of all PEs for mapping decisions. In summary, the experiments presented in Figure 1.17 demonstrate the capabilities of the simulation environment. DS3 allows the end user to evaluate workload scenarios exhaustively by sweeping the p_{TX}, p_{RX}, p_{range} and p_{pulse} configuration space to determine the scheduling algorithm that is most suitable for a given SoC architecture and set of workload scenarios. Readers may look into [14] for further details of DS3.

I would like to thank the collaborators – Samet Arda (Arizona State University), Alper Goksoy (University of Wisconsin-Madison), Nirmal Kumbhare (University of Arizona), Joshua Mack (University of Arizona), Anderson Sartor (Carnegie Mellon University), and Xing Chen (Arizona State University) for their contributions towards this work.

1.2.4 FALCON: An FPGA Emulation Platform for Domain-Specific Systems-on-Chip (DSSoCs)

In literature, FPGAs have extensively been used for network-on-chip (NoC) emulation, SoC prototyping and performance evaluation of novel special-purpose architectures. However, there does not exist an end-to-end framework for emulation and prototyping of DSSoCs. To address these challenges, we propose an end-to-end FPGA based emulation framework for prototyping and performance evaluation of DSSoCs. Since DSSoCs integrate a number of compute and on-chip elements such as general-purpose cores, special-purpose cores, hardware accelerators, caches and interconnect, it is crucial to ensure the different aspects such as hardware, software, firmware and drivers are well validated before the chip fabrication to avoid the expensive cost of identifying post-silicon bugs and failures. To maximize the energy efficiency, microarchitects and designers typically prefer to iterate over multiple versions of hardware architectures for a particular task, as well as accelerators for different tasks in the applications. To simplify the development process, the framework includes an accelerator

sandbox which standard AMBA based interfaces to the rest of the DSSoC. The accelerator sandbox critically improves the productivity of developers by providing a plug-and-play environment to include, remove and modify hardware accelerators, thereby improving the design time cycles by several orders of magnitude. While the performance of hardware accelerators can be analyzed standalone, the proposed emulation framework allows their evaluation from the full system perspective where cache, memory and interconnect behaviors can strongly influence their key performance indicators. Hardware accelerators define their own interfacing mechanisms to the rest of the SoC, and also provide their programming sequence. The proposed framework also allows DSSoC designers to develop drivers for such non-standard ISA designs before the chip becomes available. Finally, the framework also enables software and firmware development which include aspects such as the boot firmware, operating system bundles and device management.

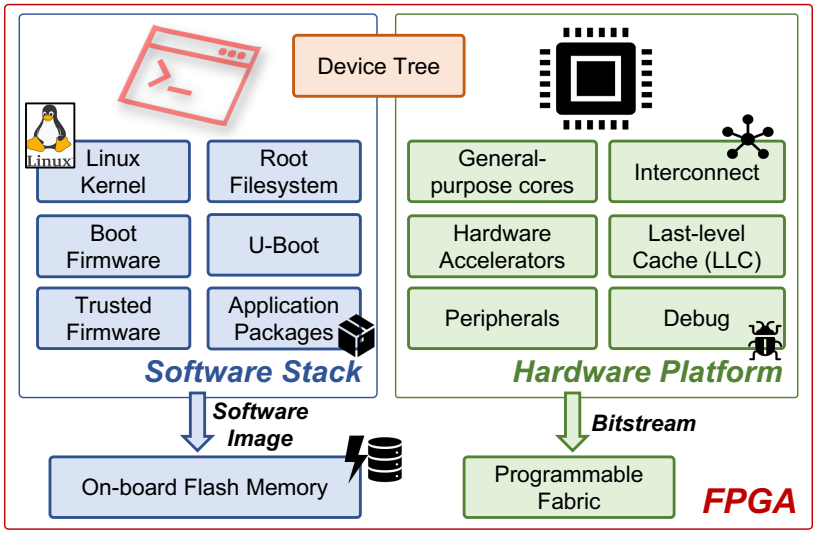


Figure 1.18: An overview of the key components and organization of the FALCON framework for DSSoC emulation.

This section presents FALCON's full-system architecture for DSSoC design and emulation, as outlined in Figure 1.18. Broadly, FALCON comprises the hardware platform and the software stack. While these components are typical in any SoC, DSSoCs are highly customized to include specialized processing elements and optimized runtime frameworks to maximize the energy-efficiency of domain applications, and are described in the later portions of this section. The hardware platform integrates general-purpose cores that offer programmability, hardware accelerators and specialized cores for energy efficiency, a high-speed interconnect for low-latency on-chip data movement, last-level cache (LLC), peripherals and debug logic [62]. The entire hardware architecture is packaged a bitstream to program to the programmable logic on the FPGA after synthesis and automatic place-and-route. The software stack which comprises the Linux operating system (OS) kernel, filesystem and embedded system software components such as the boot firmware and U-boot. All the components in the software stack are integrated and built into a software image, which is programmed to the flash memory on the FPGA. Applications can be executed on the underlying hardware of the DSSoC with the use of software runtimes such as CEDR and SPARTA [57].

As we emphasized earlier, FALCON enables the evaluation of resource management algorithms, functional validation, early software and firmware development. Furthermore, it also allows us to obtain realistic estimates of DSSoC performance that we can use to calibrate the power and performance estimates used in DS3 for high-level design space exploration. To this end, FALCON allows us to develop two different paths to move data between general-purpose cores and hardware accelerators. Conventionally, the main memory (DDR) allocates a memory space to server as a buffer for the core and accelerator to exchange data. Enabling coherency in the DSSoC improves the chances of the required data to be fetched from on-chip caches. However, as the size of the data to be moved in-

creases, it fails to sit in the cache, still requiring the need for main memory accesses. In FALCON, we provision for an on-chip scratchpad memory, which allows us to minimize the latency spent in sending and receiving data from the main memory. We present the execution time results for FFT and matrix multiplication in hardware accelerators that utilize the two different datapaths, as shown in Figure 1.19. First, enabling coherency when the data is moved through main memory significantly improves the latency by ~25% on average. For smaller data movement sizes (128-point FFT), scratch and main memory based data movement result in similar latencies. However, as the amount of data to be moved into the accelerator increases (512-point FFT and $(4\times64)\times(64\times4)$ matrix multiplication), moving the data through the scratchpad results in 20% improvement in execution time for 512-point FFT and 44% improvement for the matrix multiplication operation. Therefore, FALCON allows us to obtain realistic performance estimates of the DSSoC, functional validation and early software development.

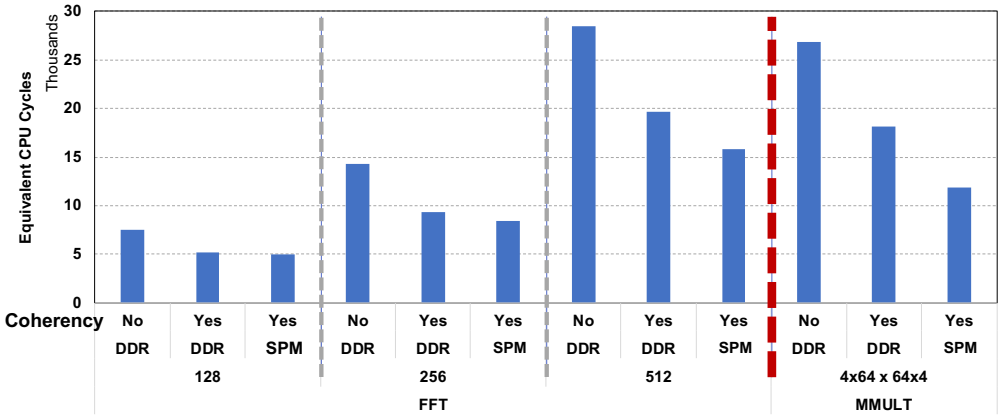


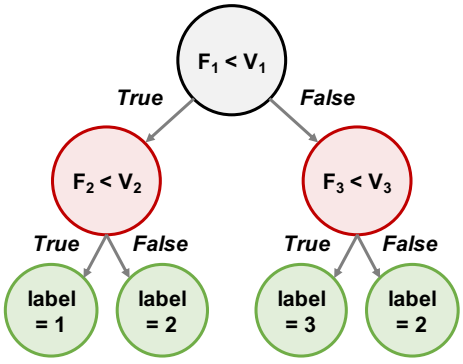
Figure 1.19: A comparison of end-to-end execution time for FFT (various transform sizes) and matrix multiplication with two different datapaths (through main memory and through on-chip scratchpad).

I would like to thank the collaborators – Tutu Ajayi (University of

Michigan – Ann Arbor), Hanguang Yu (Arizona State University), Vishrut Pandey (University of Wisconsin-Madison), Joshua Mack (University of Arizona) and Sahil Hassan (University of Arizona) for their contributions towards this work.

1.2.5 Optimization of Decision Tree Classifiers

Decision trees, traditionally used as machine learning (ML) classifiers in data mining, are gaining traction in resource management algorithms in systems-on-chip [91, 87, 50]. A decision tree (DT) is characterized by a tree with conditional statements at the root and internal nodes, which evaluate either True or False. Finally, each leaf node denotes an outcome, i.e., the output decision, as illustrated in Figure 1.20. DTs are control-flow oriented, with a maximum of **D** branching instructions for a tree of maximum depth **D**. The simple set of conditions make them understandable and easy to use [91].




```
function decision_tree :
  # Input: FEATURES(F1,F2,F3)
  if F1 < V1 :
    if F2 < V2 :
      label = 1
    else :
      label = 2
  else :
    if F3 < V3 :
      label = 3
    else :
      label = 2
return label
```

Figure 1.20: An illustration of a decision tree of maximum depth 2 (*left*) and its corresponding pseudo code for classification (*right*).

Decision trees are commonly used in applications with widely varying requirements. While applications, such as data mining and bioinformatics, use ensembles and classify large datasets, decision trees used for resource

management in embedded devices target ultra-low latency [91, 87, 50]. DTs can be implemented using both software (using sequential code execution) and hardware accelerators to satisfy the different requirements. While hardware architectures offer a high degree of parallelism (hence, throughput), software approaches provide the following advantages: (1) re-use of existing CPU cores, (2) avoid the complexity of hardware design and integration, (3) reduce data movement on the interconnect, and (4) eliminate data and control handoff overheads between CPU and accelerator. Data mining and bioinformatics applications benefit from the parallelism in hardware implementations since they classify large datasets [91]. In contrast, decision tree classifiers used in resource management applications use a singular decision tree invoked periodically and target ultra-low latency (tens of nanoseconds) [87, 50]. Therefore, the nature of the application plays a crucial role in selecting between execution in software and hardware.

This section focuses on DT classifiers used for resource management applications. While hardware architectures for decision trees have shown substantial speed-ups over software approaches in the literature, most approaches do not consider the data transfer overheads between the host and hardware accelerator [91]. Considering these overheads is crucial for low-latency applications since the speed-up obtained with hardware accelerators is a function of the amount of data to be moved [94]. Moreover, the data movement and control overheads may be prohibitively high (typically in microseconds) to achieve low latency, especially when the classification operation is a handful of instructions in a software program. Existing software approaches do not use fixed-point data and also do not accurately profile code that executes in the order of nanoseconds. Hence, we propose both hardware and software implementation and optimization techniques to accelerate the inference of classification on decision tree classifiers.

Pipelined Hardware Architecture for Decision Tree

A fully pipelined hardware accelerator architecture for the decision tree is presented in Figure 1.21. Each stage of the decision tree requires one pipeline stage. Currently, we provision for up to 32 input features. The feature addresses for each stage and the coefficients are stored in a small internal memory in the pipeline stage. Each stage utilizes the features per the addresses and performs the computation for each level of the decision tree. The memory access takes one cycle, and the address computation takes another cycle, in effect, it is two cycles per pipeline stage. In total, it is 12 cycles for a decision tree of depth 6, and successive pipeline stages can operate on different set of input features, pertaining to scheduling different tasks.

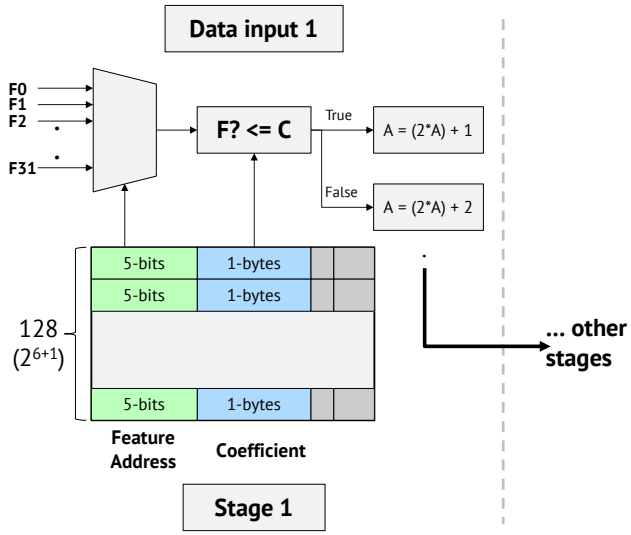


Figure 1.21: Microarchitecture of a pipelined hardware accelerator for classification using a decision tree classifier.

Latency Analysis of Hardware Architecture: Now, we present a breakdown at the expected performance of the decision tree in hardware. Each pipeline stage takes 2 cycles, one for memory access and one for compari-

son and address computation. It is 12 cycles latency within the accelerator (for a depth of 6), and with full pipelining, the hardware provides an output every other cycle. Right now, there are up to 32 inputs which are 8 bits each and hence, a total of 256 bits, which has to be sent in two cycles, since the CMN-600 bus width to the accelerators is 128 bits. To hide the latency, we send the critical word first such that the hardware pipeline isn't stalled. The total latency is about 34 cycles considering the latency within the accelerator, CMN-600 round trip latency and the Arm latency to dispatch the commands to the accelerator. At a frequency of 1 GHz, the latency of 34 cycles translates to 34 nanoseconds.

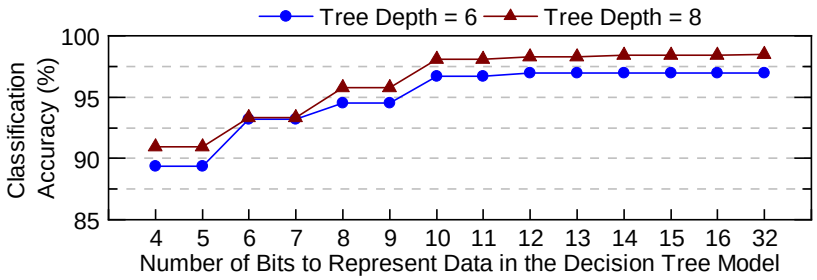


Figure 1.22: Effect of number of bits on the classification accuracy (Fixed-point: 4–16 bits, single-precision floating point: 32 bits).

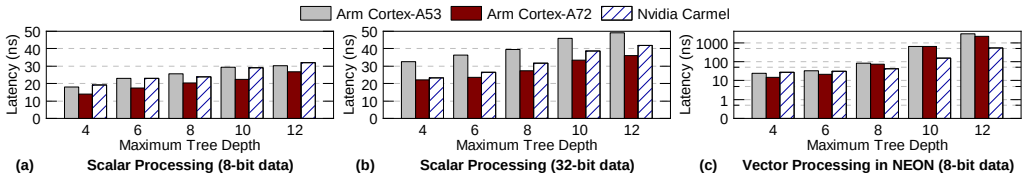


Figure 1.23: Latencies of decision tree classifiers for varying tree depths on Arm Cortex-A53, Arm Cortex-A72 and Nvidia Carmel cores with (a) scalar processing using 8-bit data, (b) scalar processing with floating-point (32-bit) data, and (c) vector processing in NEON using 8-bit data.

Software Optimization Techniques for Decision Tree Classification

This section describes the software optimization techniques to enhance the latency of DTs. We use Scikit-learn [74] to design the DTs used in this work. Then, the `sklearn-porter` tool translates the rules of the decision tree to generate a **C**-language implementation [68], which is our baseline. Finally, we propose optimization techniques that improve its performance, listed as follows.

Fixed-point number representation: The inherent robustness in ML models allows us to reduce computational complexity by using fewer bits for data representation [54]. We designed a DT optimizer that parses the baseline **C**-language implementation to transform the single-precision floating-point (FP) data types into a parameterized fixed-point representation. Classification accuracy of ~95% is achieved with an 8-bit data format with a marginal impact to accuracy, as shown in Figure 1.22. Using 8-bit numbers reduces the size of the model and, subsequently, the memory footprint by 75%. This transformation also allows for processing in integer functional units in the processors, which incur lower latency than FP units.

Transformation of Decision Variable: The default **C**-code generated by `sklearn-porter` uses weighted arrays for the possible output labels. The final decision label is computed by performing an argmax operation on the array. This approach has a memory footprint (due to the array) and latency (due to the argmax operation). Our decision tree optimizer transforms the weighted array assignments (performs argmax operation) to a single variable that directly holds the decision label, as shown in Figure 1.20.

Implementation on Arm NEON SIMD co-processor: We exploit the presence of a tightly coupled vector extension (NEON), available in most Arm-based processor architectures, to leverage the parallelism they offer [66]. The NEON extension performs SIMD processing on 128-bit registers that can be fragmented into 16 lanes of 8-bits each. This design choice nicely fits our 8-bit data format and allows for maximum parallel processing. We utilize the `VCLE` instruction to perform up to 16 comparisons simultane-

ously. The designed decision tree optimizer transforms the **C**-code into a NEON-friendly code, and the implementation is outlined in Figure 1.24.
Profiling and Setup: The key functional component in a DT classifier comprises a handful of instructions. Thus, even high-resolution timers cannot capture the latency of a single classification call accurately. Hence, we measure the execution time by repeatedly calling the classifier to obtain the average time per classification invocation. We use the `GCC` toolchain and compile the programs with the `-O3` flag to enable the highest level of compiler optimization.

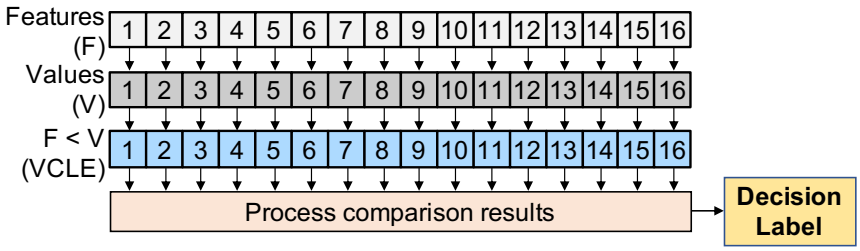


Figure 1.24: Transformation of sequential decision tree code into NEON-friendly code for SIMD processing.

Experimental Results of Software Optimization Techniques: We evaluate the proposed DT classifiers using a dataset for dynamic resource management of a system-on-chip (SoC) [50, 51]. The dataset comprises task scheduling decisions for the target SoC that uses 16 processing cores, organized into five processing clusters. Two clusters consist of Arm big.LITTLE cores, while the others have signal processing hardware accelerators. The proposed optimization techniques are evaluated on a DT that schedules tasks to these clusters. Figure 1.23 presents the latency of the DT classifier for various tree depths on Arm Cortex-A53 at 1.2 GHz, Arm Cortex-A72 at 1.5 GHz, and Nvidia Carmel cores at 1.9 GHz on Xilinx Zynq UltraScale+ SoC, Raspberry Pi 4, and Nvidia Jetson Xavier NX, respectively. The latency improves on average by ~35% by converting 32-bit data (Fig-

ure 1.23(b)) format to 8-bits (Figure 1.23(a)). The latency for a tree depth of 4 is similar with scalar processing (18.1 ns) and SIMD processing in NEON (23.98 ns). However, as tree depths increase, the latency is substantially higher with processing in the NEON co-processor. Upon a detailed characterization, we observed that data transfer between the scalar core and the NEON unit contributes to ~60% of the latency. The rest of the overhead comes from additional comparisons performed to allow any branch to be taken in the tree. So, for highly control-oriented programs such as DT classifiers, scalar cores provide better latency due to lower computation and data movement overheads. These results show the critical need to analyze the interplay between control flow, parallelizable vector operations, and data movement when designing latency-sensitive kernel tasks.

Summary of Results: This section presented optimization techniques for DT classification. The proposed software optimization techniques achieve lower than 50 ns decision time for trees up to a depth of 12, making them highly suitable for resource management in embedded devices. Our experiments also show that the data transfer overhead between a scalar processor and its tightly coupled vector processor (Arm NEON) belittles the parallelism that NEON offers. Furthermore, the data transfer and software programming overheads with a hardware accelerator overshoot the benefits of special-purpose hardware. We conclude that the proposed software optimization techniques for decision tree classification provide the best latencies (< 50ns) for trees of up to depth 12.

Table 1.9: Execution time profiles of applications on Arm A53 core in Xilinx ZCU102, Arm A7/A15 cores in Odroid-XU3, and hardware accelerators

Application	Task	Latency (µs)			
		Zynq A53	Odroid A7	Odroid A15	HW Acc.
WiFi TX	Scrambler-Encoder	22	22	10	8
	Interleaver	8	10	4	
	QPSK Modulation	15	15	8	
	Pilot Insertion	4	5	3	
	Inverse-FFT	225	296	118	16
	CRC	5	5	3	
WiFi RX	Match Filter	15	16	5	
	Payload Extraction	5	8	4	
	FFT	218	290	115	12
	Pilot Extraction	4	5	3	
	QPSK Demodulation	79	191	95	
	Deinterleaver	10	16	9	
	Decoder	1983	1828	738	2
	Descrambler	2	3	2	
Pulse Doppler	FFT	30	35	15	6
	Vector Multiplication	30	100	35	
	Inverse-FFT	30	35	15	6
	Amplitude Computation	25	70	40	
	FFT Shift	6	7	3	
Range Detection	LFM Waveform Generator	20	90	60	
	FFT	68	150	60	30
	Vector Multiplication	52	75	60	
	Inverse-FFT	68	150	60	30
	Detection	10	20	20	

2 PROPOSED WORK - 1: ONLINE TRAINING OF DECISION TREE CLASSIFIERS FOR TASK SCHEDULING IN DSSOCS

DSSoCs comprise a vast number and type of computing resources on-chip and the ability to efficiently exploit their potential at run-time to execute multiple simultaneous applications raises the need for efficient task scheduling. For instance, a feature detection algorithm in autonomous driving applications can be executed by both GPU and a detection accelerator. However, its actual execution resource is determined by several run-time parameters such as its execution time profile on different resources, resource utilizations and their earliest availability. Several task scheduling algorithms explore the wide search space offline to find the most efficient execution resource for a given task either by solving optimization problems, utilizing heuristic approaches or machine-learning techniques [47, 96, 50].

Task scheduling policies designed offline [47, 96] are typically optimized to a particular optimization objective (such as performance or energy consumption), SoC configuration and set of applications. However, SoC architectures, applications and hardware accelerators are rapidly evolving to adapt to the performance and energy efficiency needs of DSSoCs. For instance, SoC designers and developers analyze new applications in a particular domain and identify the need for new hardware accelerators and special-purpose processors to improve the energy efficiency of domain applications [28, 79]. The newer 5G applications pose higher performance hardware requirements to provide high-speed wireless data transfers [100], than other domain applications such as single-carrier and WiFi. Furthermore, SoC architects update the general-purpose cores in SoCs frequently as they continue to evolve rapidly incorporating novel microarchitectures and technology process nodes [46]. Newer microprocessor cores differ in power and performance characteristics than

their previous generation counterparts thereby rendering the scheduling policies developed for the previous generations obsolete. SoCs are also used in conflicting optimization objective requirements such as performance, power and energy consumption [29]. Therefore, there is a strong need for scheduling policies in SoCs to be updated to be in-tune with the evolution of the SoC and the end-user applications, as outlined in Figure 2.1.

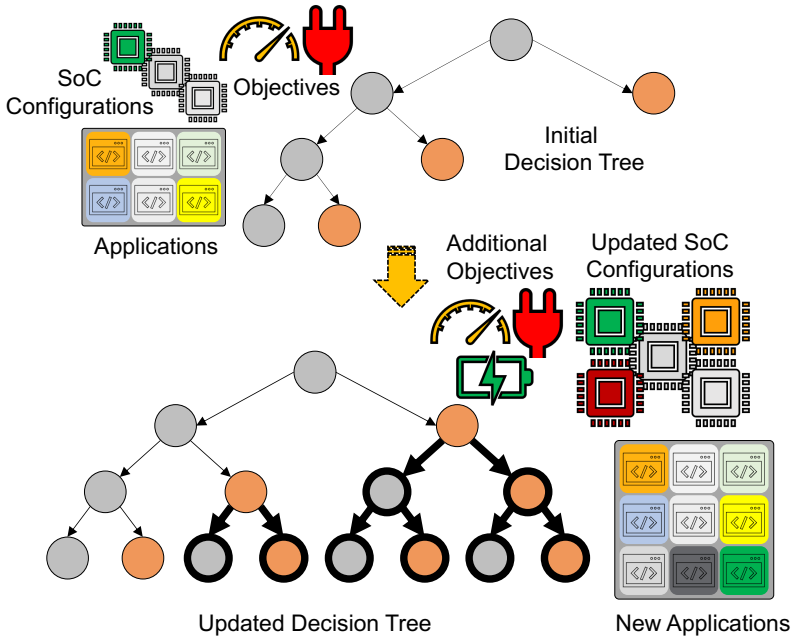


Figure 2.1: Decision tree based scheduling policies trained offline (top), and the decision tree updated incrementally for newer generation SoCs, applications and objectives (bottom).

In our preliminary work on task scheduling using imitation learning for DSSoCs, the decision tree based scheduling policy must be updated online with change in SoC configuration and applications. In general, online updates and training are performed using two techniques: (1) analytical models, and (2) reinforcement learning [58, 64]. In this case, the com-

plexity in formulating analytical models for a NP-complete problem such as task scheduling in DSSoCs makes them an unfavorable choice. Reinforcement learning (RL) is widely used for online training and adaptation scenarios [64, 58]. However, there are several complexities in deploying RL for task scheduling in DSSoCs. First, the design of a reward function is complex, and plays a crucial role in the rate of convergence with RL [58]. Second, the state space required to accurately represent the system state is extremely large for a DSSoC dealing with streaming job arrivals and a large number of processing elements, and converging towards the optimal solution involves impractical runtimes [50]. Finally, decisions trees are typically trained in entire batches of datasets, unlike neural networks that use stochastic gradient to incrementally update the weights of the network. While the last challenge can be solved by utilizing differentiable decision trees [3], the rest of the challenges make RL a practically less feasible solution for online training of decision trees.

Algorithm 3: Algorithm for Training of Decision Tree Classifiers

```
best_gain = 0
foreach node n in tree do
    foreach f ∈ F do
        foreach unique value v of f in dataset do
            Gini score, G_{n,f} = 1 - ∑_{i=0}^{C} P_i^2
            Information gain, I_{n,f} = G_{n,f} - G_{n,f,true_branch} - G_{n,f,false_branch}
            if G_{n,f} < best_gain then
                best_info_gain = G_{n,f}
                best_feature = f
                best_threshold = v
            end
        end
        if best_info_gain = 0 or maximum depth reached then
            Terminate that branch of tree
        end
    end
end
Employ decision tree pruning techniques to reduce the number of decision
  nodes and branches
```

To address the above described challenges, we propose to develop an algorithm to incrementally update decision trees. We denote the original training data as $\mathcal{X}_{\mathtt{orig}}$ and the corresponding tree trained with this data as $\mathcal{T}_{\mathtt{orig}}$. After we obtain $\mathcal{T}_{\mathtt{orig}}$, we receive an incremental training dataset denoted by $\mathcal{X}_{\mathtt{inc}}$. The full dataset is marked by $\mathcal{X}_{\mathtt{full}}$, where $\mathcal{X}_{\mathtt{full}} = \mathcal{X}_{\mathtt{orig}} \cup \mathcal{X}_{\mathtt{inc}}$. Let the tree trained with full dataset $\mathcal{X}_{\mathtt{full}}$ be called $\mathcal{T}^*$. We propose an algorithm to achieve $\mathcal{T}^*$, given that we use $\mathcal{T}_{\mathtt{orig}}$ as the starting point and only using $\mathcal{X}_{\mathtt{inc}}$. The incremental training approach exploits the information embedded into the tree when trained with $\mathcal{X}_{\mathtt{orig}}$. Furthermore, we embed additional information to the tree to allow us to achieve identical trees with only incremental training. To understand this process better, let us understand the training algorithm of a decision tree. From Algorithm 3, we understand that the Gini score and information gain of each node in the decision tree is computed using every unique value of every feature in the dataset, demanding that the tree is trained only as an entire batch. To modify this algorithm to allow for incremental training, we propose to store meta-information comprising a small subset of Gini scores, features and thresholds. The meta-information may be exploited in future training iterations which contain incremental data for which the decision tree needs to be updated. As part of the proposed work, we plan to fully evaluate the proposed algorithm along with extensive experimental results to substantiate the claims.

3 PROPOSED WORK - 2: AN INTEGRATED SYSTEM-ON-CHIP AND NETWORK-ON-CHIP POWER MANAGEMENT TECHNIQUE FOR SOCS

SoCs should be designed to meet aggressive performance requirements while coping with limited battery capacity, thermal design power, and real-time constraints. A step in this direction consists of exploiting heterogeneity, e.g., using big cores when high performance is needed and switching to little cores otherwise. Furthermore, DSSoCs integrate special-purpose and hardware accelerator cores to maximize the energy efficiency of applications in a specific domain. Techniques such as dynamic voltage-frequency scaling and power gating can be used at runtime to manage the power consumption of SoCs. However, the design space of runtime decisions explodes combinatorially with the number of cores, frequency levels, and power states. Additionally, DSSoCs serve a wide range of applications with distinct characteristics and requirements. The extensive design space and the growing variety of applications demand highly efficient runtime techniques to efficiently manage the power and performance of DSSoCs [58, 87, 80, 53].

DSSoCs that optimize the on-chip data communication latencies employ networks-on-chip (NoCs). NoCs offer substantially better latencies (1 cycle per hop, leading to 5-6 cycle latency for a mesh of upto 4×4) than older generation crossbar interconnects (around ~100s of cycles) [5, 62]. NoCs achieve the ultra-low latency at the expense of power consumption, and therefore, there is a strong need for dynamic frequency scaling techniques to minimize the power and energy consumption in NoCs [110, 10].

Several state-of-the-art techniques address the dynamic voltage and frequency scaling aspects of the processing elements in SoCs and NoCs independently to minimize the energy consumption at runtime [80, 53,

110]. In reality, the frequency tuning of the processing elements and NoCs cannot be separated since application execution is a strong function of both of these aspects. For instance, lowering only the frequency of the processing elements forces the NoC to move data on-chip faster than they are actually required to. On the contrary, NoC operating at a low frequency makes the processing elements to wait for the data to arrive. Therefore, there is a strong need for coordinated dynamic voltage and frequency scaling techniques to jointly minimize the power and energy consumption with negligible impact to performance.

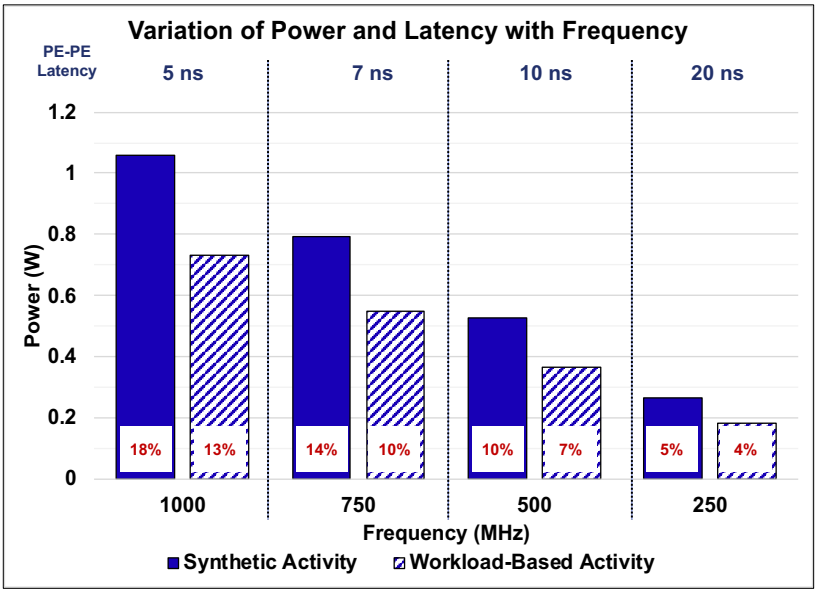


Figure 3.1: Plot showing the total SoC power and latency between two PEs as a function of NoC frequency for synthetic and real workload based activity. The percentage value in each column denotes the fraction of NoC power as a function of the SoC power.

To understand the impact of NoC frequency on total SoC power and worst case latencies between two PEs in the SoC, we sweep the NoC frequencies and obtain latency and power estimates as outlined in Figure 3.1. We observe that the NoC power reduces by ~3× as the frequency drops

from 1 GHz to 250 MHz, with a increase in PE-PE latency from 5 ns to 20 ns. Therefore, we propose to develop an integrated SoC and NoC DVFS technique to exploit the potential of DSSoCs while minimizing the energy consumption. As emphasized earlier, the design space of runtime decisions is large if we consider tuning the frequency of only the PEs. The problem is further complicated by adding the NoC to the design space. To explore this vast space efficiently, we propose to utilize Monte Carlo tree search algorithms together with reinforcement learning to obtain the optimal runtime frequencies that minimize the overall power and energy consumption of SoCs and NoCs.

4 CONCLUSION OF THE REPORT

In this report, we introduce the need for DSSoCs to overcome the limitation of today's hardware in maximizing energy efficiency. While DSSoCs integrate a number of general-purpose, special-purpose and hardware accelerator cores to accelerate applications in a specific domain, there is a critical need for efficient scheduling techniques to manage the resources at runtime and exploit the potential of DSSoCs. Task scheduling in DSSoCs is key to improve the system performance but is very challenging due to its NP-hardness. In this report, we presented an imitation learning based approach for task scheduling in many-core platforms executing streaming applications from wireless communications and radar systems. We have presented a hierarchical imitation learning framework that learns from an Oracle to develop task scheduling policies to minimize the execution time of applications. The framework is evaluated comprehensively with six domain-specific applications and analyzed the storage and latency overheads of the IL policies. We showed that the IL policies approximate the Oracle better than 99%. Our IL policies achieve application execution times within 9.3% of optimal schedules obtained offline using constraint programming. Furthermore, we propose optimization techniques for decision tree classification and achieve a latency of less than 50 nanoseconds for trees up to a depth of 12, making them highly suitable for resource management in embedded devices.

The design and development phase of DSSoCs raises the strong need for tools, evaluation and emulation frameworks to explore the vast design space, evaluate scheduling algorithms and functionally validate them. To address this challenge, we presented DS3, a Python-based open-source system-level framework for rapid design space exploration of domain specific SoCs. We developed a scalable, modular, and flexible simulation framework to evaluate scheduling algorithms, dynamic power-thermal

management techniques and architecture exploration. We also presented benchmark applications, built-in scheduling algorithms and DVFS policies which can be used as reference by users and developers. Finally, the framework is thoroughly validated against a commercial SoC, asserting the fidelity of the simulator to successfully simulate domain-specific SoCs. We also developed FALCON, a FPGA based emulation platform that deploys general-purpose cores, hardware accelerators and a mesh-based network-on-chip interconnect. Together with software runtime frameworks, FALCON can be used to implement efficient scheduling techniques and maximize the energy efficiency of domain applications.

As future work, we plan to explore techniques to update decision tree classifiers online to adapt to evolving applications, SoC configurations and new optimization objectives. Another area of focus is to develop integrated dynamic power management techniques that utilize voltage and frequency scaling to minimize power and energy consumption with negligible impact to application performance.

BIBLIOGRAPHY

[1] Arm Cortex-A72 Technical Reference Manual. https://developer.arm.com/documentation/100095/0003, Accessed 1 January 2022.

[2] Arm Cortex-A72 Technical Reference Manual. Spectre and Meltdown are Now a Legal Pain for Intel, the Chip Maker Faces 35 Lawsuits over the Attacks., Accessed 1 January 2022.

[3] Interpretable Reinforcement Learning via Differentiable Decision Trees, author=Rodriguez, Ivan Dario Jimenez and Killian, Taylor W and Son, Sung-Hyun and Gombolay, Matthew C, year=2019.

[4] ODROID-XU3. `https://wiki.odroid.com/old_product/odroid-xu3/odroid-xu3` Accessed 3 Mar. 2019.

[5] White Paper on Interconnect Solutions from Cadence. https://ip.cadence.com/uploads/251/white-paper-interconnect-solutions-debugging-issues-advanced-ARM-CoreLink-pdf, Accessed 20 January 2022.

[6] Zynq Ultrascale+ ZCU102 Technical Reference Manual. https://www.xilinx.com/support/documentation/user_guides/ug1085-zynq-ultrascale-trm.pdf, Accessed 1 January 2022.

[7] Zynq ZCU102 Evaluation Kit. https://www.xilinx.com/products/boards-and-kits/ek-u1-zcu102-g.html, Accessed 10 April 2020.

[8] V12.8: User's Manual for CPLEX. *International Business Machines Corporation*, 46(53):157, 2009.

[9] Y. Abarbanel, E. Singerman, and M. Y. Vardi. Validation of SoC Firmware-hardware Flows: Challenges and Solution Directions. In *Proceedings of the Design Automation Conference*, pages 1–4, 2014.

[10] A. Abbas, M. Ali, A. Fayyaz, A. Ghosh, A. Kalra, S. U. Khan, M. U. S. Khan, T. De Menezes, S. Pattanayak, A. Sanyal, et al. A Survey on Energy-efficient Methodologies and Architectures of Network-on-chip. *Computers & Electrical Engineering*, 40(8):333–347, 2014.

[11] A. M. Aji, A. J. Peña, P. Balaji, and W.-c. Feng. MultiCL: Enabling Automatic Scheduling for Task-Parallel Workloads in OpenCL. *Parallel Computing*, 58:37–55, 2016.

[12] Apple. Apple M1 SoC. `https://www.apple.com/newsroom/2021/10/introducing-m1-pro-and-m1-max-the-most-powerful-chips-apple-has-ever-built` Accessed 20 January 2022, 2021.

[13] 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.

[14] S. E. Arda et al. DS3: A System-Level Domain-Specific System-on-Chip Simulation Framework. *IEEE Trans. on Computers*, 69(8):1248–1262, 2020.

[15] D. Atienza, P. G. Del Valle, G. Paci, F. Poletti, L. Benini, G. De Micheli, and J. M. Mendias. A Fast HW/SW FPGA-based Thermal Emulation Framework for Multi-processor System-on-chip. In *Proceedings of the 43rd annual Design Automation Conference*, pages 618–623, 2006.

[16] C. Augonnet et al. Starpu: a unified platform for task scheduling on heterogeneous multicore architectures. *Concurrency and Computation: Practice and Experience*, 23(2):187–198, 2011.

[17] S. Baskiyar and R. Abdel-Kader. Energy Aware DAG Scheduling on Heterogeneous Systems. *Cluster Computing*, 13(4):373–383, 2010.

[18] T. Basten et al. Model-driven design-space exploration for embedded systems: The octopus toolset. In *Int. Symp. On Leveraging Applications of Formal Methods, Verification and Validation*, pages 90–105. Springer, 2010.

[19] T. Beisel, T. Wiersema, C. Plessl, and A. Brinkmann. Cooperative Multitasking for Heterogeneous Accelerators in the Linux Completely Fair Scheduler. In *IEEE International Conference on Application-Specific Systems, Architectures and Processors*, pages 223–226, 2011.

[20] G. Beltrame, L. Fossati, and D. Sciuto. Resp: a nonintrusive transaction-level reflective mpsoc simulation platform for design space exploration. *IEEE Trans. Comput.-Aided Des. Integr. Syst.*, 28(12):1857–1869, 2009.

[21] G. Bhat et al. Algorithmic Optimization of Thermal and Power Management for Heterogeneous Mobile Platforms. *IEEE Trans. Very Large Scale Integr. (VLSI) Syst.*, 26(3):544–557, 2018.

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

[23] J. Castrillon et al. Communication-Aware Mapping of KPN Applications onto Heterogeneous MPSoCs. In *Proc. of the 49th Annu. Design Automat. Conf.*, pages 1266–1271. ACM, 2012.

[24] C.-L. Chou, U. Y. Ogras, and R. Marculescu. Energy-and performance-aware incremental mapping for networks on chip with multiple voltage levels. *IEEE Trans. Comput.-Aided Des. Integr. Syst.*, 27(10):1866–1879, 2008.

[25] J. Cong et al. Architecture Support for Domain-Specific Accelerator-Rich CMPs. *ACM Trans. on Embedded Computing Syst. (TECS)*, 13(4s):131, 2014.

[26] E. L. de Souza Carvalho, N. L. V. Calazans, and F. G. Moraes. Dynamic Task Mapping for MPSoCs. *IEEE Design & Test of Computers*, 27(5):26–35, 2010.

[27] P. G. Del Valle, D. Atienza, I. Magan, J. G. Flores, E. A. Perez, J. M. Mendias, L. Benini, and G. De Micheli. A Complete Multi-processor System-on-Chip FPGA-based Emulation Framework. In *IEEE International Conference on Very Large Scale Integration*, pages 140–145, 2006.

[28] A. Y. Ding and M. Janssen. Opportunities for Applications using 5G Networks: Requirements, Challenges, and Outlook.

[29] B. Donyanavard, T. Mück, A. M. Rahmani, N. Dutt, A. Sadighi, F. Maurer, and A. Herkersdorf. SOSA: Self-optimizing Learning with Self-adaptive Control for Hierarchical System-on-chip Management. In *Proceedings of the 52nd Annual IEEE/ACM International Symposium on Microarchitecture*, pages 685–698, 2019.

[30] J. Duke. Memory Forensics Comparison of Apple M1 and Intel Architecture Using Volatility Framework. 2021.

[31] H. Esmaeilzadeh, E. Blem, R. S. Amant, K. Sankaralingam, and D. Burger. Dark Silicon and the End of Multicore Scaling. In *38th Annual International Symposium on Computer Architecture (ISCA)*, pages 365–376. IEEE, 2011.

[32] M. R. Garey and D. S. Johnson. *Computers and Intractability: A Guide to the Theory of NP-Completeness*. W. H. Freeman & Co., USA, 1979.

[33] D. Geer. Chip Makers Turn to Multicore Processors. *Computer*, 38(5):11–13, 2005.

[34] P. Gepner and M. F. Kowalik. Multi-core Processors: New Way to Achieve High System Performance. In *International Symposium on Parallel Computing in Electrical Engineering (PARELEC'06)*, pages 9–13. IEEE, 2006.

[35] V. Goel, M. Slusky, W.-J. van Hoeve, K. C. Furman, and Y. Shao. Constraint Programming for LNG Ship Scheduling and Inventory Management. *European Journal of Operational Research*, 241(3):662–673, 2015.

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

[37] X. Guo, E. Ipek, and T. Soyata. Resistive Computation: Avoiding the Power Wall with Low-leakage, STT-MRAM based Computing. *ACM SIGARCH computer architecture news*, 38(3):371–382, 2010.

[38] Z. Han, K. Devarajegowda, A. Neumeier, and W. Ecker. IP-Coding Style Variants in a Multi-layer Generator Framework. In *Design and Verification Conference and Exhibition (DVCon)*, 2020.

[39] Hardkernel. ODROID-XU3. `https://wiki.odroid.com/old_product/odroid-xu3/odroid-xu3` Accessed 20 Mar. 2020, 2014.

[40] J. L. Hennessy and D. A. Patterson. A New Golden Age for Computer Architecture. *Commun. of the ACM*, 62(2):48–60, 2019.

[41] C.-Y. Huang, Y.-F. Yin, C.-J. Hsu, T. B. Huang, and T.-M. Chang. SoC HW/SW Verification and Validation. In *IEEE Asia and South Pacific Design Automation Conference (ASP-DAC)*, pages 297–300, 2011.

[42] 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.

[43] J. Jeffers et al. *Intel Xeon Phi Processor High Performance Programming: Knights Landing Edition*. Morgan Kaufmann, 2016.

[44] N. P. Jouppi and D. W. Wall. Available Instruction-level Parallelism for Superscalar and Superpipelined Machines. *ACM SIGARCH Computer Architecture News*, 17(2):272–282, 1989.

[45] N. Khalilzad, K. Rosvall, and I. Sander. A modular design space exploration framework for multiprocessor real-time systems. In *2016 Forum on Specification and Design Languages (FDL)*, pages 1–7. IEEE, 2016.

[46] F. H. Khan, M. A. Pasha, and S. Masud. Advancements in Microprocessor Architecture for Ubiquitous AI—An Overview on History, Evolution, and Upcoming Challenges in AI Implementation. *Micromachines*, 12(6):665, 2021.

[47] R. Khasanov, J. Robledo, C. Menard, A. Goens, and J. Castrillon. Domain-specific Hybrid Mapping for Energy-efficient Baseband Processing in Wireless Networks. *ACM Transactions on Embedded Computing Systems (TECS)*, 20(5s):1–26, 2021.

[48] B. Kienhuis et al. An approach for quantitative analysis of application-specific dataflow architectures. In *Proc. IEEE Int. Conf. on Application-Specific Systems, Architectures and Processors*, pages 338–349. IEEE, 1997.

[49] 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.

[50] A. Krishnakumar et al. Runtime Task Scheduling using Imitation Learning for Heterogeneous Many-core Systems. *IEEE Trans. on CAD of Integr. Circuits and Syst.*, 39(11):4064–4077, 2020.

[51] A. Krishnakumar and U. Y. Ogras. Performance analysis and optimization of decision tree classifiers on embedded devices: work-in-progress. In *Proceedings of the 2021 International Conference on Embedded Software*, pages 37–38, 2021.

[52] Y.-K. Kwok and I. Ahmad. Dynamic Critical-Path Scheduling: An Effective Technique for Allocating Task Graphs to Multiprocessors. *IEEE Trans. Parallel Distrib. Syst.*, 7(5):506–521, 1996.

[53] C.-C. Lin et al. Energy-efficient Task Scheduling for Multi-core Platforms with Per-core DVFS. *Journal of Parallel and Distributed Computing*, 86:71–81, 2015.

[54] D. Lin, S. Talathi, and S. Annapureddy. Fixed Point Quantization of Deep Convolutional Networks. In *Intl. Conf. on Machine Learning*, pages 2849–2858, 2016.

[55] S. Liu et al. Computer Architectures for Autonomous Driving. *Computer*, 50(8):18–25, 2017.

[56] S. Lotlikar, V. Pai, and P. V. Gratz. AcENoCs: A Configurable HW/SW Platform for FPGA Accelerated NoC Emulation. In *24th International Conference on VLSI Design*, pages 147–152, 2011.

[57] J. Mack, N. Kumbhare, A. NK, U. Y. Ogras, and A. Akoglu. User-Space Emulation Framework for Domain-Specific SoC Design. In *2020 IEEE Int. Parallel and Distrib. Process. Symp. Workshops)*, pages 44–53, 2020.

[58] 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.

[59] 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 Trans. on Very Large Scale Integration (VLSI) Systems*, 2019.

[60] S. K. Mandal et al. Analytical performance models for NoCs with multiple priority traffic classes. *ACM Transactions on Embedded Computing Systems (TECS)*, 18(5s):52, 2019.

[61] S. K. Mandal, A. Krishnakumar, R. Ayoub, M. Kishinevsky, and U. Y. Ogras. Performance Analysis of Priority-aware NoCs with Deflection Routing under Traffic Congestion. In *Proceedings of the 39th International Conference on Computer-Aided Design*, pages 1–9, 2020.

[62] S. K. Mandal, A. Krishnakumar, and U. Y. Ogras. Energy-Efficient Networks-on-Chip Architectures: Design and Run-Time Optimization. *Network-on-Chip Security and Privacy*, page 55, 2021.

[63] 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.

[64] 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.

[65] A. Mirhoseini et al. Device Placement Optimization with Reinforcement Learning. In *International Conference on Machine Learning-Volume 70*, pages 2430–2439. JMLR. org, 2017.

[66] G. Mitra et al. Use of SIMD Vector Operations to Accelerate Application Code Performance on Low-powered ARM and Intel Platforms. In *IEEE Intl. Symp. on Parallel & Distrib. Processing, Workshops and PhD Forum*, pages 1107–1116, 2013.

[67] K. Moazzemi, B. Maity, S. Yi, A. M. Rahmani, and N. Dutt. HESSLE-FREE: Heterogeneous Systems Leveraging Fuzzy Control for Run-time Resource Management. *ACM Transactions on Embedded Computing Systems (TECS)*, 18(5s):1–19, 2019.

[68] D. Morawiec. sklearn-porter. Transpile Trained Scikit-learn Estimators to C, Java, JavaScript and others.

[69] K. Neubauer et al. Exact multi-objective design space exploration using aspmt. In *2018 Design, Automation & Test in Europe Conference & Exhibition (DATE)*, pages 257–260. IEEE, 2018.

[70] B. Noethen, O. Arnold, E. P. Adeva, T. Seifert, E. Fischer, S. Kunze, E. Matúš, G. Fettweis, H. Eisenreich, G. Ellguth, et al. 10.7 A 105GOPS 36mm 2 Heterogeneous SDR MPSoC with Energy-aware Dynamic Scheduling and Iterative Detection-decoding for 4G in 65nm CMOS. In *IEEE International Solid-State Circuits Conference Digest of Technical Papers (ISSCC)*, pages 188–189, 2014.

[71] G. O'Regan. Intel. In *Pillars of Computing*, pages 129–133. Springer, 2015.

[72] C. S. Pabla. Completely Fair Scheduler. *Linux Journal*, (184), 2009.

[73] E. L. Padoin, L. L. Pilla, M. Castro, F. Z. Boito, P. O. A. Navaux, and J.-F. Méhaut. Performance/energy Trade-off in Scientific Computing: The Case of ARM big. LITTLE and Intel Sandy Bridge. *IET Computers & Digital Techniques*, 9(1):27–35, 2015.

[74] F. Pedregosa et al. Scikit-learn: Machine Learning in Python. *Journal of Machine Learning Research*, 12:2825–2830, 2011.

[75] A. D. Pimentel, C. Erbas, and S. Polstra. A systematic approach to exploring embedded system architectures at multiple abstraction levels. *IEEE Trans. on Computers*, 55(2):99–112, 2006.

[76] A. D. Pimentel et al. Exploring embedded-systems architectures with artemis. *Computer*, 34(11):57–63, 2001.

[77] D. Price. Pentium FDIV Flaw-lessons Learned. *IEEE Micro*, 15(2):86–88, 1995.

[78] N. Rajovic, A. Rico, J. Vipond, I. Gelado, N. Puzovic, and A. Ramirez. Experiences with Mobile Processors for Energy Efficient HPC. In *Design, Automation & Test in Europe Conference & Exhibition (DATE)*, pages 464–468. IEEE, 2013.

[79] S. Raza, S. Wang, M. Ahmed, and M. R. Anwar. A Survey on Vehicular Edge Computing: Architecture, Applications, Technical Issues, and Future Directions. *Wireless Communications and Mobile Computing*, 2019, 2019.

[80] B. K. Reddy, A. K. Singh, D. Biswas, G. V. Merrett, and B. M. Al-Hashimi. Inter-Cluster Thread-to-Core Mapping and DVFS on Heterogeneous Multi-Cores. *IEEE Transactions on Multi-Scale Computing Systems*, 4(3):369–382, 2017.

[81] P. Rosenfeld, E. Cooper-Balis, and B. Jacob. Dramsim2: A cycle accurate memory system simulator. *IEEE computer architecture letters*, 10(1):16–19, 2011.

[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] F. Rossi, P. Van Beek, and T. Walsh. *Handbook of Constraint Programming*. Elsevier, 2006.

[84] E. Rotem, Y. Mandelblat, V. Basin, E. Weissmann, A. Gihon, R. Chabukswar, R. Fenger, and M. Gupta. Alder Lake Architecture. In *2021 IEEE Hot Chips 33 Symposium (HCS)*, pages 1–23. IEEE, 2021.

[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] S. Sarma and N. Dutt. FPGA emulation and prototyping of a cyberphysical-system-on-chip (CPSoC). In *25nd IEEE International Symposium on Rapid System Prototyping*, pages 121–127, 2014.

[87] A. L. Sartor et al. HiLITE: Hierarchical and Lightweight Imitation Learning For Power Management Of Embedded SoCs. *IEEE CAL*, 19(1):63–67, 2020.

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

[89] B. Senouci, F. Petrot, et al. Large Scale On-chip Networks: An Accurate Multi-FPGA Emulation Platform. In *11th EUROMICRO Conference on Digital System Design Architectures, Methods and Tools*, pages 3–9, 2008.

[90] L. T. Smit, J. L. Hurink, and G. J. Smit. Run-time Mapping of Applications to a Heterogeneous SoC. In *2005 Int. Symp. on System-on-Chip*, pages 78–81. IEEE, 2005.

[91] R. Struharik. Decision Tree Ensemble Hardware Accelerators for Embedded Applications. In *13th Intl. Synp. on Intell. Syst. and Inform.*, pages 101–106, 2015.

[92] R. S. Sutton, D. A. McAllester, S. P. Singh, and Y. Mansour. Policy Gradient Methods for Reinforcement Learning with Function Approximation. In *Advances in Neural Information Processing Systems*, pages 1057–1063, 2000.

[93] V. Swaminathan and K. Chakrabarty. Real-Time Task Scheduling for Energy-Aware Embedded Systems. *Journal of the Franklin Institute*, 338(6):729–750, 2001.

[94] X. Tan et al. Performance Analysis of a Hardware Accelerator of Dependence Management for Task-based Dataflow Programming Models. In *IEEE Intl. Symp. on Performance Analysis of Systems and Software*, pages 225–234, 2016.

[95] T. N. Theis and H.-S. P. Wong. The End of Moore's Law: A New Beginning for Information Technology. *Computing in Science & Engineering*, 19(2):41–50, 2017.

[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] N. Trčka et al. Integrated model-driven design-space exploration for embedded systems. In *2011 Int. Conf. on Embedded Comput. Syst. Arch., Modeling and Simulation*, pages 339–346. IEEE, 2011.

[98] R. Uhrie et al. 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. Int. Society for Optics and Photonics, 2019.

[99] J. Ullman. NP-Complete Scheduling Problems. *Journal of Computer and System Sciences*, 10(3):384 – 393, 1975.

[100] P. Varga, J. Peto, A. Franko, D. Balla, D. Haja, F. Janky, G. Soos, D. Ficzere, M. Maliosz, and L. Toka. 5G Support for Industrial IoT Applications—Challenges, Solutions, and Research Gaps. *Sensors*, 20(3):828, 2020.

[101] M.-A. Vasile, F. Pop, R.-I. Tutueanu, V. Cristea, and J. Kołodziej. Resource-Aware Hybrid Scheduling Algorithm in Heterogeneous Distributed Computing. *Future Generation Computer Systems*, 51:61–71, 2015.

[102] A. Vehtari, A. Gelman, and J. Gabry. Practical Bayesian Model Evaluation using Leave-one-out Cross-validation and WAIC. *Statistics and Computing*, 27(5):1413–1432, 2017.

[103] L. Wang and K. Skadron. Implications of the Power Wall: Dim Cores and Reconfigurable Logic. *IEEE Micro*, 33(5):40–48, 2013.

[104] Y. Wang et al. Multi-Objective Workflow Scheduling with Deep-Q-Network-based Multi-Agent Reinforcement Learning. *IEEE Access*, 7:39974–39982, 2019.

[105] Y. Wen, Z. Wang, and M. F. O'Boyle. Smart Multi-Task Scheduling for OpenCL Programs on CPU/GPU Heterogeneous Platforms. In *International Conf. on High Performance Computing*, pages 1–10, 2014.

[106] C. Xian, Y.-H. Lu, and Z. Li. Dynamic Voltage Scaling for Multitasking Real-time Systems with Uncertain Execution Time. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 27(8):1467–1478, 2008.

[107] 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.

[108] 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.

[109] 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.

[110] H. Zheng and A. Louri. An Energy-efficient Network-on-chip Design using Reinforcement Learning. In *Proceedings of the 56th Annual Design Automation Conference*, pages 1–6, 2019.