Energy-Efficient Domain Specific Architectures: Design Space Exploration, Algorithms and FPGA Prototyping

By

Anish Nallamur Krishnakumar

A dissertation submitted in partial fulfillment of
the requirements for the degree of

Doctor of Philosophy

(Electrical and Computer Engineering)

at the

UNIVERSITY OF WISCONSIN–MADISON

2022

Date of final oral examination: 12/05/2022

Dedicated to my parents Annapurna and Krishnakumar,
in-laws Rajalakshmi and Gopal,
brother Adarsh, and my wife Reshmi.

ACKNOWLEDGMENTS

First and foremost, I extend my deepest gratitude to my advisor, Prof. Umit Y. Ogras, for the exemplary guidance, advice, and encouragement he has offered me throughout my Ph.D. journey. His insightful thoughts and recommendations have helped me solve several challenges I faced during my studies. Specifically, his advice on organizing thoughts and writing research papers, developing simulators, time management, and, most importantly, enabling me to develop the ability to perform independent research will help me a long way in my career ahead. I cherish the bond we have developed over the years and am sure it will continue beyond my journey at the University of Wisconsin-Madison.

I am immensely thankful to Prof. Yu-Hen Hu, Prof. Matthew Sinclair, and Prof. Chaitali Chakrabarti for taking the time to serve on my Ph.D. defense committee. Their suggestions and feedback have been invaluable in improving the quality of my research and this dissertation.

I sincerely thank Prof. Chaitali Chakrabarti, Prof. Radu Marculescu, and Prof. Ali Akoglu for our insightful research discussions over the years. I greatly value their collaboration, suggestions, and feedback. I also extend my sincere thanks to Prof. Daniel Bliss, Prof. David Blaauw, Prof. Trevor Mudge, Prof. Hun-Seok Kim, Prof. Ronald Dreslinski, and Brittany McCall for their input and support during my research.

I am grateful for the exciting discussions, words of encouragement, and lighter moments with friends and colleagues at eLab: Dr. Samet Arda, Prof. Ganapati Bhat, Prof. Sumit Mandal, Shruti Narayana, Yigit Tuncel, Sizhe An, Alper Goksoy, Toygun Basaklar, Vishrut Pandey, Jie Tong, Nuriye Yildirim, Aditya Ukarande, and Conrad Holt. I also thank colleagues and friends at other universities: Dr. Tutu Ajayi, Dr. Hanguang Yu, Dr. Alex Chiriyath, Dr. Arindam Dutta, Joshua Mack, Sahil Hassan, Serhan Gener, Alexander Fusco, Ilkin Aliyev, Xing Chen, Adarsh Akkshai Venkataramani, and Kuan-Yu Chen.

At this juncture, I thank Nagadastagiri Challapalle, who motivated me to pursue a Ph.D. and has been a constant pillar of support for over five years. I thoroughly enjoyed the words of encouragement and lighter moments with my friends; at Madison: Sneha Chavali, Srinath Namburi, Akhil, Bodduppalli, Srinivas Pothuraju, Siddharth RCS, Ramakrishna Raju, Rahim Shaik, and Indu Kilaru; at Arizona: Surya Golkonda, Yashasvi Reddy, Naga Venkatesh, Aswin Baskaran, and Medhini Vuyyuru; Ranjith Kumar, Makesh Tarun and Raghuraj Krishnamurthy; at India: Vijay Vigneshan, Praveen Kumar, Sona Aishwarya, Rajesh Ganesan, Rajbarath, Jatin Kumar, Kajal Luthra, Anupam Sobti and Shruti Sharma.

I am thankful to Defense Advanced Research Projects Agency (Grant FA8650-18-2-7860) for funding the research presented in this dissertation. My deepest gratitude goes to Prof. Moorthi Sridharan, Prof. Selvan M.P, and Prof. Raja Pitchaimuthu for planting the seeds of pursuing a research career during my undergraduate days.

Finally, this would not have been possible without the support of my parents, Annapurna and Krishnakumar, in-laws Rajalakshmi and Gopal, my brother Adarsh, and my family. Their love, support, and encouragement have been of immense support in this journey. Finally, I am deeply grateful to my loving wife Reshmi, whose unconditional support and immense sacrifice have enabled me to complete this journey seamlessly.

CONTENTS

LIST OF TABLES

LIST OF FIGURES

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 cannot cater to the contrasting performance and energy requirements of 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 compared to special-purpose solutions. 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 runtime. To address all the above challenges, domain-specific architectures (DSAs) judiciously combine processing elements such as general-purpose cores, special-purpose cores, and hardware accelerators to maximize the energy efficiency of domain applications and provide programming flexibility.

The major challenge in DSAs is to optimally utilize the diverse processing elements at runtime to exploit their potential. Mapping tasks to the processing elements (task scheduling) and controlling their voltage and frequencies are critical aspects of resource management. To this end, we pose task scheduling as a classification problem and propose a novel offline imitation learning framework and decision tree (DT) classifiers. Our imitation learning-based scheduling policy achieves performance that is within 1% of an Oracle for multiple optimization objectives and SoC configurations. The offline-trained scheduling policies become ineffective when new applications or processing clusters are introduced in the workload; hence, they must be updated online. DTs pose an additional challenge in online training since they use the entire dataset. To address this challenge, we propose an incremental and online lightweight training framework for DTs that achieves a performance within 5% of a baseline DT by storing only 1-8% of the original dataset. To support the rapid exploration and evaluation of resource management algorithms, we developed a high-level discrete-event full-system simulation framework that models the processing elements, scheduling pipeline, and other components of the system. We also developed an FPGA-based prototyping and emulation framework to enable functional validation and early software development. This dissertation addresses critical challenges in DSA runtime resource management and evaluation frameworks that accelerate their design and development for mainstream adoption.

1 INTRODUCTION

Process technology-driven power, performance, and energy efficiency improvements have recently slowed down significantly [5, 6]. In addition, performance cannot be further improved by scaling the frequency arbitrarily due to the power wall [7, 8]. Consequently, two primary drivers of higher performance-per-watt cease to provide the expected gains. At the same time, the instruction-level parallelism techniques, such as processor pipelining, prefetching, and out-of-order execution, provide only marginal benefits, thereby leaving a substantial scope for improvement in performance and energy efficiency [9].

Homogeneous multicore architectures integrate multiple identical cores onto the same die to provide higher computational capabilities under similar area budgets [10, 11]. They opened new avenues to parallel processing capabilities with higher performance at a modest power consumption increase, thereby allowing drastic energy efficiency improvements [12]. However, homogeneous cores cannot simultaneously satisfy competing application requirements, such as low power and high performance. Low-power cores, such as the Arm Cortex-M series, have limited performance. In contrast, high-performance cores, such as the Arm Cortex-A72/A76 processors, consume higher power due to the out-of-order execution nature, large caches, and deep execution pipelines. Heterogeneous multiprocessor architectures address this problem by integrating low-power and high-performance cores [13, 14]. Therefore, heterogeneous architectures are extensively used in most processing systems, such as mobile phones, laptops, desktops, and servers [15, 16, 17].

Heterogeneous architectures significantly improve performance and energy

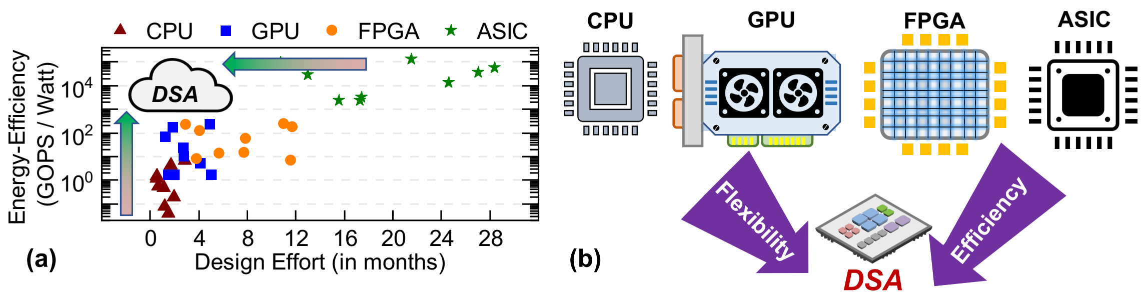


Figure 1.1: (a) Trends in energy efficiency and design effort in Giga operations per Watt (GOPS / Watt) for applications implemented on CPU, GPU, FPGA, and fixed-function/special-purpose ASIC. (b) An illustration of domain-specific architecture (DSA) combining the flexibility benefits of CPU and GPU implementations, the performance benefits of FPGA, and the energy efficiency of fixed-function ASIC implementations.

efficiency compared to their homogeneous counterparts. However, they still have a substantial gap with application-specific integrated circuits (ASIC). To provide a quantitative comparison, Figure 1.1(a) shows the energy efficiency of applications implemented on CPU, GPU, FPGA, and ASIC. CPU implementations require the least design effort and also provide low energy efficiency [18]. GPUs and FPGAs improve energy efficiency and performance by exploiting single-instruction multiple data (SIMD) execution and parallelism benefits, respectively [19, 20]. Application code is converted to GPU-compatible code to run on GPUs, and hardware description languages or high-level synthesis for FPGAs. ASICs provide the highest energy efficiency since they are specifically designed for the target application [21]. However, the ASIC effort, which includes design, development, fabrication, and software development could require several months to years. Therefore, there is a critical need to continue the evolution of computing architectures to provide

ASIC-like energy efficiency with the shortest possible time-to-market.

Domain-specific architectures represent an emerging instance of heterogeneous architectures that optimize data flow for applications in a target domain through hardware acceleration while providing programming flexibility [22, 23, 24, 25]. Examples of recently growing domains include wearable and human activity recognition for health monitoring, machine learning and artificial intelligence (AI), autonomous driving, and software defined radios [26, 27, 28, 29, 30, 23]. For instance, machine learning and AI are extensively being used for image processing, scheduling, recommendation systems, spam filtering, stock market analysis, video analytics, and medical applications [31, 32, 33, 34, 35, 29, 36, 37, 38]. Hence, there is a strong need for computing architectures that enable seamless, high-performance, and energy-efficient execution of these domain applications. DSAs aim at improved programmability by including general-purpose cores and the highest energy efficiency by integrating special-purpose processors and hardware accelerators. The domain-specific nature of DSAs stems from the fact that the hardware accelerators (e.g., as [39, 40]) and data flows are highly tailored to the type of computations in the applications of a particular domain. Broadly speaking, DSAs encompass any computing architecture that provides:

- **Superior energy efficiency through specialized processing:** The specialized processors accelerate the frequently occurring domain-specific computations in hardware, thereby boosting energy efficiency. For example, a custom-designed fast Fourier transform (FFT) hardware accelerates the direct- and inverse-FFT operations, while a systolic matrix multiplication processor accelerates machine learning and AI applications.

- **Programmability/flexibility:** DSAs aim to improve the programming flexibility for both domain and non-domain applications. For example, DSAs that target neural network inference must be programmable to execute multi-layer perceptrons, convolutional neural networks, and recurrent neural networks. *Second*, they must be capable of executing other neural network inference operations that cannot be easily implemented using specialized hardware. *Finally*, they should be able to execute non-domain applications to improve flexibility and enable broader usage.

- **Heterogeneous processing elements (PEs):** The diverse types of PEs in DSAs cater to contrasting application requirements such as low power, high performance, energy efficiency, and programmability.

The potential of DSAs is also evident in recent and growing commercial examples. Google's tensor processing unit (TPU) comprises hardware designs, systems, and software stacks to accelerate machine learning training and inference [41, 42]. TPUs provide $3\times$–$7\times$ speedup over state-of-the-art GPUs and $80\times$ better energy efficiency than general-purpose processors [22, 43, 44]. Nvidia's data center processing unit (DPU) is another DSA that integrates high-performance Arm cores and hardware accelerators with an extensive software eco-system optimized for AI, cloud supercomputing, network security, and wireless communication [45]. Intel's infrastructure processing unit (IPU) is a programmable network device that integrates with server CPUs to accelerate networking control, storage management, and security. Offloading the infrastructure operations to the IPU reduces the overhead of infrastructure tasks to improve overall performance and energy consumption [46]. In the low-power domain, RedMulE offers a sub-100 mW DSA for deep learning that comprises RISC-V cores and dedicated matrix-multiplication accelerators [47]. DSAs have started making substantial strides in all domains to offer superior energy efficiency and short time-to-market.

1.1 Contributions

DSAs offer multiple alternative PEs to execute tasks, such as general-purpose cores, hardware accelerators, and specialized processors. To exploit the potential of DSAs, one of the most critical aspects remains the ability to efficiently utilize the available PEs for task execution [48, 49, 18]. To this end, system-level design - including scheduling, power-thermal management algorithms and design space exploration studies - plays a crucial role. The first contribution of this dissertation presents a system-level domain-specific SoC simulation (DS3) framework to address this need. DS3 enables both design space exploration and dynamic resource management for power-performance optimization of domain applications. We showcase DS3 using six real-world applications from wireless communications and radar processing domain. DS3, as well as the reference applications, is shared as open-source software to stimulate research in this area.

Reaching the full potential of these architectures depends critically on optimally scheduling the applications to available resources at runtime. The second contribution of this dissertation addresses these two requirements simultaneously. Existing optimization-based techniques cannot achieve this objective at runtime due to the combinatorial nature of the task scheduling problem. As the third theoretical contribution, this work poses scheduling as a classification problem and proposes a hierarchical imitation learning (IL)-based scheduler that learns from an Oracle to maximize the performance of multiple domain-specific applications. Extensive evaluations with six streaming applications from wireless communications and radar domains show that the proposed IL-based scheduler approximates an offline Oracle policy with more than 99% accuracy for performance- and energy-based optimization objectives. Furthermore, it achieves almost identical performance to the Oracle with a low runtime overhead and successfully adapts to new applications, many-core system configurations, and runtime variations in application characteristics. The fourth contribution focuses on optimization of decision tree classifier inference in hardware and software and achieves a latency of less than 50 nanoseconds for trees of up to depth 12.

Decision trees (DTs) perform high-quality, low-latency task scheduling to utilize the massive parallelism and heterogeneity in DSSoCs effectively. However, offline trained DT scheduling policies can quickly become ineffective when applications or hardware configurations change. There is a critical need for runtime techniques to train DTs incrementally without sacrificing accuracy since current training approaches have large memory and computational power requirements. To address this need, we propose INDENT, an incremental online DT framework to update the scheduling policy and adapt it to unseen scenarios. INDENT updates DT schedulers at runtime using only 1-8% of the original training data embedded during training. Thorough evaluations with hardware platforms and DSSoC simulators demonstrate that INDENT performs within 5% of a DT trained from scratch using the entire dataset and outperforms current state-of-the-art approaches.

DSSoCs integrate several hardware components, resulting in higher design, implementation, and validation complexities. Significant emphasis on pre-silicon validation is required to eliminate functional and performance bugs to avoid the post-silicon failure cost and market loss penalties. Emulation frameworks accelerate pre-fabrication design validation by prototyping target designs. They also allow the early development of software, drivers, firmware, and performance analysis. As the final contribution, we present FALCON, a full-system DSSoC emulation platform that allows functional validation of DSSoCs as they interact with the operating system and runtime frameworks. We show that FALCON allows early software and driver development for accelerators and validates the system before the silicon availability. Finally, we demonstrate that FALCON enables rapid and extensive design space exploration to obtain early pre-silicon power and performance estimates.

In summary, this dissertation makes the following contributions:

- A detailed and comprehensive literature review on DSAs and corresponding research directions [50],
- DS3, a domain-specific system-on-chip simulation framework to perform rapid design space exploration and evaluate resource management algorithms [51],

- An imitation learning (IL) based task scheduling approach [49],
- Optimization techniques for decision tree classifiers [52],
- An incremental and online decision tree training framework [53], and
- An FPGA-based emulation framework for domain-specific architectures.

The rest of the dissertation is organized as follows. The literature survey is discussed in Chapter 2. Chapter 3 presents DS3 and discusses its role in high-level design space exploration and evaluation of resource management algorithms. The IL based task scheduling approach for heterogeneous SoCs is presented in Chapter 4. The optimization of decision tree (DT) classifiers for IL is discussed in Chapter 5. Chapter 6 presents a novel approach to adapt standalone DT classifiers online to variations in the environment. Chapter 7 presents the FPGA-based emulation framework to perform pre-silicon functional validation, early software development, performance and power analysis. Finally, Chapter 8 concludes this dissertation with directions for future work.

2 LITERATURE REVIEW

2.1 Research Directions in Domain-Specific Architectures

DSAs have the potential to enable high energy efficiency and programmability across multiple applications. However, critical research and infrastructure design challenges must be addressed before DSAs can become a mainstream computing paradigm [6, 22, 54]. For instance, designers must choose the optimal number and type of PEs to balance design time, cost, complexity, area, and energy efficiency. Novel and rapid hardware design techniques that condense the design time and costs allow for a shorter time-to-market [54, 55]. Similarly, DSAs require novel and state-of-the-art simulation, compilation, and emulation frameworks to minimize the gap between conceptualization and market availability of a product [56]. In summary, there is a strong need to understand the factors that currently limit the design and deployment of DSAs. To this end, this dissertation identifies the key research areas (summarized in Figure 2.1) that need new ideas and solutions to make DSAs default choices for designers, developers, and end-users:

- **Domain Representation:** Application source code must be analyzed to extract the domain-specific kernels and construct the data flow graphs that can exploit the data- and task-level parallelism both in applications and hardware [57]. Understanding the domain applications plays a critical role in selecting the PEs for the DSA.

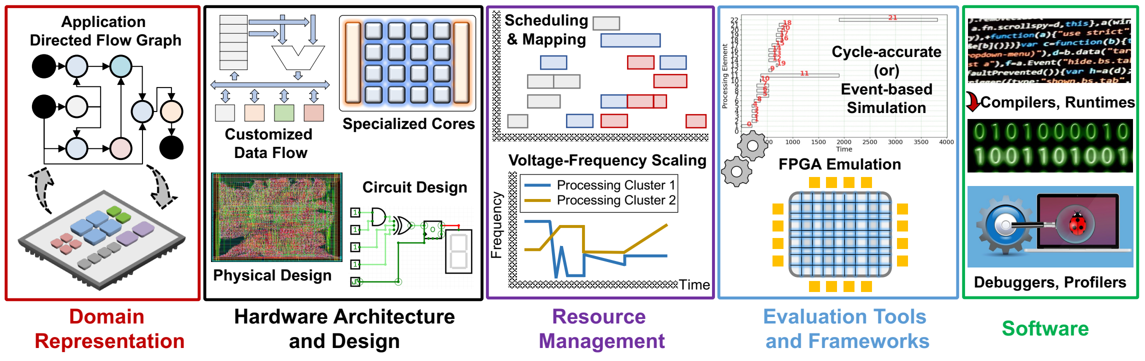


Figure 2.1: Prime research directions in the conceptualization, design and development of DSAs. Applications are represented as directed flow graphs. The nodes in the graph represent the key computational kernels within each application and the edges of the graph denote the communication volumes between kernels.

- **Hardware Architecture and Design:** With the saturation of energy efficiency of general-purpose processors, DSAs require novel hardware architectures and innovative solutions to exploit parallelism and maximize energy efficiency for domain-specific kernels.

- **Resource Management in DSAs:** Exploiting the full potential of DSAs involves optimally allocating the tasks to PEs, and selecting their voltage-frequency levels at runtime using resource management algorithms.

- **Evaluation Frameworks and Productivity Tools:** DSAs demand the need for rapid design space exploration frameworks to aid top-level design decisions in the early development phase, and emulation platforms to aid functional validation and software development.

- **Software Development:** The challenge in programming DSAs with heterogeneous PEs demands innovation in software frameworks and toolchains.

Furthermore, the software stack also serves as a bridge between domain representation, hardware and resource management techniques.

2.1.1 Interactions between the Research Directions

This section discusses the interactions between the research directions presented so far, as outlined in Figure 2.2. Domain representation efforts analyze the applications and present computational kernels that are potential candidates for implementation in special-purpose hardware. The hardware architecture and design exploit this information to develop customized and efficient processing for these potential candidates using either fixed-function or specialized accelerators. Hardware design techniques also leverage the data dependencies between the kernels in applications to design an appropriate on-chip communication network, such as bus, point-to-point network, and network-on-chip. Similarly, the domain analysis provides design-time and runtime information to the resource management algorithms. Design-time information includes the kernel characteristics and their interactions that affect latency, execution time, and communication volumes. Resource management algorithms exploit this information offline to deploy targeted scheduling and power management techniques for the chosen PEs. The domain-specific information in the applications helps narrow down the vast design space. Simulation frameworks perform rapid design space explorations and systematically evaluate resource management algorithms. The domain representation techniques and software stack share similar tools and infrastructure, such as compilation and performance profiling APIs. They target specific hardware by providing the rele-

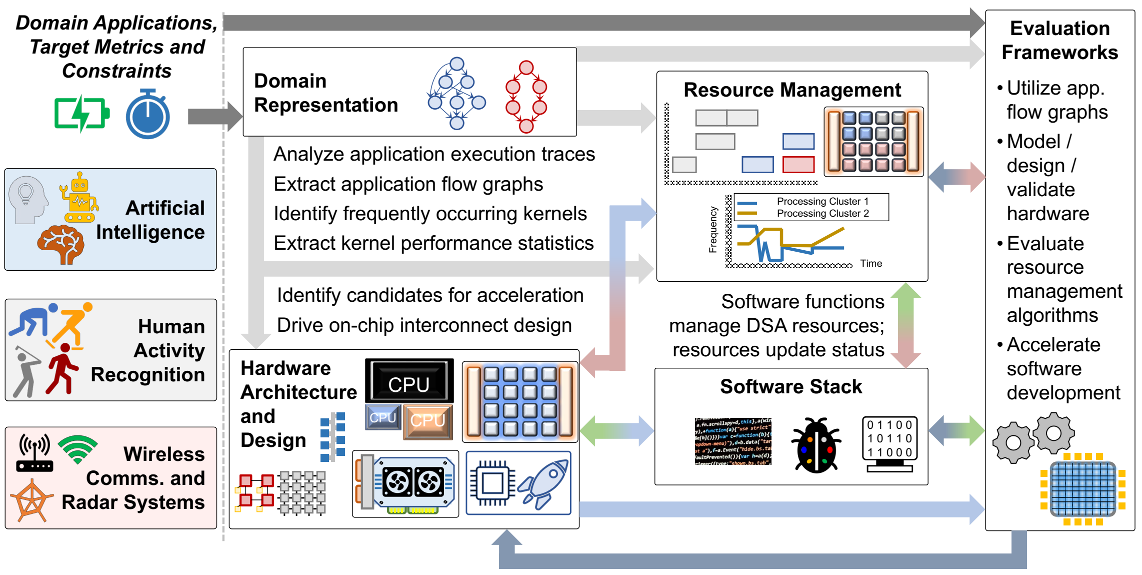

Domain Applications, Target Metrics and Constraints
Artificial Intelligence
Human Activity Recognition
Wireless Comms. and Radar Systems
Domain Representation
Analyze application execution traces
Extract application flow graphs
Identify frequently occurring kernels
Extract kernel performance statistics
Identify candidates for acceleration
Drive on-chip interconnect design
Resource Management
Processing Cluster 1
Processing Cluster 2
Frequency
Time
Software functions manage DSA resources; resources update status
Hardware Architecture and Design
CPU
CPU
CPU
Software Stack
01100
10110
11000
Evaluation Frameworks
• Utilize app. flow graphs
• Model / design / validate hardware
• Evaluate resource management algorithms
• Accelerate software development

Figure 2.2: Interactions between the research directions for the realization of DSAs. The kernel and flow graph information of domain applications and target metrics drive the hardware architecture and design of PEs. The application and hardware PE information is exploited by the resource management techniques. The DSA configurations are evaluated for functionality and performance using simulators, emulation frameworks, and the software stack.

vant compiler and API support. Finally, emulation frameworks accelerate software development, enable performance evaluation, and facilitate functional validation to improve the time-to-market for DSAs.

2.1.2 Insights and Open Challenges

DSAs are making solid advances toward becoming the preferred choice for future computing systems. DSA design and development efforts require significant attention as the algorithms, design methodologies, and tools evolve. Furthermore, the subtle interaction between the different DSA research aspects requires substantial research focus. The ever-lasting pursuit of maximizing performance and energy efficiency while minimizing the cost and design effort leaves the following open research questions:

- How can we reduce the time required to generate application traces, scope them and extract the flow graphs?
- Can we perform DSA hardware-aware application code compilation using state-of-the-art and just-in-time compilation techniques?
- Can we automatically generate highly optimized and specialized hardware based on high-level requirements, such as performance, power, and throughput?
- How can we architect easily programmable and flexible, yet highly specialized hardware?
- Can we design light-weight and near-optimal resource management algorithms considering all application requirements, such as performance, power consumption, energy efficiency, and deadlines?
- Can we explore preemption-based resource management techniques with hardware accelerators (that do not allow context switching) to address real-time needs?
- How can we automatically generate software support for custom-designed hardware accelerators and reduce the development time?

- How can we accurately and quickly calibrate high-level simulators with pre-silicon data or real hardware, speed up cycle-accurate simulations and reduce the development times for emulation frameworks?
- How can we embed security and privacy into all the DSA design aspects and components?
- How can we seamlessly integrate the different DSA research directions and tools to maximize performance and energy efficiency with minimum intervention from users and developers?

A few challenges involved in DSA design listed in the different sections are compiled here. These and similar questions demand innovation and research for performant and energy-efficient DSA-based computing systems. A comprehensive review on the research problems in DSAs and promising approaches is presented in [50]. This dissertation addresses some of the challenges described here to make DSAs a mainstream computing paradigm.

2.2 Evaluation Frameworks for DSAs

2.2.1 Simulation Frameworks

As the use cases for a simulation environment intersect with a number of distinct research areas, we break the related work into three parts. First, we discuss existing work in the area of scheduling, power, and thermal optimization algorithms, and we motivate a need for a unified framework that integrates these with rich design space exploration capabilities. Second, we discuss existing work in the area of design space exploration for embedded systems, and we note the lack of rich support for thermal/power models or plug-and-play scheduling frameworks. Third, for completeness, we give a brief overview of related works in the scope of high performance computing or non-embedded environments. Together, this set of related works serves to motivate the need for an environment such as DS3 that unifies all of these aspects into a single, open-source framework for embedded DSA development.

Starting with works on scheduling, power, and thermal optimization algorithms, one of the most important goals of heterogeneous SoC design is to optimize energy-efficiency while satisfying the performance constraints. To this end, a variety of offline and runtime algorithms have been proposed to schedule applications to PEs in multi-core architectures [58, 59, 60, 61]. Similarly, DVFS policies, such as HiCAP [62], power management governors, such as *ondemand* [63], and thermal management techniques [64] have been proposed to efficiently manage the power and temperature of SoCs. However, existing approaches are typically evaluated in isolated environments and different in-house tools. Hence, there is a strong need for a unified simulation framework [65] to compare and evaluate various scheduling algorithms in a common environment.

Next, 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. [66] 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 [67] 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. [68] utilize the Y-chart [69] philosophy for design space exploration and introduces an integrated framework using the Octopus toolset [70] 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 [71] aims to evaluate embedded-systems architecture instantiations at multiple abstraction levels. Later, authors extend the work and introduce the Sesame framework [72] 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 [73] 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.

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 [74] 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 [75] 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.

To the best of our knowledge, DS3 is the first open-source framework to integrate all of these distinct elements into a unified simulation environment targeting embedded DSAs. It includes built-in analytical models, scheduling algorithms, DTPM policies, and six reference applications from wireless communication and radar processing domain.

2.2.2 Emulation Frameworks

Emulation frameworks overcome the limitation of simulators by enabling functional verification, early software and firmware development [76, 73, 77]. The emulation frameworks are broadly classified into virtual model-based emulators and FPGA-based frameworks that rely on the actual implementation [78]. It is worth noting that only FPGA-based frameworks allow functionality validation since they use the actual implementation, as compared to representative models in virtual platforms.

The emergence of SystemC and TLM-based modeling also significantly enhanced the development of virtual emulation frameworks [79]. The quick emulator (QEMU) deploys abstract models of the computing elements and transaction-level models for its interactions with the rest of the system [80]. QEMU also enables developers to bringup a variety of guest operating systems and execute applications on the CPU through dynamic binary translations [81]. Along these lines, Arm provides fast models which are accurate and representative models of their IPs such as CPUs, interconnects, subsystems, and other peripheral components [82]. Fast models allow the bringup of the Linux OS, and programmers to develop software, firmware, and applications.

FPGA-based frameworks also improve task scheduling and DTPM policies by utilizing more realistic estimates in MPSoCs [83, 84]. An MPSoC-based sensor- and actuator-rich cyber-physical SoC is prototyped in [20], enabling hardware and software co-design. Other frameworks for MPSoCs are presented and discussed in [85, 86]. The MPSoC-based frameworks must be adapted for DSAs by integrating the other components, such as specialized cores, hardware accelerators, and on-chip interconnects. In that direction, the FPGA-based user-space emulation framework in [87] integrates hardware and software to evaluate resource management policies for DSAs. While this work is an initial step in the DSA direction, frameworks that scale to the entire design are critical to bridge the gap between the requirements and state-of-the-art approaches.

To the best of our knowledge, FALCON is the first FPGA-based emulation platform that integrates general-purpose processors, hardware accelerators, on-chip interconnect, and a detailed software stack that supports the Linux operating system (OS).

2.3 Resource Management Techniques for Heterogeneous Architectures

DSAs offer multiple alternative PEs to execute tasks, such as general-purpose cores, hardware accelerators, and specialized processors. To exploit the potential of DSAs, one of the most critical aspects remains the ability to efficiently utilize the available PEs for task execution [48, 49, 18, 88]. This section discusses the resource management aspects of DSAs, key bottlenecks, and outstanding research problems.

The techniques fall broadly into two categories: (1) static (or design-time) and (2) dynamic (or runtime) techniques. Static algorithms utilize the design time information to manage the resources [1, 89, 90]. These algorithms can provide optimal or heuristic solutions since they are not bounded by computation and latency constraints [91]. Static approaches cannot access runtime information and are inefficient in several scenarios [49]. DSAs inherently support several simultaneous applications that could demand a substantial amount of system resources. While static algorithms may suffice in limited application-specific scenarios, DSAs require efficient runtime resource management techniques. While several static and dynamic approaches have been proposed previously [92, 93, 94, 95, 96], the following fundamental challenges drive the research need for novel dynamic techniques that target DSAs:

- **Heterogeneity:** PEs with different power and performance characteristics for various applications require algorithms to evaluate all valid execution alternatives to obtain the optimal solution. Considering the characteristics of all heterogeneous PEs at runtime makes the resource management problem complex.

- **Streaming Arrivals:** Most applications (e.g., video/signal processing, autonomous driving, radar systems) continuously perform identical operations on streaming data frames. The complexity lies in efficiently managing the resources when randomly arriving frames overlap with currently executing and pending tasks from previous frames.

- **Concurrent Applications:** SoCs execute several applications simultaneously. Resource management techniques must recognize the divergent application

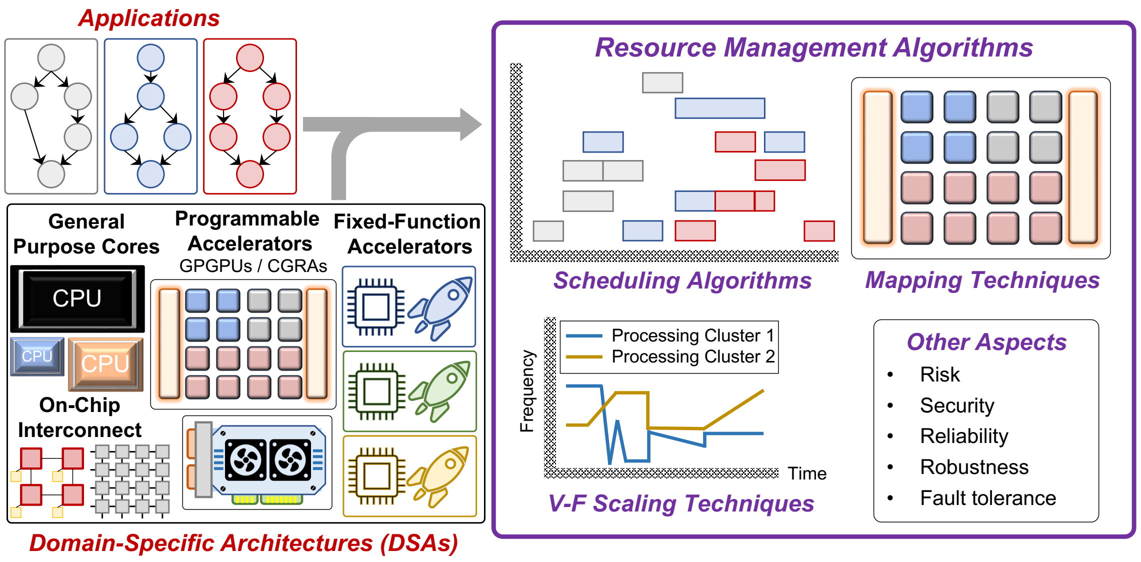


Figure 2.3: The key areas of focus in DSA resource management techniques which interface with the domain applications and DSA hardware. The domain representation outputs and hardware design decisions are used to design task scheduling, voltage-frequency scaling, and other resource management techniques.

characteristics and satisfy their compute requirements, performance, power, and deadline constraints.

The type of applications and the choice of hardware components in the DSA play a critical role in developing resource management techniques. The various aspects of resource management algorithms shown in Figure 2.3 are discussed in the next sections.

2.3.1 Task Scheduling Techniques

Current many-core systems use runtime heuristics to enable scheduling with low overheads. For example, the completely fair scheduler (CFS) [97], 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 [98]. With the same limitations, shortest job first (SJF) [99] 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 [100, 101] for DAGs [1, 102, 89] prioritize various objectives, such as energy [103, 104], fairness [105], security [106]. 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) [1] is one example, in which an upward rank is computed to perform the scheduling decisions. The authors in [102] 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 [89]. 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 [33]. A recent support vector machine (SVM)-based scheduler for OpenCL kernels assigns kernels (tasks) between CPUs and GPUs [107]. In contrast to schedulers that use supervised learning, authors in [108] uses reinforcement learning (RL) to schedule Tensorflow device placement, but lacks the ability of scheduling streaming jobs. DeepRM [33] 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 [109] 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 [109] 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 [110]. Recently, the use of imitation learning to intelligently manage power and energy consumption in SoCs has been demonstrated [111, 112]. 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*.

2.3.2 Dynamic Thermal-Power Management (DTPM) Techniques

State-of-the-art PEs and cores support multiple voltage and frequency (V-F) levels. Another critical aspect in exploiting the potential of DSAs is optimally selecting these power states at runtime [113, 114, 115]. The V-F levels of PEs play a primary role in determining power and performance, while power consumption determines the temperature of the PEs [116]. For example, using the highest V-F levels to maximize performance increases power consumption and, in turn, the temperature [117]. Furthermore, portions of the SoC may be placed in different levels of sleep states where they are partially or entirely powered off to save energy and control temperature [8]. Like task scheduling, V-F selection for the cores is also NP-complete [118]. Therefore, efficient DTPM techniques are essential to utilize DSAs efficiently while maintaining the chip temperature within limits [83].

The DTPM techniques also fall into categories similar to the task scheduling algorithms, namely optimization-, heuristic-, and machine learning-based techniques. The extensive use of heterogeneous multiprocessor systems-on-chip (MPSoCs) in battery and energy-constrained systems has attracted substantial research in this domain. The heuristic techniques proposed in [119] and [120] use the difference between achieved and target metrics to adjust the frequencies. The approach presented in [121] combines the benefits of traditional control theoretic approaches and heuristics to develop a lightweight and efficient frequency scaling policy. The control theoretic approach presented in [122] proposed a DVFS technique that calculates the change in application execution time with change in frequency to honor soft deadline constraints. Recent approaches presented in [123, 113, 124] use machine learning to train policies to determine the optimal operating frequency and generalize to unseen workloads. Temperature management on heterogeneous SoCs is also critical as the on-die power density critically increases [116, 125, 126]. Current approaches for power and thermal management techniques focus on homogeneous and heterogeneous CPU cores and also on GPUs, comprehensively discussed in [127]. A few techniques consider hardware accelerators in their power management policies [128, 129]. However, DSAs demand novel techniques that consider all types of hardware accelerators and specialized cores since they can significantly contribute to the overall power, energy, and temperature.

2.3.3 Other Resource Management Research Directions

While task scheduling, mapping, and DTPM ideas dominate the primary aspects of resource management, modern-day SoCs look at other aspects to satisfy requirements such as reliability and security to meet user expectations and privacy standards. For instance, SoCs and processors from Apple, Intel, and RISC-V include a secure enclave to protect user data when the platform is experiencing security attacks [130, 131, 132]. Instead of relying solely on the secure enclave to protect sensitive data, building security into other aspects enhances the security of the design. DSAs seek adaptation and advancement of ideas from prior work on heterogeneous MPSoCs.

Risk and Security: Integrating security into scheduling algorithms and dynamic voltage frequency scaling (DVFS) governors is at the expense of chip area, scheduling latency, power and energy overheads, and design complexity. Therefore, low complexity and runtime overheads remain essential requirements of security-aware techniques. Commercial SoCs integrate several third-party IPs to promote design reuse and improve the design cycle [133]. However, using external IPs has the severe risk of untrusted designs, leading to security flaws. To this end, a multi-dimensional optimization approach improves the security of the MPSoC through task scheduling with negligible impact on performance and no additional hardware cost [134]. In this approach, task duplication and isolation are the two techniques that aid in detecting hardware trojans in the presence of third-party IPs. Recent literature has shown that security attacks can extract sensitive data by exploiting the temperature patterns on the chip [135]. Indeed, the ThermalAttackNet [136] discusses the potential of DVFS governors in avoiding the detection of stored passwords using on-chip temperature patterns. Therefore, integrating such techniques into resource management algorithms improves the security of DSAs.

Reliability and Robustness: With the increasing use of SoCs in safety-critical applications (such as autonomous driving, avionics, and medical applications), there is a critical emphasis on reliable and robust computing [137, 138]. The chip temperature plays a critical role in the mean time to failure since it directly impacts the metal fatigue. A tradeoff between power consumption and reliability (in the mean time to failure) is explored by estimating the failure rate at a given chip temperature in [139]. The DVFS technique proposed in this paper integrates a reliability metric into its optimization problem to increase the mean time to failure. Furthermore, differential aging of cores in an SoC results in certain cores failing sooner than the other counterparts. To address this challenge, the scheduling approach presented in [140] includes the mean time to failure in the objective function along with deadline constraints. While the above techniques focus on prolonging the time to failure, task scheduling techniques also provide reliable and robust decisions in the presence of system faults [141, 142]. For space-based applications, state-of-the-art approaches must provide reliable and robust decision in high-radiation environments [143, 142, 144]. In summary, emerging reliability and robustness requirements demand resource management approaches for DSAs to take them into consideration to enhance usability.

2.4 Incremental and Online Updates to Decision Tree Classifiers

DSAs integrate multiple heterogeneous processing elements and offer massive parallelism to execute several jobs in parallel [6, 22]. Task scheduling algorithms must effectively allocate the tasks to the processing elements at runtime to fully exploit their potential [23]. Scheduling policies range from simple lookup tables to static and runtime heuristic schedulers [1, 145, 102, 146, 147]. More recently, DT based machine learning schedulers have attracted significant interest since they provide interpretability and ultra-low inference latencies, making them highly suitable for DSA resource management [49, 52, 148].

Current ML approaches are *static*, i.e., their rules do not change at runtime. Consequently, these models cannot adapt to workload changes and new scenarios.

Therefore, DT classifiers used for resource management must possess the capability to adapt on-the-fly to the variations in the environment, which are inevitable. Since DTs are conventionally batch trained using the CART algorithm, there is a strong need for novel techniques to allow DTs to be updated online [149]. To this end, we categorize the prior work on adaptive, online, and incremental updates to DTs into three broad categories, and compare them qualitatively in Table 2.1.

Reinforcement Learning (RL): Neural networks use stochastic gradient descent (SGD) to update the model parameters in multiple batches. Differentiable and fuzzy approaches utilize SGD approaches to update the DT parameters [150, 3, 151]. After converting a DT to a differentiable model, RL and SGD algorithms can update the parameters when new applications or SoC configurations are encountered. However, RL suffers from significant drawbacks because of: (1) excessive time needed to explore the search space for large problem sizes (such as DSAs), (2) high computational power required to perform SGD iteratively for several iterations, (3) the complexity in designing a good reward function, and (4) difficulty in converging to a well-performing policy [49, 33].

Ensemble Decision Tree Algorithms: Ensemble DT algorithms [4, 152] comprise two key steps: (1) training multiple weak DTs, and (2) combining the weak predictions to obtain a final prediction [153]. Resource management applications target inference latencies in the order of tens of nanoseconds [52]. Hence, executing multiple inferences and combining their predictions in DT ensembles have prohibitive latency overheads, as quantified in Chapter 6.

Hoeffding Trees: Hoeffding trees are used when the training sample size exceeds

Table 2.1: Comparing INDENT with prior work on DT training.

Technique	**Allows Online Update**	**Computation and Memory Resources**	**Training Time**	**Inference Latency**	**Ease of Convergence**
CART DT [149, 4]	No	Low	Low	Low	High
RL [3]	Yes	High	High	Low	Low
DT Ensembles [4, 152, 154]	Yes	Low	Low	High	Moderate
Hoeffding Trees [2]	Yes	Low	Low	Low	Low

the memory capacity. They stream the data samples, i.e., the algorithm only observes a portion of the data at a time to construct the DT [2]. However, they rely on a critical assumption: the data distribution does not change over time, and a small set of observed samples is representative of the entire dataset [155]. However, this vital assumption does not apply to SoC resource management. First, the training samples are not independent and identically distributed since they depend on previous scheduling decisions. Second, the initial training samples are not representative of the new applications and additional processing clusters. Therefore, the Hoeffding tree family of algorithms does not converge to a model that performs well for resource management applications; hence is not suitable for online updates [156, 155].

To the best of our knowledge, INDENT is *the first approach to incrementally update a single DT model online*. INDENT stores just 1-8% of the original training data and yet performs within 5% of a DT that was trained with the entire training data.

3 DS3: A DOMAIN-SPECIFIC SYSTEM-ON-CHIP SIMULATION FRAMEWORK

3.1 Background, Motivation and Contributions

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 [56]. 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 [30]. Similarly, SoCs optimized for autonomous driving applications integrate deep neural network (DNN) accelerators [157]. Then, a wide range of design- and run-time algorithms are employed to schedule the applications to the processing elements (PEs) in the DSSoC [61, 58, 59, 60]. Finally, dynamic power and thermal management (DTPM) techniques optimize the SoC for energy efficient operations at runtime. Throughout this process, evaluation frameworks, ranging from analytical models and hardware emulation, are needed to explore the design space and ensure that the DSSoC achieves performance, power and energy targets [158].

Full-system simulators, like gem5 [159], have the ability to perform instruction-level cycle-accurate simulation. However, this level of detail leads to long execution

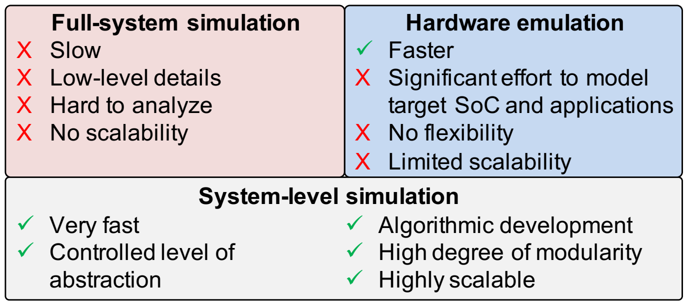


Figure 3.1: Common DSE methodologies with their advantages and disadvantages.

times, in the order of hours to simulate a few milliseconds of workloads [160]. Hence, they are not suitable for rapid design space exploration (DSE). It is also important to note that the level of detail provided by cycle-accurate simulations is beyond the requirements of high-level design space exploration. The most critical system-level questions are *where tasks should run* and *how fast PEs should operate* to satisfy the design requirements, e.g., maximizing performance per Watt (PPW) or energy-delay product (EDP). On the other hand, hardware emulation using Field-Programmable Gate Array (FPGA) prototypes are substantially faster [161]. However, they involve significantly higher development effort to implement the target SoC and applications (see Figure 3.1). Given the design complexities and the cost of considering a large design space, there is a strong need for a simulation environment which allows rapid, high-level, simultaneous exploration of scheduling algorithms and power-thermal management techniques, both of which can significantly influence energy efficiency.

In this work, 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. 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 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 [162] and Odroid-XU3 [14], and included as a benchmark suite in DS3 distribution. The benchmark suite enables realistic design space explorations, as we demonstrate in this work.

The major contributions of this chapter include:

- A unified, high-level DSSoC simulator, DS3, that enables design space exploration together with scheduling and DTPM strategies,
- A benchmark suite of real-world applications and their reference hardware implementations and
- Extensive design space exploration studies for fine-grained architecture tuning.

3.2 Overall Goals and Architecture

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.

The organization of the DS3 framework designed to accomplish these objectives is shown in Figure 3.2. 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

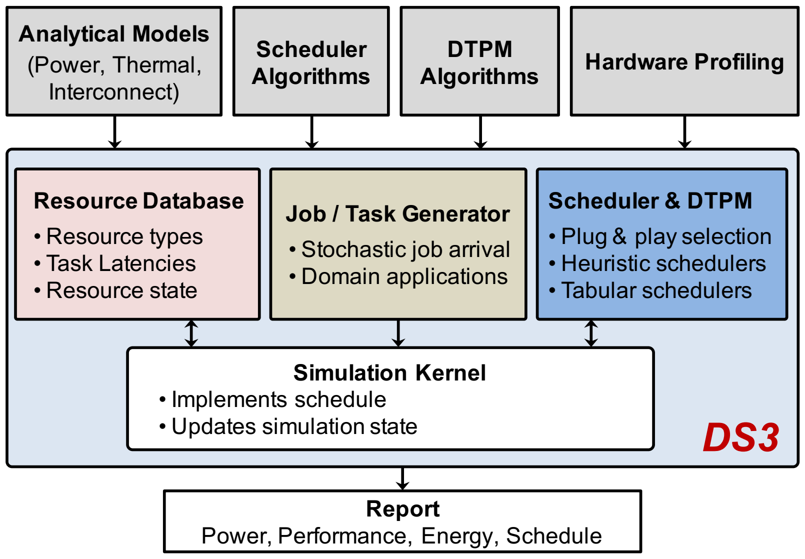


Figure 3.2: Organization of DS3 framework describing the inputs and key functional components to perform rapid design space exploration and validation.

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 [163]. 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 [116]. 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 following two sections present the implementation details and capabilities for new developers and users, respectively.

3.3 Developer View: DS3 Implementation

This section describes the implementations of the DS3 components (Figure 3.2) from a developer's perspective.

3.3.1 Resource Database

DS3 enables instantiating a wide range of SoC configurations with different types of general- and special-purpose PEs. A list of PEs and its characteristics are stored in the resource database. Each PE in the database has the *static* and *dynamic attributes* described in Table 3.1.

The *static* and *dynamic* attributes are determined based on the current industry practice and upon a careful examination of available systems on the market. For example, CFS scheduler, the default Linux kernel scheduler [164], makes task mapping decisions based on the utilization of PEs. In addition, ARM big.LITTLE architecture [116], combining Cortex-A15 cluster with energy-efficient Cortex-A7 cluster, supports different operating frequencies for each cluster. The voltage level and thus energy consumption depend on the operating point and DS3 takes these effects into account. Finally, commercial SoCs utilize DVFS policies [63] to control power and performance of PEs. For this reason, we integrated these policies into

Table 3.1: List of PE attributes in resource database

	Attribute	Description
Static	**Type**	Defines type of PE Example: CPU, accelerator etc.
	Capacity	Number of simultaneous threads a PE can execute
	DVFS policy	Policy which controls PE frequency and voltage at runtime
	Operating performance point	Operating frequencies and corresponding voltages for a PE
	Execution time profile	Defines the execution time of supported tasks on each PE
	Power consumption profile	Provides the power consumption profile of each PE
Dynamic	**Utilization**	Defines active time of a PE for a particular time window
	Blocking	Probability that a PE is busy when a task is ready
	State	Indicates whether a PE is busy or idle

DS3 and assigned the current DVFS policy as an attribute to a PE. This list in Table 3.1 can be extended either by defining a new parameter in the corresponding SoC file and parsing it, or assigning as an attribute in the *PE* class of DS3 framework.

3.3.2 Job Generator

Figure 3.3 presents block diagrams for a WiFi transmitter (WiFi-TX) and receiver (WiFi-RX) both of which are composed of multiple tasks. The tasks and dependencies in an application are represented using a DAG. The job generator produces the tasks shown in Figure 3.3, for a WiFi-TX job along with the dependencies. The basic unit of data processed by this chain is a frame, which is 64 bits in this work. DS3 defines each new input frame of an application as a job. Hence, each job is a

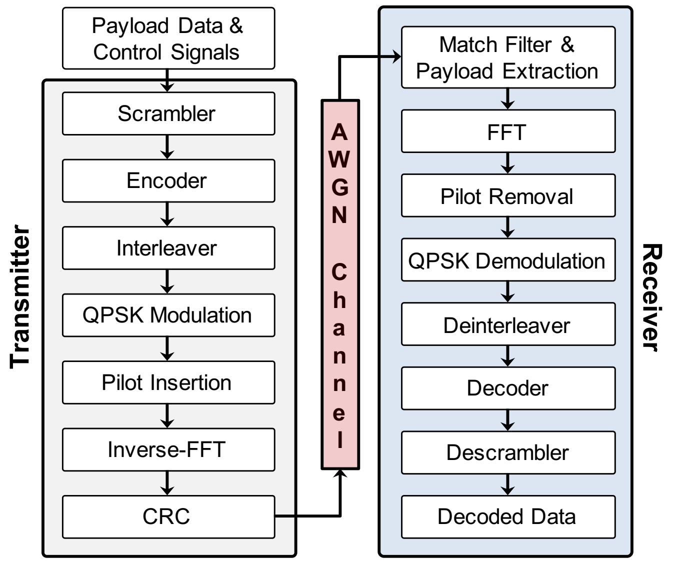


Figure 3.3: Block diagrams for WiFi-TX and WiFi-RX applications.

64-bit frame streaming through the WiFi-TX chain.

The job generator produces the tasks and DAG for each active application following a user-specified job injection model. DS3 currently models the traffic by injecting jobs based on an exponential distribution. The framework provides the flexibility to model other distributions as well. DS3 is scalable in terms of job generation and is capable of spawning jobs from multiple applications. For example, suppose that both WiFi-TX and WiFi-RX applications are active and the corresponding injection ratio is 0.8:0.2. On an average, DS3 generates 4 WiFi-TX jobs for every WiFi-RX job. This capability plays a crucial role in exploring mix of multiple workloads, as demonstrated in Section 3.6.

3.3.3 Scheduling and DTPM Algorithms

DS3 provides a plug-and-play interface to choose between different scheduling and DTPM algorithms. Hence, developers can implement their own algorithms and easily integrate them with the framework. To achieve this, developers define the new scheduling algorithm as a member function of the *Scheduler* class. Then, the new scheduler is invoked from the *run* method of the simulation core. Scheduling algorithms vary significantly in their complexities and hence, require different inputs to map tasks to PEs. To support this, DS3 provides a loosely defined interface to specify inputs to the schedulers as required. The framework supports list schedulers (such as HEFT [1] as well as table-based schedulers (such as constraint programming), where schedule for all the tasks in a job is generated at the time of job injection.

DS3 provides built-in DTPM policies and facilitates the design of new DTPM algorithms. The policy is invoked periodically at every control epoch, which is parameterizable by the user. To minimize the run-time and power overhead of DTPM decisions, we use 10ms–100ms range following the common practice [116]. A policy of low complexity may use only the power state information of the PEs. On the other hand, advanced algorithms may use PE utilization and more detailed performance metrics, such as number of memory accesses and retired instructions. In addition, the DTPM policies have access to the resource management, including the power consumption and performance profiles therein. The decisions of the DTPM policy are evaluated and applied to the PEs at every control epoch.

Developers can add new scheduling and DTPM algorithms without modifying

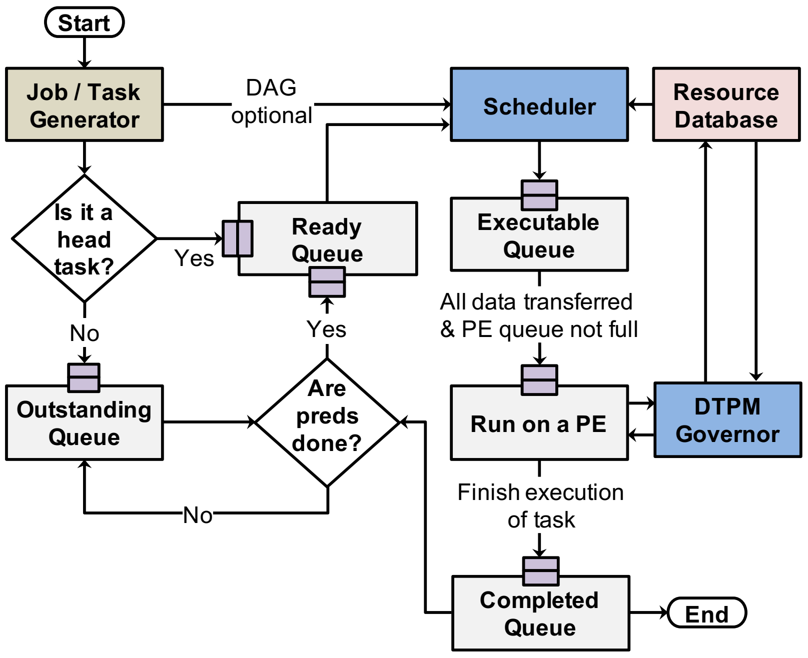


Figure 3.4: Life-cycle of a task in DS3 queues.

the rest of DS3. Modular design enables both maintaining existing interfaces and expanding them to support radically different algorithms.

3.3.4 Simulation Kernel

The life cycle of a task in DS3 is shown in Figure 3.4. The job generator constructs a task graph as described in Section 3.3.2. The tasks that are ready to 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 [163, 165]. 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 [166]. DRAMSim2 is used to obtain memory latencies at varying bandwidth requirements as shown

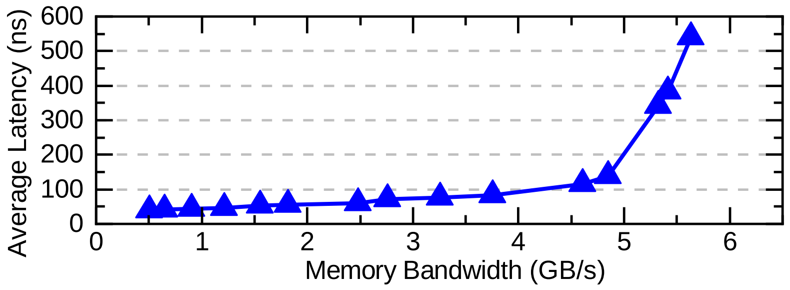


Figure 3.5: Bandwidth-Latency curve used to model DRAM latency in DS3 framework.

in Figure 3.5. 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.

The simulation kernel also calls the DTPM governor periodically at every decision epoch. The DTPM governor determines the power states of the PEs as a function of their current load and information provided by the resource database. Subsequently, the simulation kernel updates the power states of the PEs and the decisions are retained until the next evaluation at the next epoch.

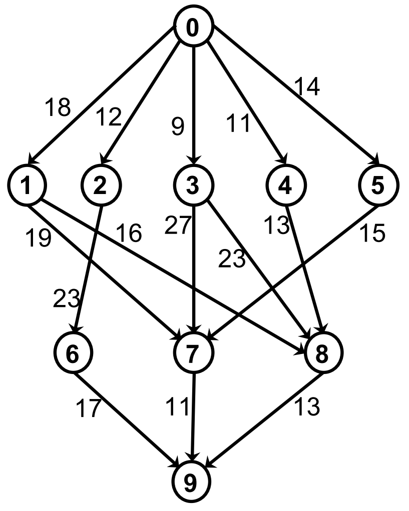
3.4 User View: DS3 Capabilities

This section presents the built-in scheduling and DTPM algorithms provided by the framework.

3.4.1 Scheduling Algorithms

DS3 provides a set of commonly used built-in scheduling algorithms, which can be specified by the users in the main configuration file. The framework generates Gantt charts to visualize the schedulers (see Figure 3.7). This allows the end-user to understand the dynamics of the scheduler under evaluation. We describe the built-in scheduling algorithms in DS3 using one of the most commonly used canonical task graphs [1] shown in Figure 3.6. Since this task graph is used commonly as reference for many list-scheduling studies, it serves as a representative example before analyzing the results from real-world applications in Section 3.6.3.

Minimum Execution Time (MET) Scheduler: The MET scheduler assigns a ready task to a PE that achieves the minimum expected execution time following a FIFO policy [167]. If there are multiple PEs that satisfy the minimum execution criterion, the scheduler then reads the current state information of all these PEs and assign the tasks to one of the most idle PEs. Figure 3.7(a) shows the schedule generated by the MET scheduler in DS3 for the DAG shown in Figure 3.6. All tasks are assigned to their best-performing PEs as expected.



Task	P0	P1	P2
0	14	16	9
1	13	19	18
2	11	13	19
3	13	8	17
4	12	13	10
5	13	16	9
6	7	15	11
7	5	11	14
8	18	12	29
9	21	7	16

Execution times of the tasks on the different PEs for the representative example.

Figure 3.6: A canonical task flow graph [1] with 10 tasks. Each node represents a task and each edge represents average communication cost across the available pool of PEs for the pair of nodes sharing that edge. The computation cost table on the right indicates the execution time for each of the PEs.

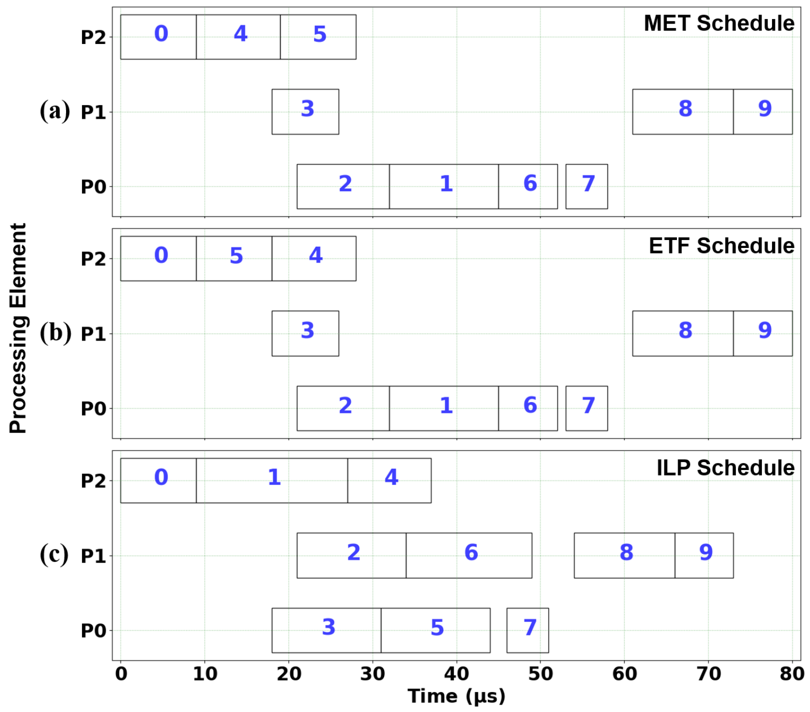


Figure 3.7: Schedule of task graph in Figure 3.6 with (a) MET, (b) ETF, and (c) CP.

Earliest Task First (ETF) Scheduler: The ETF scheduler utilizes the information about the communication cost between tasks and the current status of all PEs to make a scheduling decision [168]. Figure 3.7(b) shows the schedule produced by DS3 for a single instance of DAG shown in Figure 3.6 with ETF scheduler. Although the execution times are the same for both MET and ETF based schedules, the mapping decisions are different as shown in Figure 3.7(a) and 3.7(b). This difference becomes evident when multiple applications are executed together, as shown in Section 3.6.3.

Table-based Scheduler: DS3 also provides a scheduler which stores the scheduling decisions in a look-up table. This allows users to utilize any offline schedule, such as an assignment generated by an constraint programming (CP) solver, with the help of a small look-up table. For example, we use IBM ILOG CPLEX Optimization Studio [169] to generate the CP solution for a job. The schedule from the CP solution depicted in Figure 3.7(c) outperforms the other two schedulers for a single job instance as expected. However, we note that the schedule stored in the table guarantees optimality only if there is a single job in the system. Hence, its performance degrades when multiple jobs overlap, as shown in Section 3.6.3.

3.4.2 DTPM Policies

State-of-the-art SoCs support multiple voltage-frequency domains and DVFS, which enables users to optimize for various power-performance trade-offs. To support this capability, DS3 allows each PE to have a range of OPPs, configurable in the resource database. The OPPs are voltage-frequency tuples that represent all supported frequencies of a given PE, which can be exploited by DTPM algorithms to tune the SoC at runtime. In addition, DS3 integrates analytical power dissipation and thermal models for Arm Cortex-A15 and Cortex-A7 cores [116], and power profiles for FFT [170], scrambler encoder, and Viterbi accelerators [171].

The power models capture both dynamic and static power consumption. The dynamic power consumption ($P = CV^2Af$) varies according to the load capacitance (C), supply voltage (V), activity factor (A), and operating frequency (f). Voltage and frequency are modeled through the OPPs, while the load capacitance and activity factors are modeled using measurements on real devices and published data. Static power consumption depends mainly on the current temperature and voltage, and DS3 uses thermal models obtained from measurements in the Odroid-XU3 SoC to accurately model both power and temperature.

The integrated power, performance, and temperature models enable us to implement a wide range of DTPM policies using DS3. To provide a solid baseline to the user, we also provide built-in DVFS policies that are commonly used in commercial SoCs. More specifically, users use the input configuration file to set the DTPM policy to *ondemand*, *performance* and *powersave* [63] governors, or to a custom DTPM governor.

Ondemand Governor: The *ondemand* governor controls the OPP of each PE as a function of its utilization. The supported voltage-frequency pairs of a given PE are given by the following set:

$$\mathcal{OPP} = \{(V_1, f_1), (V_2, f_2), \dots, (V_k, f_k)\} \tag{3.1}$$

where k is the number of operating points supported by that PE. Suppose that the PE currently operates at (V_2, f_2). If the utilization of the PE is less than a user-defined threshold, then the *ondemand* governor decreases the frequency and voltage such that the new OPP becomes (V_1, f_1). If the utilization is greater than another user-defined threshold, the OPP is increased to the maximum frequency. Otherwise, the OPP stays at the current value, i.e., (V_2, f_2).

Performance Governor: This policy sets the frequency and voltage of all PEs to their *maximum* values to minimize execution time.

Powersave Governor: This policy sets the frequency and voltage of all PEs to their *minimum* values to minimize power consumption.

User-Specified Values: This policy enables users to set the OPP (i.e., frequency and voltage) of each PE individually to a constant value within the permitted range. It enables thorough power-performance exploration by sweeping the OPPs. Finally, developers can also define custom DTPM algorithms in the *DTPM* class, similar to the scheduler.

3.5 Simulator Validation

Simulation frameworks serve as powerful platforms to perform rapid design space exploration, evaluation of scheduling algorithms and DTPM techniques. However, the fidelity of such simulation frameworks is questionable. Particularly, the level of abstraction in high-level simulators is significant. Hence, estimations from simulations may diverge due to differences in modeling, ineffective representation and limitations to capture overheads observed on hardware platforms. In this section, we comprehensively evaluate our DS3 framework in terms of performance, power and temperature estimations with two commercial SoC platforms — Odroid-XU3 and Zynq Ultrascale+ ZCU102.

3.5.1 Validation with Odroid-XU3

We choose Odroid-XU3 as one of the platforms for validation because of its abilities to measure power, performance and temperature. This platform comprises in-built current and temperature sensors enabling us to measure power, performance and temperature simultaneously and accurately at runtime. Since the design space

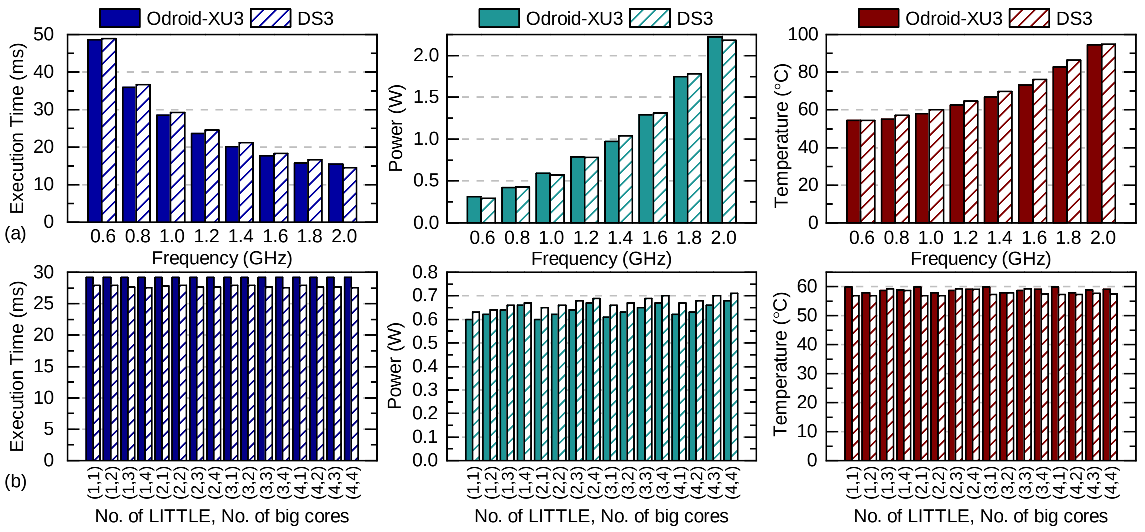


Figure 3.8: Comparison of execution time, power and temperature between DS3 and Odroid-XU3 for single-threaded applications when (a) Freq-Sweep: Number of cores is constant, frequencies of the cores are varied (b) Core-Sweep: Frequencies of cores are constant, number of cores is varied.

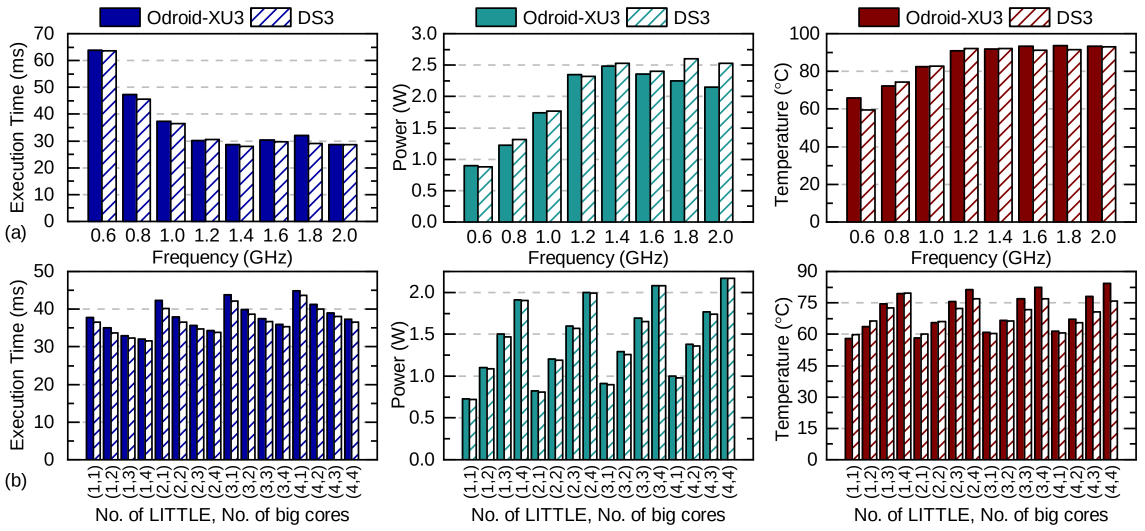


Figure 3.9: Comparison of execution time, power and temperature between DS3 and Odroid-XU3 for multi-threaded applications when (a) Freq-Sweep: Number of cores is constant, frequencies of the cores are varied (b) Core-Sweep: Frequencies of cores are constant, number of cores is varied.

Table 3.3: Error percentages in comparison of execution time, power and temperature between DS3 and Odroid-XU3

Application Type	Scenario	Execution Time	Power	Temperature
Single Threaded	Freq-Sweep	3.6%	3.1%	2.7%
	Core-Sweep	5.3%	5.1%	2.1%
Multi Threaded	Freq-Sweep	2.8%	6.1%	2.4%
	Core-Sweep	2.7%	1.3%	3.8%

comprising the number of cores, frequency levels and applications is very large, we choose representative configurations and applications for validation against the Odroid-XU3, as shown in Figure 3.8 and Figure 3.9. For a comprehensive validation, we consider two cases: (1) Freq-Sweep: the number of cores is fixed, frequencies of the cores are varied and (2) Core-Sweep: the frequencies of the cores are fixed, the number of cores is varied. To ensure completeness in validation, we validate both single-threaded and multi-threaded applications in both scenarios. For single-threaded applications, Figure 3.8(a) describes the execution time, power and temperature for Freq-Sweep scenario and Figure 3.8(b) for Core-Sweep. For Freq-Sweep, we vary the frequency from 0.6-2.0 GHz. Odroid-XU3 has four LITTLE cores and four big cores, leading to 16 possible combinations of configurations of active cores. Core-Sweep compares the parameters of DS3 and Odroid-XU3 for all 16 combinations. Similarly, Figures 3.9(a) and (b) show the comparisons of DS3 and Odroid-XU3 for multi-threaded applications. Table 3.3 shows the error percentages in comparison of execution time, power and temperature for single-threaded and multi-threaded applications in both Freq-Sweep and Core-Sweep configurations.

In Freq-Sweep scenario, the frequency of the LITTLE and big cores are varied in unison until we reach the maximum frequency of the LITTLE cores, which is 1.4 GHz. Thereafter, the frequency of the big cores are swept until the maximum frequency of 2.0 GHz. From Table 3.3, we observe that the average error in performance, power and temperature estimates are *3.6%*, *3.1%*, and *2.7%*, respectively, for single-threaded applications. Similarly, the errors are *2.8%*, *6.1%*, and *2.4%* for multi-threaded applications.

In Core-Sweep scenario, the number of active cores is varied in all combinations while the frequencies of the LITTLE and big cores are retained at 1.0 GHz. The performance, power and temperature mean absolute errors are *5.3%*, *5.1%*, and *2.1%*, respectively, for single-threaded applications while multi-threaded applications have an average error of *2.7%*, *1.3%*, and *3.8%*.

On an average, the error in accuracy is mostly less than 6%. We note that the platform experiences frequency throttling when the temperature reaches trip points (95C). The throttling behavior is modeled in DS3 and hence, we obtain highly correlated estimates for execution time, power and temperature even when the platform is throttled by the on-board thermal management agent.

In summary, the estimates from DS3 closely match real-time measurements obtained by the execution of similar workloads on the platform, as summarized in Table 3.3. The strong validation results aid in reinforcing the fidelity of the framework in simulating DSSoCs with high accuracy.

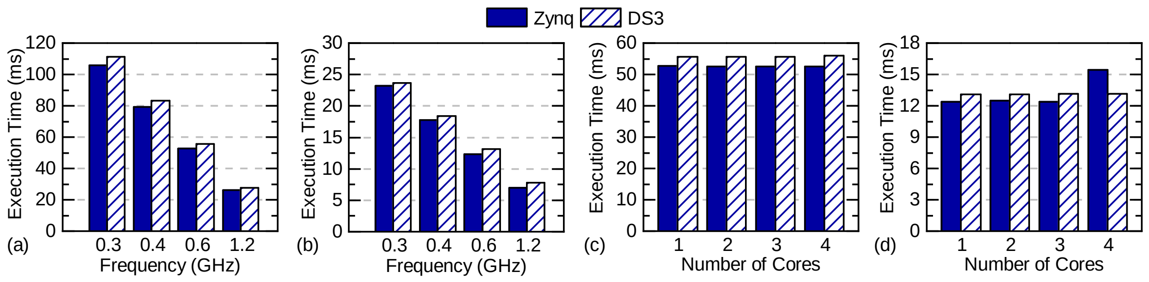


Figure 3.10: Comparison of execution time between DS3 and Zynq MPSoC when (a) Freq-Sweep with only Cortex A53 cores, (b) Freq-Sweep with Cortex A53 cores and hardware accelerators (c) Core-Sweep with only Cortex A53 cores, (d) Core-Sweep with Cortex A53 cores and hardware accelerators.

3.5.2 Validation with Zynq Ultrascale+ ZCU102

The second platform used to validate the results of the DS3 framework is Zynq Ultrascale+ ZCU102 FPGA SoC. Zynq serves as a crucial platform for validation as it supports the implementation of hardware accelerators, unlike Odroid-XU3. The support for hardware accelerators aids in validating DS3 against highly heterogeneous SoCs. However, lack of on-board sensors prevent us from accurately measuring power and temperature. Hence, we chose to validate the execution time of Zynq and DS3 in the presence of hardware accelerators in various scenarios.

We pick multiple scenarios to validate the execution time. First, we sweep the frequencies across the four supported frequencies on the Zynq board, which is the Freq-Sweep scenario. We then measure the execution times when applications are executed only on Cortex A53 cores on the platform and then with both A53 cores and hardware accelerators. Secondly, we measure the execution times in Core-Sweep scenario with only A53 cores, and with A53 cores and hardware accelerators. Figure 3.10 shows high correlation between the measurements obtained from the

Zynq board and DS3. However, when four A53 cores and hardware accelerators are enabled, we observe an anomaly with 15% error between DS3 and Zynq.

Current Linux kernels do not support scheduling and enablement of hardware accelerators in the operating system. Hence, we implement the scheduling mechanism for accelerators in user-space and identify the anomaly as an overhead incurred due to the user-space implementation. This overhead is expected to be significantly minimized when operating systems include support for accelerators. Finally, the average error in execution time is 6.85%.

3.6 Application Case Studies

This section presents case studies and experiments for design space exploration of dynamic resource management, power-thermal management, and architecture configurations. We base our studies on the benchmark applications, which are presented in the following section.

3.6.1 Benchmark Applications

DS3 comes with *six reference applications* from wireless communications and radar processing domain:

- **WiFi-TX/RX**,
- **Low-power single-carrier TX/RX**, and
- **Radar and Pulse Doppler**.

Table 3.4: 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	

The WiFi protocol consists of transmitter and receiver flows as shown in Figure 3.3. It has compute-intensive blocks, such as FFT, modulation, demodulation, and Viterbi decoder (see Table 3.4), which require a significant amount of system

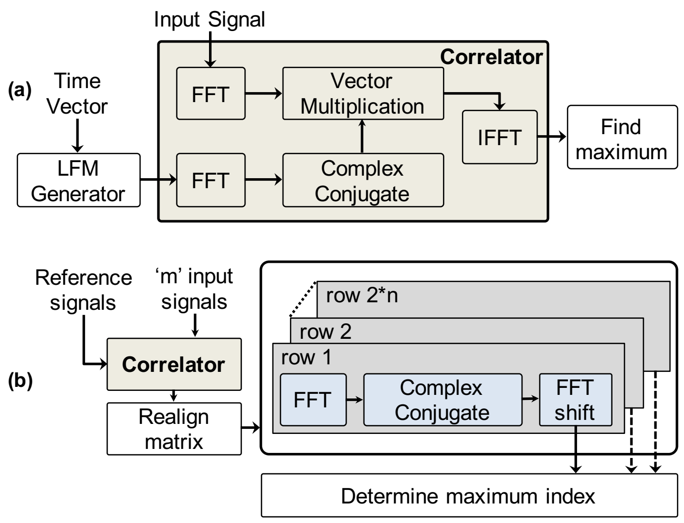


Figure 3.11: Block diagram of (a) range detection application, (b) pulse Doppler application where *m* is number of signals and *n* is number of samples for a signal.

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 3.4). Figure 3.11(a) and Figure 3.11(b) represents block diagrams of the range detection and pulse Doppler applications, respectively.

The benchmark applications enable various algorithmic optimizations and realistic design space exploration studies, as we demonstrate in this chapter. We will continuously include applications from other domains to the benchmarks.

3.6.2 Reference Design of Applications

We developed a reference design for each of the applications described in Section 3.6.1 on two popular commercial heterogeneous SoC platforms: Xilinx Zynq ZCU102 UltraScale MpSoC [162] and Odroid-XU3 [14] which has Samsung Exynos 5422 SoC. The nodes of the DAG, i.e., the tasks that constitute the target application, are scheduled on the processing elements. Since the tasks in the DAG determine the scheduling and acceleration granularity, they should be coarse enough to offset the overheads and produce benefits. In the scope of DS3, we implement hardware accelerators in the programmable logic (PL) of the Xilinx Zynq SoC. Depending on the size of data transfers required for the accelerator, we use either memory-mapped (AXI4-Lite) or streaming interfaces (AXI-Stream). To be specific, we use memory-mapped interfaces to communicate with scrambler-encoder accelerators and stream interfaces for the FFT accelerators. A direct memory access (DMA) unit facilitates data transfers between user-space mappable memory buffers and the accelerators using a streaming interface. In this regard, we profiled the computation and communication times in a Linux environment running on the Zynq SoC. The speedup of scrambler-encoder accelerator is 2.75× in comparison to the performance on Arm A53 of Zynq SoC. On the other hand, the speedup of the FFT accelerator is ~19× when comparing the total latency (computation and communication) with that of Arm A53 on Zynq SoC. The streaming interface allows for significantly improved data transfer latencies. The transfer of data by the DMA unit through the user-space mappable memory buffers can further be improved by enabling caching and higher-bandwidth interfaces in the Zynq SoC. Hence, the Viterbi decoder accelerator is assumed to have a highly efficient data transfer mechanism and consequently the speedup is three orders of magnitude (see Table 3.4).

The latency of each task in every application on different resource types of these two platforms is profiled. At runtime, the schedulers decide to allocate the task to either hardware accelerators or general-purpose cores based on their corresponding function of communication- and computation-times along with other system parameters. In addition to latency profiling, we used the power consumption and temperature sensors on the Odroid-XU3 board for power consumption profiling. DS3 power/performance models used in the resource manager incorporate these performance profiles for each task-resource pair.

3.6.3 Scheduler Case Studies

This section provides an extension to our previous work in [172], using built-in DS3 schedulers and applications in the benchmark suite. For the simulations, 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 3.12. 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 3.12(a) and (b) depict the results with WiFi applications for a download

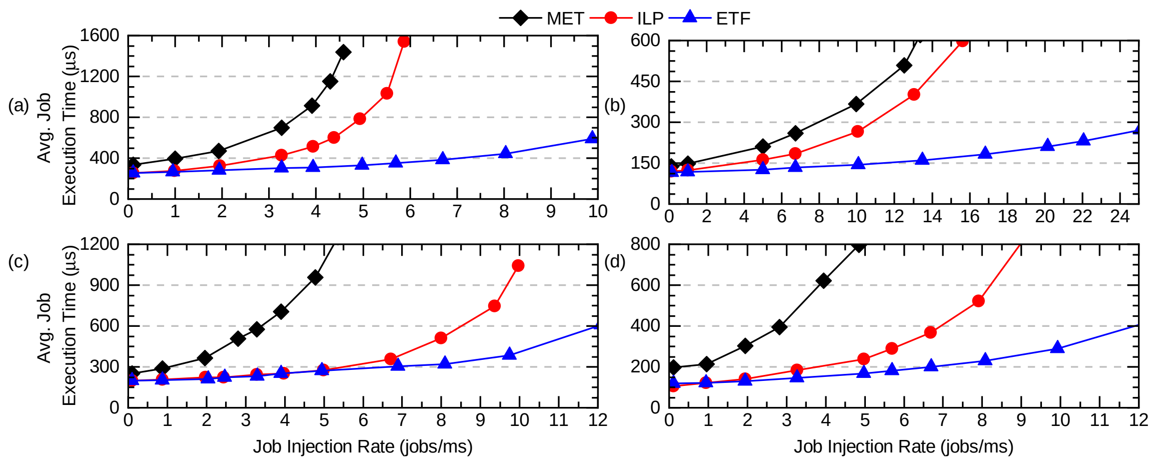


Figure 3.12: Results from different schedulers with a workload consisting of (a) WiFi-TX ($\mathbf{p_{TX}}$=0.2) and WiFi-RX ($\mathbf{p_{RX}}$=0.8), (b) WiFi-TX ($\mathbf{p_{TX}}$=0.8) and WiFi-RX ($\mathbf{p_{RX}}$=0.2), (c) range detection ($\mathbf{p_{range}}$=0.8) and pulse Doppler ($\mathbf{p_{pulse}}$=0.2), (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).

and upload intensive workload, independently. To understand the performance of scheduling algorithms, we analyze the average execution time at varying rates of job injection. MET uses a naive representation of the system state for scheduling decisions (described in Section 3.3.3), which results in higher execution time. On the other hand, CP uses a static table-based schedule which is optimal for one job instance. At low injection rates (less than 1 job/ms), CP is suitable as jobs do not interleave. However, as the injection rate increases, the CP schedule is not optimal. ETF scheduler is superior in comparison to the others, as observed in Figures 3.12(a),(b).

Figure 3.12(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 CP schedulers are similar until 5 jobs/ms, following which performance of ETF is superior in comparison to CP. Although the trend in execution time for radar benchmarks is similar to WiFi, the job injection rate at which ETF and CP diverge is different because of the differences in execution times of these applications, as shown in Table 3.5. At an injection rate lower than 5 jobs/ms, the level of interleaving of jobs is low which aligns with the CP solution.

Finally, we construct a workload comprising of all four applications and Figure 3.12(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 CP 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 3.12 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.

Table 3.5: Execution time of applications in benchmark suite with different schedulers

	Execution Time of Single Job (μs)			
	WiFi-TX	**WiFi-RX**	**Range Detection**	**Pulse Doppler**
MET	69	389	177	1665
ETF	69	301	177	1045
CP	69	288	177	1000

3.6.4 SoC Design Space Exploration

This section illustrates how DS3 can be utilized to identify the number and types of PEs during early design space exploration. We employ the WiFi-TX and WiFi-RX applications to explore different SoC architectures. All configurations in this study have 4 big Arm Cortex-A15 and 4 LITTLE Arm Cortex-A7 cores to start with and DS3 guides the user to determine the number of configurable hardware accelerators in the architecture. We choose accelerators for FFT and Viterbi decoder. FFT is a widely used accelerator among all applications. We also choose Viterbi decoder because its execution cost on a general-purpose core is significantly high.

DSE Using Grid Search

We vary the number of instances of FFT (0, 1, 2, 4, 6) and Viterbi decoder (0, 1, 2, 3) in this grid search study. Table 3.6 lists the representative configurations out of 20 configurations. Each row in the table represents the configuration under investigation with an estimated SoC area, and average execution time and average energy consumption per job. The DS3 framework provides metrics that aid the user

Table 3.6: Area, performance and energy for different SoC configurations with varying number of accelerators

Configuration			**Area (mm^2)**	**Average Job Execution (μs)**	**Energy per Job (μJ/job)**
ID	FFT	Viterbi			
1	0	0	14.94	2606	1744
2	0	1	14.94	1824	1244
3	2	1	15.82	293	589
4	4	0	16.29	1212	957
5	4	1	16.56	274	584
6	6	3	19.29	264	582

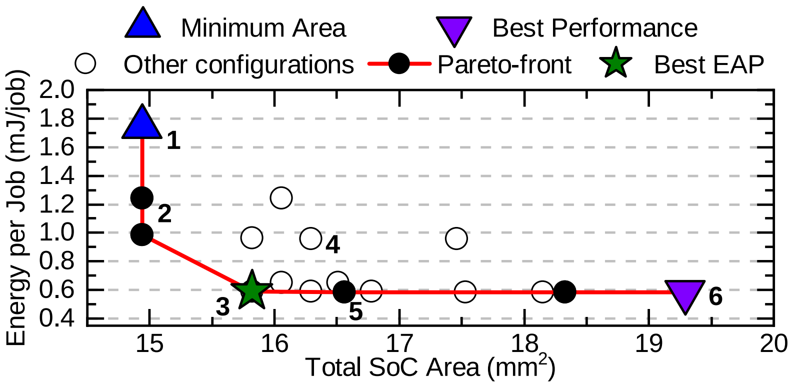


Figure 3.13: Design space exploration studies showing energy per job vs. SoC area with pareto-frontier.

in choosing a configuration that best suits power, performance, area and energy targets.

Figure 3.13 plots the energy consumption per job as a function of the SoC area. We find the area of a given configuration using a built-in floorplanner that takes the areas of PEs and other components such as system level cache and memory controllers. The energy consumption per job is computed as the ratio of total energy consumption for the entire workload with the number of completed jobs.

As the accelerator count increases in the system, the energy consumption per operation decreases. This comes at the cost of larger SoC area, as shown in Figure 3.13 and Table 3.6. For this workload, *configuration-3*, i.e., an SoC with two FFT and one Viterbi decoder accelerators, provides the best trade-off. Removing any of the accelerators leads to a significant increase in energy per operation with a small area advantage. In contrast, any further increase in the number of accelerators does not result in significant improvement in energy per job for this workload. As a result, *configuration-3* is the best configuration in terms of energy-area product

(EAP). This configuration leads to an EAP gain of almost 65% (an energy reduction of 67% with an increase in area by only less than 6%) compared to *configuration-1*. After this point, the improvement in the overall performance by adding more FFT accelerators and Viterbi decoders does not overcome the cost of increase in the total area as seen in both Figure 3.13 and Table 3.6.

DSE Using Guided Search

DS3 also supports a guided search in the design space. Figure 3.14 depicts a 2-D performance plane for a PE (or a PE cluster) where x-axis and y-axis are utilization and blocking, both as percentages, respectively. Ideally, a PE should be on the lower-right corner where utilization is high, and blocking is low. If both utilization and blocking are high, upper-right corner, then it means that there is a need for more resources in the system. If, however, the opposite is true, the utilization of a PE is low, and it also does not block tasks very often. In this case, resources in the system are abundant. Finally, a PE should never be on the upper-left corner, low utilization and high blocking, which is unrealistic.

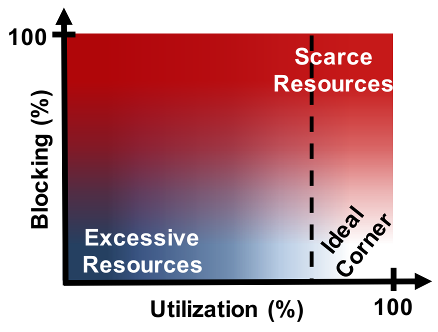


Figure 3.14: PE blocking vs utilization (2-D performance plane)

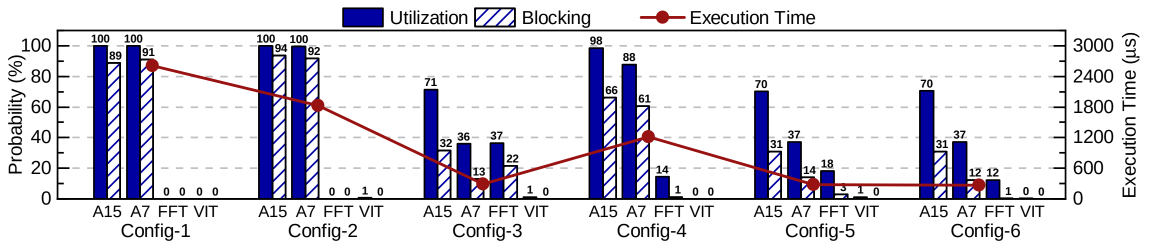


Figure 3.15: Utilization vs blocking for PE clusters in representative configurations with average job execution times.

Considering the case study in Section 3.6.4 where we explore 20 different configurations, the guided search will converge on the best configuration faster. Figure 3.15 shows how utilization, blocking, and average job execution time differ for six representative configurations. *Configuration-1*, with no FFT and Viterbi accelerator, yields a high utilization and blocking for both Arm clusters, hence the SoC requires hardware accelerators. The results with *configuration-2* (addition of one Viterbi accelerator) indicate that the Viterbi accelerator is a critical component for the system since it provides a huge gain in average job execution time (a reduction from 2606 µs to 1824 µs) although the utilization for this accelerator is very small (0.61%). *Configuration-2* also suggests that one Viterbi accelerator is enough for the system since both utilization and blocking is low. Based on this observation, we directly eliminate configurations with no and more than one Viterbi accelerator (i.e., *configuration-4* and *-6*, see Table 3.6). The comparison between *configuration-3* and *-5* based on utilization, blocking, and estimated SoC area draws a conclusion that *configuration-3* is the best configuration for this case study.

Figure 3.16 depicts the same results for the representative configurations on the 2-D plane. As seen, *configuration-3* is closest to the ideal region and provides the

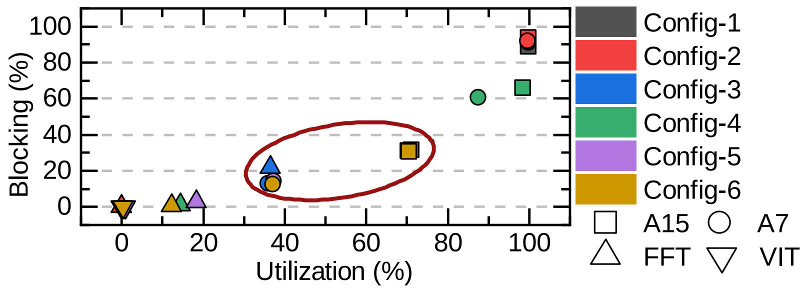


Figure 3.16: Results for representative configurations on 2-D performance plane for PE clusters.

best trade-off.

These approaches can be used to utilize DS3 for design space exploration of SoC architectures by analyzing energy efficiency, area, and developmental effort involved in the development of specialized cores for hardware acceleration. This approach can also be extended to explore the effect of modifying the number of general-purpose cores and hardware accelerators in a DSSoC architecture.

3.6.5 DTPM Design Space Exploration

In this section, we evaluate a subset of five applications from our benchmark set: WiFi-RX/TX, single-carrier RX/TX, and range detection on an SoC with 16 heterogeneous PEs. We explore 8 frequency points for the big cluster (0.6-2.0GHz) and 5 for the LITTLE (0.6-1.4GHz), using a 200MHz step. All possible DVFS modes were evaluated, i.e., all possible combinations of power states for each PE in addition to *ondemand*, *powersave*, and *performance* [63] modes.

Figure 3.17 presents the Pareto frontier for all aforementioned configurations. The *ondemand* and *performance* policies provide low latency with high energy con-

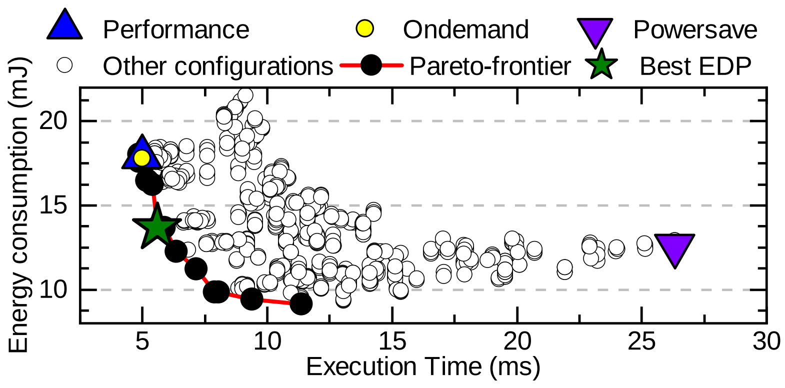


Figure 3.17: Pareto frontier in the energy-performance curve for an SoC with 16 processing elements (PEs) executing a representative workload.

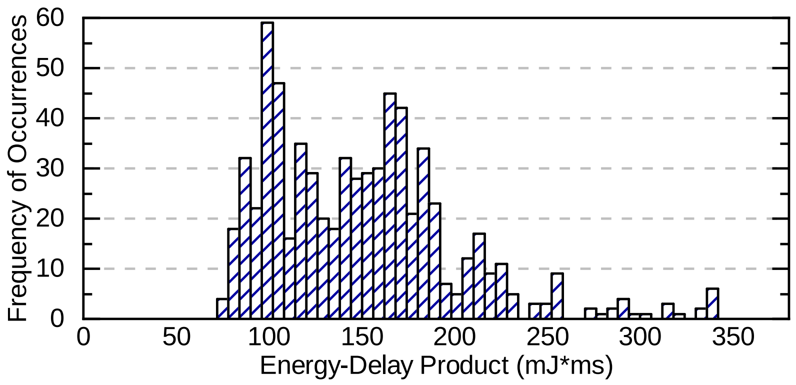


Figure 3.18: Energy-Delay Product (EDP) histogram.

sumption, while *powersave* minimizes the power at the cost of high latency, which results in sub-optimal energy consumption due to the increased execution time. The best configuration in terms of EDP uses 1.6GHz and 4 active cores for the big cluster, and 600MHz and 3 active cores for the LITTLE cluster, achieving 5.6ms and 13.7mJ. This figure also shows a 5$\times$ variation in execution time and energy consumption between different configurations. The best EDP configuration achieves up to 4$\times$ better EDP than default DTPM algorithms. This indicates that there is opportunity for users to propose their own power management mechanisms to improve the energy efficiency of the system and integrate those mechanisms into DS3.

In addition to the Pareto frontier, DS3 also provides energy, performance, and EDP histograms to aid the design space exploration. Due to space constraints, Figure 3.18 depicts only the EDP histogram. This histogram shows that only a small fraction of configurations achieve an EDP below 80mJ*ms. While the *performance* and *ondemand* governors are in the range of 90mJ*ms, the *powersave* mode gets 330mJ*ms EDP. Therefore, users can use these visualization tools to quickly identify the most promising configurations for the SoC.

3.6.6 Scalability Analysis

This section illustrates the scalability of DS3 as a function of simulated number of jobs, SoC size, and the number of tasks in a single job (application). To maximize the load, we run four applications (WiFi TX/RX, range detection and pulse Doppler) simultaneously while sweeping the job injection rates and number of PEs. For the last case, however, we fix the job injection rates and SoC configuration while running applications different in size, separately.

Figure 3.19(a) shows the total simulation time as a function of the number of jobs injected throughout the simulation. As the relation between two metrics is linear, DS3 simulation run-time increases linearly with the workload size. Similarly, Figure 3.19(b) presents the simulation time when the number of PEs increase. This relationship is also linear, leading to 6ms per simulation cycle (1μs) for a 56-core configuration. Finally, in Figure 3.19(c), the simulation time with respect to application size (number of tasks in a single job) is depicted. As application size

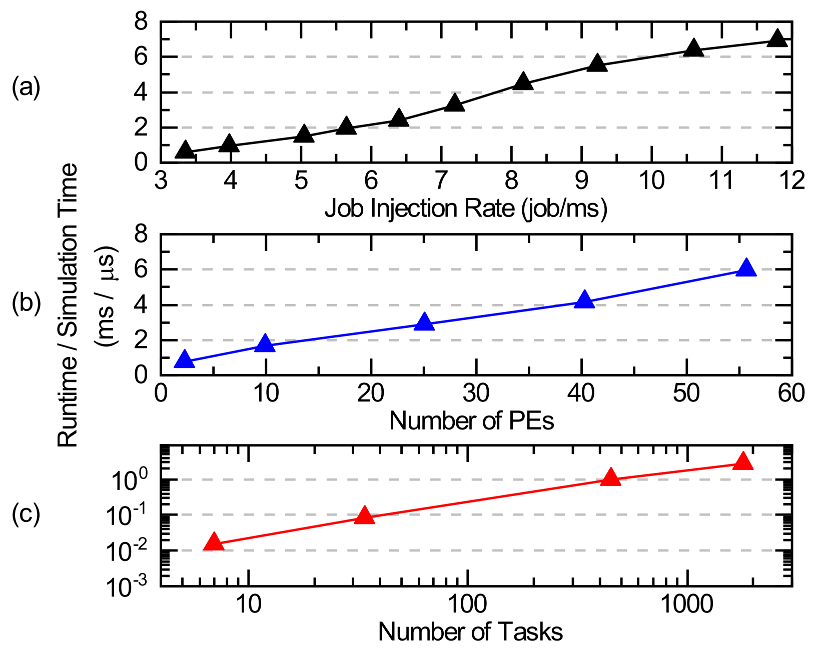


Figure 3.19: Results for scalability analysis showing DS3 runtime versus (a) Different job injection rates, (b) Varying SoC configurations and (c) Number of tasks executed.

grows, the runtime to simulate of 1μs also increases linearly.

The scalability analysis provided in this section demonstrates that DS3 runtime is a linear function of workload, SoC and application size. As a side note, we obtain a simulation speedup of 600× when running 1675 jobs of WiFi-TX in comparison to gem5. Hence, DS3 facilitates rapid design space exploration with relatively short turn-around times. This feature makes DS3 very powerful and helps users with extensive design space exploration in relatively short time while avoiding unnecessary simulation of low-level details.

4 IMITATION LEARNING BASED TASK SCHEDULING FOR HETEROGENEOUS SYSTEMS

4.1 Background, Motivation and Contributions

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 [22]. 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 [6].

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 chapter 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 [1, 103, 100]. Scheduling these tasks to the available PEs is a well-known NP-complete problem [173, 174]. An optimal *static schedule* can be found for small problem sizes using optimization techniques, such as mixed-integer programming (MIP) [175] and constraint programming (CP) [176]. 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) [99] and complete fair schedulers (CFS) [97], 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 chapter 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 [177]. 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 [175]. 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) [178]. 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 [179, 113, 180]. Indeed, RL has shown promise when applied to the scheduling problem [33, 109, 181], but it suffers from slow convergence and sensitivity of the reward function [111, 112]. In contrast, IL takes advantage of the expert's inherent knowledge and produces policies that imitate the expert decisions [110]. 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 chapter are as follows:

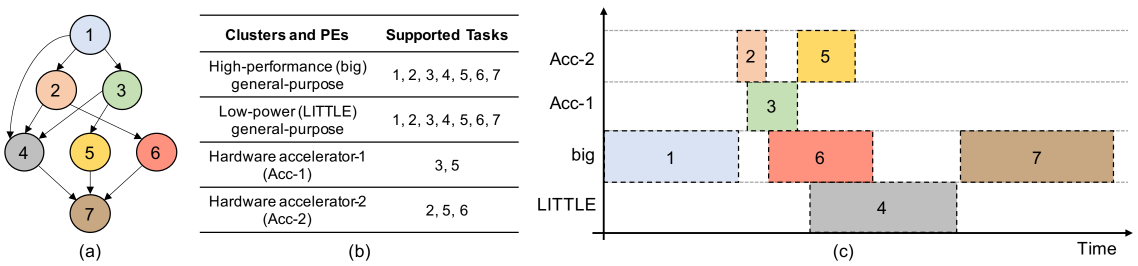
- An imitation learning framework to construct policies for task scheduling in heterogeneous many-core platforms;
- Oracle design using both optimal and heuristic schedulers for performance- and energy- based optimization objectives;
- Extensive experimental evaluation of the proposed IL policies along 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 chapter is organized as follows. Section 4.2 provides an overview of DAG scheduling and imitation learning. In Section 4.3, we discuss the proposed methodology, followed by relevant experimental results in Section 4.4.

4.2 Overview

The runtime scheduling problem addressed in this chapter is illustrated in Figure 4.1. We consider streaming applications that can be modeled using directed acyclic graphs, such as the one shown in Figure 4.1(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 Figure 4.1(a) can start processing



Clusters and PEs	Supported Tasks
High-performance (big) general-purpose	1, 2, 3, 4, 5, 6, 7
Low-power (LITTLE) general-purpose	1, 2, 3, 4, 5, 6, 7
Hardware accelerator-1 (Acc-1)	3, 5
Hardware accelerator-2 (Acc-2)	2, 5, 6

Figure 4.1: (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.

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 Figure 4.1(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 $T_i, T_j \in \mathcal{T}$. ν_{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.

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 Figure 4.1(b) assumes one big core cluster, one LITTLE core cluster, and two hardware accelerators each with a single PE in them for simplicity.

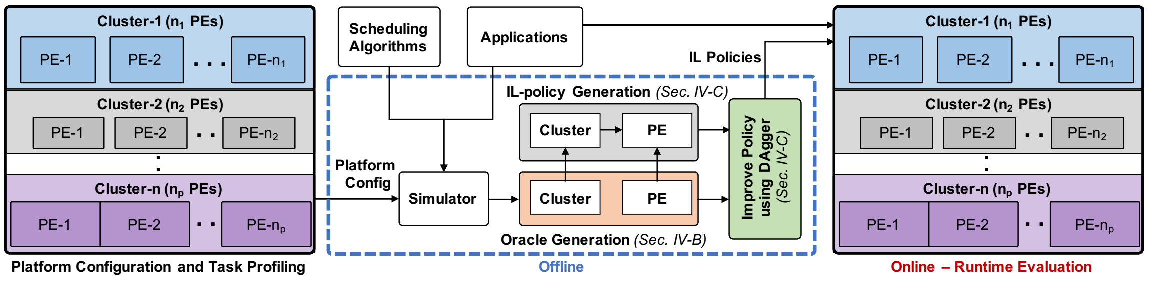

Cluster-1 (n₁ PEs)
PE-1
PE-2
PE-n₁
Cluster-2 (n₂ PEs)
PE-n₂
Cluster-n (nₚ PEs)
PE-nₚ
Platform Configuration and Task Profiling
Scheduling Algorithms
Applications
IL Policies
IL-policy Generation (Sec. IV-C)
Cluster
PE
Platform Config
Simulator
Oracle Generation (Sec. IV-B)
Improve Policy using DAgger (Sec. IV-C)
Offline
Online – Runtime Evaluation

Figure 4.2: 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.

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 Figure 4.1(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 Figure 4.1(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 Figure 4.2.

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 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 Figure 4.2. 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.

4.3 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 4.1 details the notations that will be used hereafter.

4.3.1 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

Table 4.1: Summary of the notations used in the IL-based task scheduling approach

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

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 4.2. 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 4.2 as:

$$s_k = \mathcal{F}_{S,k} \cup \mathcal{F}_{D,k} \tag{4.1}$$

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 4.2: 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

4.3.2 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.

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:

$$\pi^*(s_k) = P_i, \tag{4.2}$$

where $P_i \in \mathcal{P}$ is the PE T_j scheduled to and s_k is the system state defined in Equation 4.1. 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 4.4.1.

4.3.3 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 4.4.4.

Our hierarchical IL-based scheduler policies approximate the Oracle with two levels, as outlined in Algorithm 1. The first level policy $\pi_C(s) : \mathcal{S} \to \mathcal{C}$ 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

Algorithm 1: Hierarchical imitation learning Framework

```
1 for task $T \in \mathcal{T}$ do
2     $s$ = Get current state for task $T$
      /* Level-1 IL policy to assign cluster */
3     $c = \pi_C(s)$
      /* Level-2 IL policy to assign PE */
4     $p = \pi_{P,c}(s)$
      /* Assign $T$ to the predicted PE */
5 end
```

in terms of their dynamic states. The second level (i.e., fine-grained scheduling) consists of *one dedicated policy* $\pi_{P,c}(s) : \mathcal{S} \to \mathcal{P}$ 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 [182]. 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 *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

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

```
1  for task $T \in \mathcal{T}$ do
2      s = Get current state for task T
3      if $\pi_C(s) == \pi^*_C(s)$ then
4          if $\pi_{P,c}(s)$ != $\pi^*_{P,c}(s)$ then
5              Aggregate state s and label $\pi^*_{P,c}(s)$ to the dataset
6          end
7      end
8      else
9          Aggregate state s and label $\pi^*_C(s)$ to the dataset
10         $c^* = \pi^*_C(s)$
11         if $\pi_{P,c^*}(s)$ != $\pi^*_{P,c^*}(s)$ then
12             Aggregate state s and label $\pi^*_{P,c}(s)$ to the dataset
13         end
14     end
       /* Assign T to the predicted PE */
15 end
```

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.

4.4 Experimental Results

Section 4.4.1 presents the experimental methodology and setup. Section 4.4.2 explores different machine learning classifiers for IL. The significance of the proposed features is studied using a decision tree classifier in Section 4.4.3. Section 4.4.4 presents the evaluation of the proposed IL-scheduler. Section 4.4.5 analyzes the generalization capabilities of IL-scheduler. The performance analysis with multiple workloads is presented in Section 4.4.6. We demonstrate the evaluation of the proposed IL technique to energy-based optimization objectives in Section 4.4.7. Section 4.4.8 presents comparisons with RL-based scheduler and Section 4.4.9 analyzes the complexity of the proposed approach.

4.4.1 Experimental Methodology and Setup

***Domain Applications*:** The proposed IL scheduling methodology is evaluated using applications from wireless communication and radar processing domains.

Table 4.3: 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**

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 4.3. We construct workload mixes using these applications and run them in parallel.

***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 Figure 4.3. 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

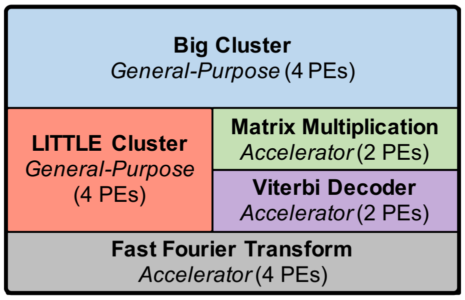


Figure 4.3: Configuration of the heterogeneous many-core platform comprising 16 processing elements, used for scheduler evaluations.

target domain applications summarized in Table 4.3. The accelerator interfaces are adapted from [87]. 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 operations. Therefore, we employ four instances of FFT hardware accelerators and two instances of Viterbi and matrix multiplication accelerators, as shown in Figure 4.3.

***Simulation Framework*:** We evaluate the proposed IL scheduler using the discrete event-based simulation framework [51], *which is validated against two commercial SoCs*: Odroid-XU3 [14] and Zynq Ultrascale+ ZCU102 [162]. 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 4.3.2.

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 [169] 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 (CP_{1-min}) and five minutes (CP_{5-min}) time-out per scheduling decision. When the model fails to find an optimal schedule, we use the best solution found within the time limit. Figure 4.4 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

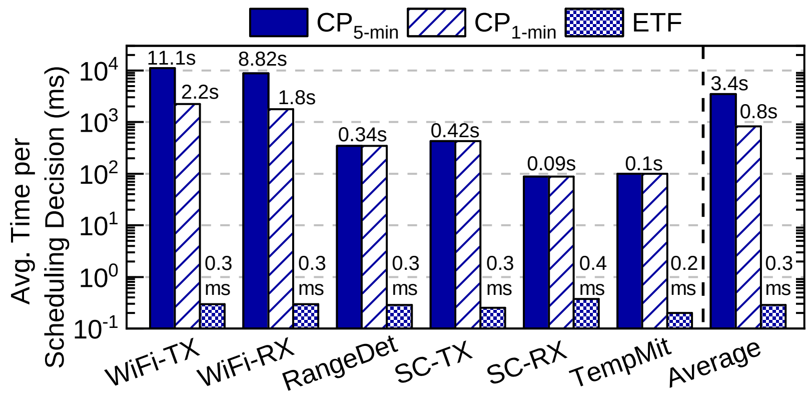


Figure 4.4: A comparison of average runtime per scheduling decision for each application with $\mathrm{CP}_{5-\mathrm{min}}$, $\mathrm{CP}_{1-\mathrm{min}}$ and ETF schedulers.

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 Figure 4.4. 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 4.4.4.

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 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 $\mathrm{CP}_{5-\mathrm{min}}$ 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 4.4.2 and evaluate their performance in Section 4.4.4.

4.4.2 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 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 4.4. 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%)

Table 4.4: 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 4.5: 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

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.

DTs 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 DTs 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 4.5. 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 DTs over NNs, we choose DT for the rest of the experiments.

4.4.3 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 4.6 and Figure 4.5 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

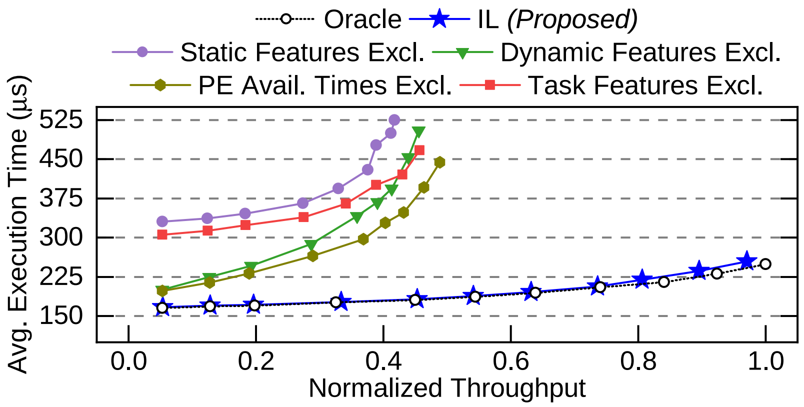


Figure 4.5: Average execution time comparison of the applications with Oracle, IL (*Proposed*) and IL policies with subsets of features. As shown, the average execution time with IL closely follows the Oracle.

Table 4.6: 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

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 Figure 4.5, 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.

4.4.4 IL-Scheduler Performance Evaluation

This section compares the performance of the proposed policy to the ETF Oracle, CP_{1-min}, and CP_{5-min}. Since heterogeneous many-core systems are capable of running multiple applications simultaneously, we stream the frames in our application mix (see Table 4.3) with increasing injection rates. For example, a normalized throughput of 1.0 in Figure 4.6 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 4.3.3. 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.

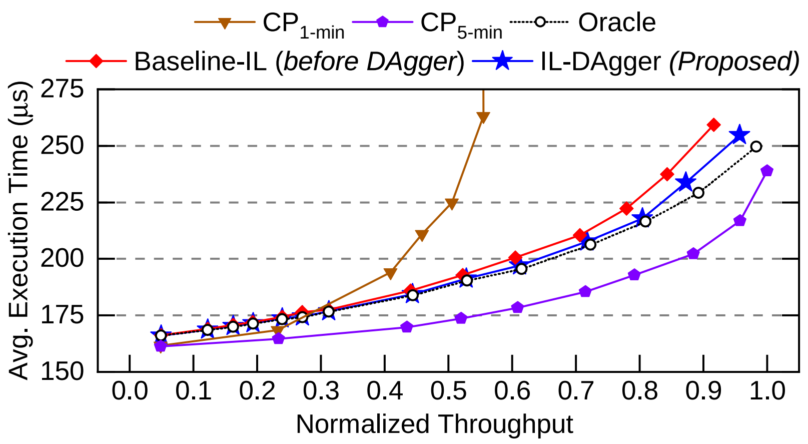


Figure 4.6: Comparison of average job execution time between Oracle, CP solutions, and imitation learning policies to schedule a workload comprising a mix of six streaming applications. IL scheduler policies with baseline-IL (*before DAgger*) and with IL-DAgger (*Proposed*) are shown in the comparison.

Figure 4.6 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 policy and the optimal $\text{CP}_{5-\mathtt{min}}$ solution is only 9.22%. We emphasize that $\text{CP}_{5-\mathtt{min}}$ 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 $\text{CP}_{1-\mathtt{min}}$, 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 4.4.5.

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.

4.4.5 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 [183].

To compare the performances of two schedulers S_1 and S_2, we use the job slow-

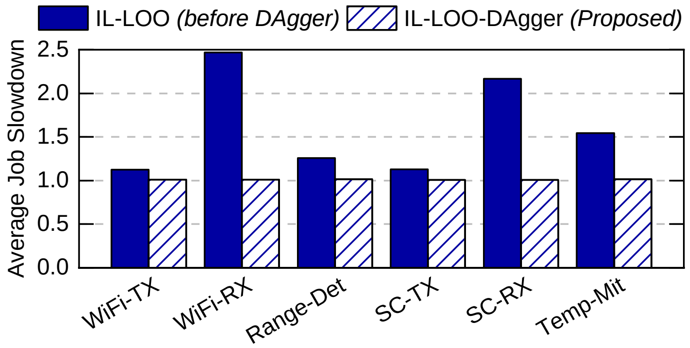


Figure 4.7: Average slowdown of IL policies in comparison with Oracle for leave-one-out (LOO) experiments before and after DAgger (*Proposed*).

down metric $slowdown_{S_1,S_2} = T_{S_1}/T_{S_2}$. $Slowdown_{S_1,S_2} > 1$ when $T_{S_1} > T_{S_2}$ [33]. 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 4.7 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 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 $2\times$ for the receiver applications. The slowdown when the transmitter applications (WiFi-TX and SC-TX) are excluded from training is approximately $1.13\times$. Range detection and temporal mitigation applications experience a slowdown of $1.25\times$ and $1.54\times$, 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 ~$1.01\times$ in comparison with the Oracle, as shown in Figure 4.7.

Figure 4.8(a)-(f) show the average job execution times for the Oracle (ETF),

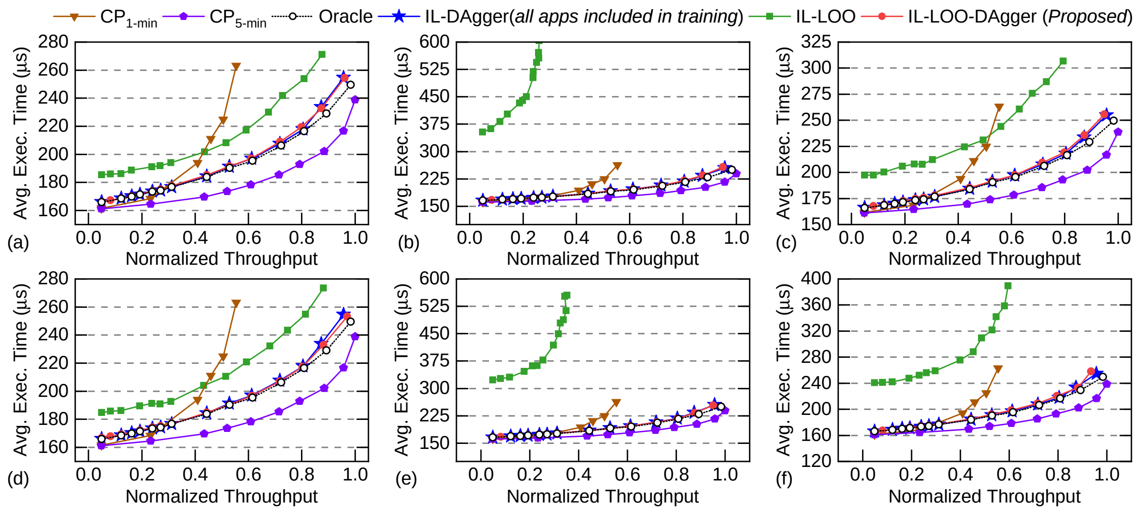


Figure 4.8: Average execution time with Oracle, IL-DAgger (*all applications are included for training*), IL with one application excluded from training (IL-LOO) and finally, the leave-one-out policy improved with DAgger (*Proposed* IL-LOO-DAgger) technique. The *excluded* applications are: (a) WiFi-TX, (b) WiFi-RX, (c) Range Detection (d) Single-Carrier TX (e) Single-Carrier RX and (f) Temporal Mitigation.

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

Table 4.7: 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

static profiles. Our simulator accounts for variations by using a Gaussian distribution to generate variations in execution time [184]. 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 4.7. 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$\times$ 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 4.8. The Oracle policy for a configuration G1 is denoted by π^{*G1}. An IL policy evaluated on configuration

Table 4.8: 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

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 Figure 4.9.

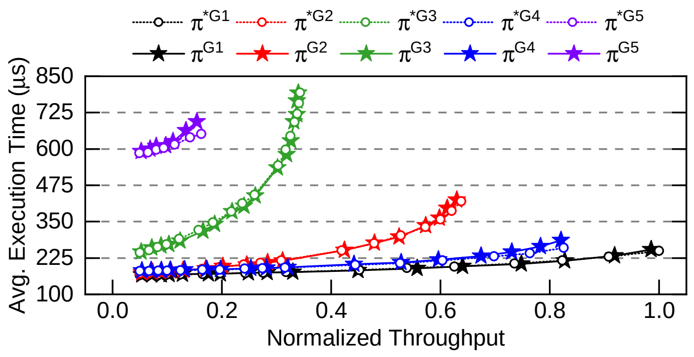


Figure 4.9: IL policy evaluation with multiple many-core platform configurations. IL policies are trained with only configuration G1.

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

4.4.6 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.

Figure 4.10 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 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

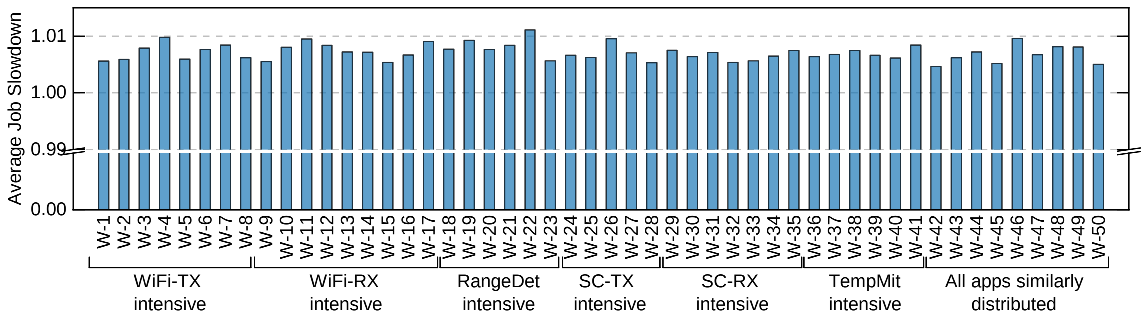


Figure 4.10: Comparison of average job slowdown normalized with IL-DAgger (*Proposed*) policies against the Oracle for *50* different workloads. The slowdown of IL-DAgger policies are shown for workloads with different *intensities* of each application in the benchmark suite.

workloads and streaming intensities.

4.4.7 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 [121, 120]. 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 remains similar for DT of depth 16 (~1.1µs) on Cortex-A53.

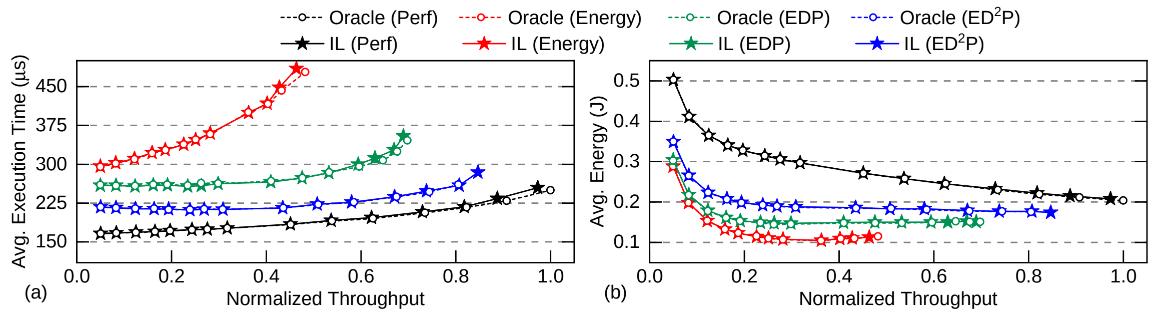


Figure 4.11: (a) Average execution time and (b) average energy consumption of the workload with Oracles and IL policies for performance, energy, energy-delay product (EDP) and energy-delay^2 product (ED^{2}P) objectives.

Figure 4.11(a) and Figure 4.11(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.

4.4.8 Comparison with Reinforcement Learning

Since the state-of-the-art machine learning techniques [33, 109] 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 learn-

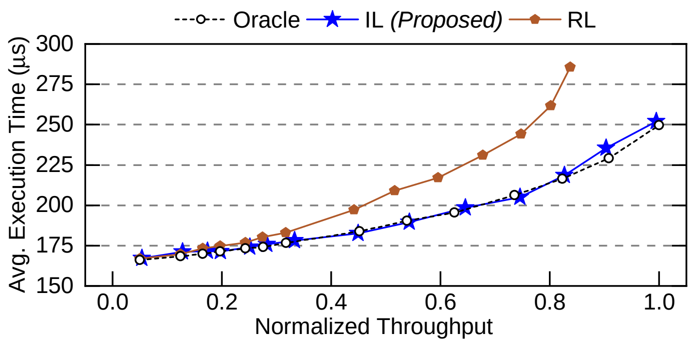


Figure 4.12: Comparison of average execution time between Oracle, IL, and RL policies to schedule a workload comprising a mix of six streaming real-world applications.

ing rate from 0.001 to 0.01. The reward function is adapted from [109]. RL starts with random weights 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 Figure 4.12, 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.

4.4.9 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 $O(n^2m)$ [178], 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.

4.4.10 Evaluation on Real Hardware Platforms

This section presents the demonstration of the intelligent IL-based scheduler for a workload that comprises a mix of GNU Radio applications, WiFi-TX, synthetic aperture radar (SAR) applications on two different platforms, Xilinx Zynq UltraScale+ ZCU102 FPGA and Nvidia Jetson AGX Xavier. Specifically, the pulse Doppler application constitutes the GNU Radio application. We use CEDR, a software runtime framework to execute the IL-based scheduler on real hardware platforms. We first instrument CEDR to run a complex scheduler and generate the Oracle. We choose the earliest finish time (EFT) heuristic for these experiments. During this step, CEDR populates the list of features and labels for each in task the workload. This information is then used to train a DT classifier in Python using the scikit-learn library. The trained model is then plugged back into CEDR to perform runtime scheduling for the workload with streaming application arrivals. The rest of the section focuses on specific details and results on the two evaluation platforms.

4.4.11 Xilinx Zynq UltraScale+ ZCU102

To demonstrate the IL-based scheduling approach on Zynq ZCU102 FPGA, we choose an SoC configuration with 3 CPU cores and 2 FFT hardware accelerators. The EFT scheduler uses only one of the CPUs and both FFT hardware accelerators for scheduling at runtime. As a consequence, the DT-based IL policy also performs likewise, as shown in Figure 4.13(a). We can observe that the hardware accelerators are the obvious PE choice for the FFT dominated workload. The scheduling policy only reverts to the CPU core when both hardware accelerators are busy executing other tasks and have a significant waiting time. Therefore, the IL scheduling policy effectively utilizes the PEs at runtime.

4.4.12 Nvidia Jetson AGX Xavier

On the Nvidia Jetson AGX Xavier platform, we choose a resource configuration of 2 CPU cores and the GPU to execute the workload. The GPU offers substantially lower execution times than the CPU cores and hence, a majority of the workload is executed on the GPU. The scheduling policy prefers to execute the tasks on the

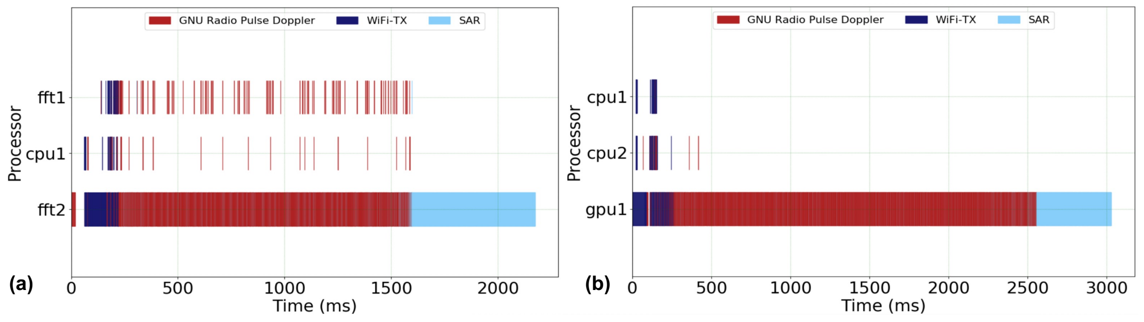


Figure 4.13: Gantt charts of the workload execution on (a) Xilinx Zynq ZCU102 FPGA and (b) Jetson AGX Xavier for a workload comprising the GNU Radio Pulse Doppler, WiFiTX and SAR applications.

CPU cores only when the GPU is significantly blocked for a long period of time, as shown in Figure 4.13(b).

4.4.13 Summary

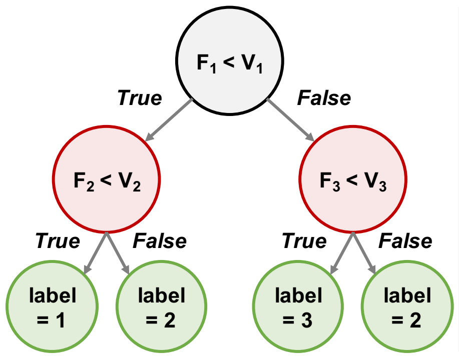
We successfully demonstrate that the IL-based scheduling polices are applicable and portable across different platforms. Furthermore, the framework seamlessly integrated into CEDR, thereby providing an easy plug-and-play environment for user evaluation.

5 OPTIMIZATION TECHNIQUES FOR DECISION TREE CLASSIFIER IMPLEMENTATION

5.1 Background, Motivation and Contributions

Decision trees, traditionally used as machine learning (ML) classifiers in data mining, are gaining traction in resource management algorithms in systems-on-chip [185, 113, 49]. 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 5.1. 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 [185].

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 [185, 113, 49]. 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



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

mining and bioinformatics applications benefit from the parallelism in hardware implementations since they classify large datasets [185]. 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) [113, 49]. Therefore, the nature of the application plays a crucial role in selecting between execution in software and hardware.

This chapter 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 [185]. 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 [186]. 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, which we demonstrate in Section 5.2. 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 software implementation and optimization techniques to achieve ultra-low latency (<50 ns) for trees up to depth 12 for resource management applications, as shown in Section 5.2.

5.2 Hardware Architecture and Evaluation of DT Classifiers

5.2.1 Microarchitecture and Hardware Design

This section first presents the hardware design for a DT classifier and then analyzes its end-to-end latency on an implementation in the Xilinx Zynq ZCU102 FPGA. The microarchitecture of the decision tree hardware design is presented in Figure 5.2. The design utilizes an internal memory structure that holds the features used up in each node in the DT, the threshold values used, and the output class value for the leaf nodes. The internal memory is pre-loaded with the weights based on a trained model. Initially, we always start with the root node whose weights are stored in the first row of the memory. The internal memory provides the feature and threshold value for comparison in the root node. After the comparison, the address of the next node is determined based on the result of the comparison, which determines the address of the next node in the memory. Iteratively, the features and thresholds are fetched for comparison until the leaf node. Once we reach the leaf node, the

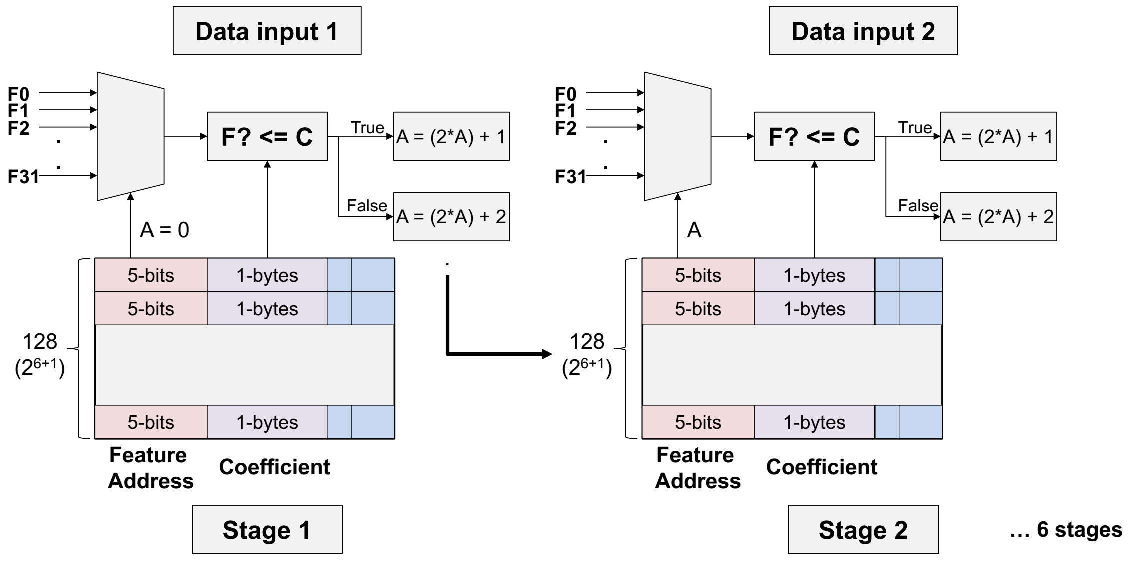


Figure 5.2: Proposed microarchitecture for DT classifier acceleration in hardware.

output class stored in the internal memory is fetched and driven as the output of the classification operation. Currently, the decision provisions for up to 32 input features and is parameterizable for different tree depths. The hardware design processes one node of the tree in 2 clock cycles and uses 1 clock cycle to drive the output. In total, for a DT of depth D, the total number of clock cycles is (2D + 1).

5.2.2 End-to-End Latency Evaluation of DT Accelerator on FPGA

The presented hardware design is translated into behavioral Verilog, synthesized, and implemented on the Xilinx Zynq ZCU102 FPGA. The input features are 8-bits each, and the AXI bus width on the FPGA is 32-bits. The maximum depth of the tree is 4. Hence, we transfer four input features in each write transaction to improve the useful bandwidth and reduce the data transfer latency. The design is implemented at the maximum supported frequency of 300 MHz for the programmable logic,

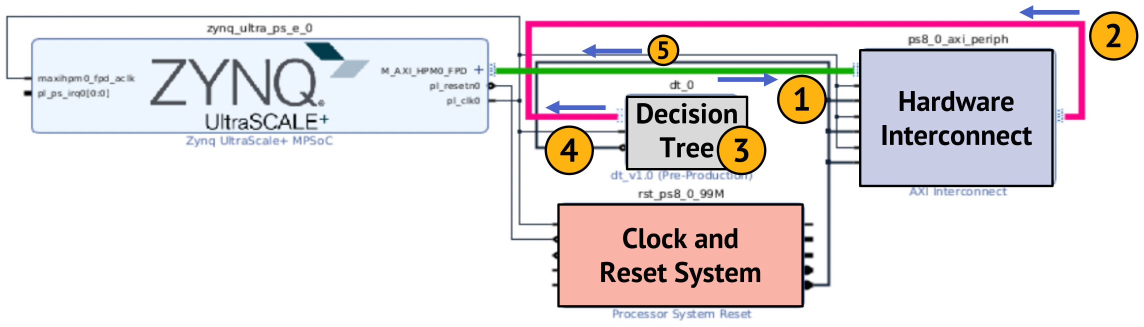


Figure 5.3: Schematic of the hardware components for DT accelerator implementation in ZCU102 FPGA. The numbered circles indicate the computation and communication steps involved in sending and receiving data from/to the accelerator.

while the host controller runs at 1.2 GHz.

Hardware accelerators incur significant overheads in terms of communication as the inputs and outputs are streamed in and out of the design. While this cost is ammortized for compute-intensive designs, it remains substantial for small designs, which we will illustrate in this section. Figure 5.3 presents the schematic of the hardware implementation in the ZCU102 FPGA, the computation and the communication paths to send and receive data to and from the accelerator respectively. We use the Xilinx integrated logic analyzer (ILA) to observe the waveforms and measure the average number of cycles for data transfer as shown in Table 5.1. We observe that the communication latency is 173 ns, which is 6.4$\times$ of the computation latency of 27 ns (2D + 1 cycles for depth 4 at 300 MHz is $\sim$ 27 ns). Therefore, the communication latencies override the benefit of highly efficient computation of DT classification in hardware. Since these latencies are prohibitive, we explore highly targeted software optimizations explained in the next section.

Table 5.1: Table showing the communication latencies of data transfers for the DT accelerator (depth 4) implementation in the ZCU102 FPGA with the Arm cores operating at 1.2 GHz and accelerator operating at 300 MHz.

Communication Step	No. of Transactions	Avg. Latency per Transaction (cycles)	Avg. Latency per Transaction (ns)	Total Expected Latency (ns)
$1 + 2$	9	5	3.33 ns $\times$ 5 = 16.7 ns	16.7 ns $\times$ 9 = 150 ns
$4 + 5$	1	7	3.33 ns $\times$ 7 = 23.3 ns	23.3 ns $\times$ 1 = 23 ns
			Total Communication Latency	173 ns

5.3 Proposed Software Optimization Approaches for DT Classifiers

This section describes the software optimization techniques to enhance the latency of DTs. We use Scikit-learn [4] a popular Python-based framework for supervised machine learning and decision tree classification [4], 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 [187], 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 [188]. 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 5.4. Using 8-bit numbers reduces the size of the model and, subsequently, the memory footprint by 75%. This transformation also allows for processing in

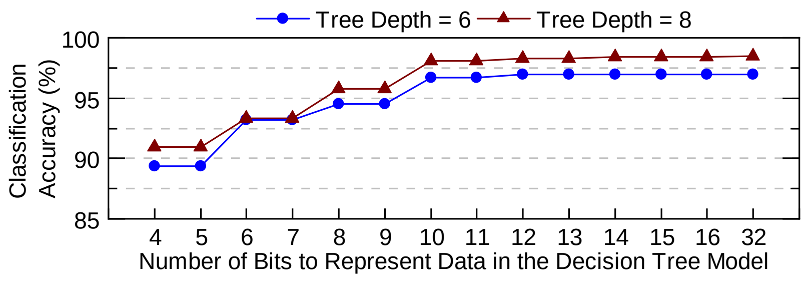


Figure 5.4: Effect of number of bits on the classification accuracy (Fixed-point: 4–16 bits, single-precision floating point: 32 bits).

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 5.1.

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 [189]. 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 simultaneously. The designed decision tree optimizer transforms the **C**-code into a NEON-friendly code, and the implementation is outlined in Figure 5.6.

5.4 Experimental Results

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.

Results and Discussion: We evaluate the proposed DT classifiers using a dataset for dynamic resource management of a system-on-chip (SoC) [49]. 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 5.5 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,

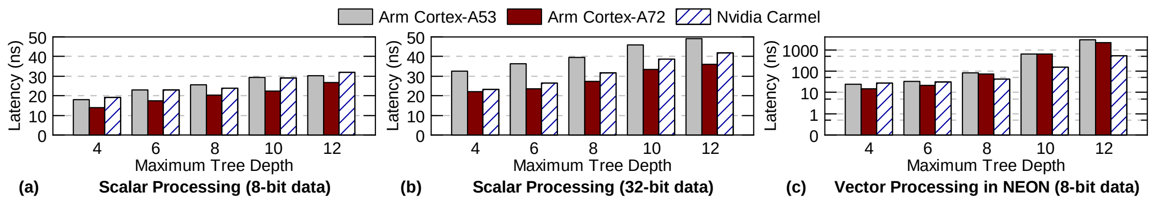


Figure 5.5: Latencies of decision tree classifiers for varying tree depths on Arm Cortex-A53, Arm Cortex-A72 and Nvidia Carmel cores with (a) scalar processing using 8-bit data, (b) scalar processing with floating-point (32-bit) data, and (c) vector processing in NEON using 8-bit data.

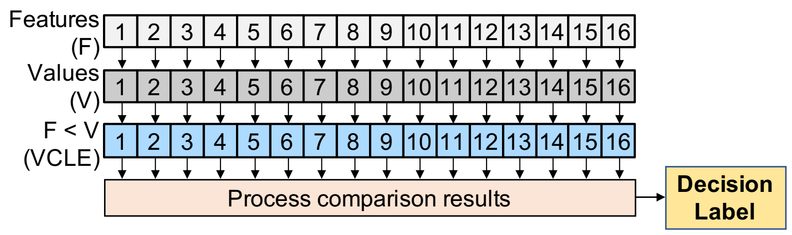


Figure 5.6: Transformation of sequential decision tree code into NEON-friendly code for SIMD processing.

respectively. The latency improves on average by ~35% by converting 32-bit data (Figure 5.5(b)) format to 8-bits (Figure 5.5(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.

6 INCREMENTAL AND ONLINE UPDATES TO DECISION TREE CLASSIFIERS

6.1 Background, Motivation and Contributions

With the slowdown of Moore's law and Dennard scaling, heterogeneous processing elements (PEs) have been the primary catalyst for the performance and energy efficiency of computing systems [5]. For example, highly optimized fixed-function hardware accelerators for signal processing and deep learning are commonly used in communication and autonomous driving applications [49, 23]. However, performance and energy efficiency boosts come at the expense of programming flexibility, as the hardware accelerators are notoriously hard to program. To address this challenge, domain-specific systems-on-chip rise as a new class of heterogeneous SoCs [24, 190]. They combine the flexibility of general-purpose cores with the performance and energy efficiency of specialized hardware accelerators tailored to applications in a target domain [6, 22, 51]. DSSoCs comprise many heterogeneous processing elements, resulting in an ample runtime decision space for task execution. Hence, scheduling algorithms try to identify the most appropriate execution resource to maximize a specific optimization objective, such as performance, power consumption, or energy-delay product [145, 191, 1, 33, 49, 192].

DSSoCs can execute tasks in the order of nanoseconds due to highly specialized hardware accelerators. Hence, task scheduling algorithms must provide high-quality scheduling decisions at ultra-low latencies [6, 147]. Decision tree (in short, DT) classifiers offer a promising solution since they provide high-quality decisions at low inference latency compared to multi-layer perceptron and deep neural networks. Furthermore, DT policies are simple and easy to interpret [52, 148, 185]. Task scheduling policies designed offline are optimized for a particular optimization objective, SoC configuration, and set of applications [193, 1, 49]. Therefore, rapidly evolving SoC architectures, emerging applications, and workloads pose a severe risk to fixed scheduling policies. As these parameters change over time, the offline designed static policies become ineffective, lowering the energy efficiency potentials of DSSoCs. Hence, there is a critical need for task scheduling policies to adapt to dynamic changes to maximize performance and energy efficiency.

Existing DT design techniques require the entire dataset to train a new standalone DT [2]. This requirement poses a significant drawback compared to other ML models, such as neural networks, since storing all training samples would require significant memory on the target platform. Hence, the classical DT training algorithms are impractical for online adaptation. Prior studies tried to address this challenge using reinforcement learning (RL), ensemble trees, and very fast decision trees (e.g., Hoeffding trees) [3, 194, 153, 155]. RL techniques suffer from computational power requirements for training [195]; DT ensembles result in higher latency and computational overheads due to several weak learners [196]; finally, the assumptions to train Hoeffding trees do not hold for online updates [155]. Hence, *existing techniques are not applicable for incremental and online DT updates* due to the resource constraints and inference latency targets for DSSoCs.

This chapter proposes INDENT, a new algorithm for incremental training to adapt DTs for task scheduling to new workloads and hardware changes. INDENT addresses the following research question in the DSSoC task scheduling context: Can we incrementally update a single DT model online *without storing all prior training data and yet obtain the same accuracy* as the DT designed with all training data? The following insights help us answer this question while addressing scalability and shortcomings of prior techniques.

Key Insight 1: We can embed a small amount of metadata (e.g., 1-8% of the original data) into a DT to store the synopsis of the original data used for initial training.

Key Insight 2: We can use the metadata and newly encountered dataset to update the DT online such that it performs similar to one trained from scratch with the entire training data.

The proposed approach is evaluated systematically by varying applications, SoC configurations, and optimization objectives. We use six applications from wireless communication and radar systems domains, three DSSoC configurations, and three optimization metrics (performance, energy, and energy-delay^2 product). Our results demonstrate that INDENT successfully adapts to unseen scenarios using 1%–8% of the training data used by the popular classification and regression tree (CART) algorithm [149]. The resulting DT schedulers achieve almost the same performance (within 5%) as schedulers trained using all the training data. Furthermore, they lead to 5.4×, 2.6×, and 2.5× application execution time speed-up compared to DTs trained by RL, Hoeffding trees, and DT ensembles, respectively. Finally, we implement these DT policies on the Nvidia Jetson Xavier NX platform. INDENT achieves the lowest inference latency of 42 ns, similar to RL and Hoeffding tree algorithms, while the random forest ensemble takes 1.4 μs per decision. INDENT establishes a base to update information theoretic machine learning classifiers online. The key contributions of this chapter are:

- The INDENT algorithm that incrementally trains DTs online by embedding a synopsis of the original training data into the tree;
- Extensive experimental evaluations that demonstrate the benefits of INDENT in adapting to new applications or new processing clusters for different performance and energy metrics;
- Hardware measurements and simulations that show INDENT outperforms RL-, Hoeffding tree- and ensemble tree approaches.

Section 6.2 presents an overview of DT classifiers and our assumptions, while Section 6.3 presents the proposed INDENT algorithm. Section 6.4 presents our extensive experimental evaluations.

6.2 An Overview of Decision Tree Classifiers and Assumptions

6.2.1 Overview of Decision Tree Classifiers

The CART algorithm is widely used to train standalone DT classifiers [149, 197]. Notable DT training algorithms also include iterative dichotomiser 3 (ID3), C4.5, C5.0, Chi-square automatic interaction detection, and MARS [198]. This section

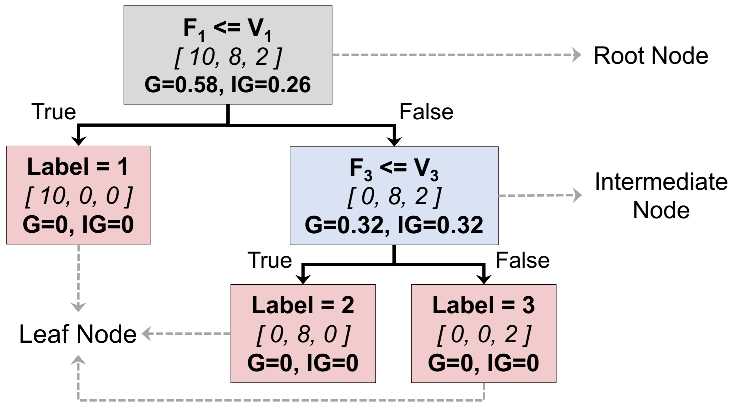


Figure 6.1: An example of a decision tree (DT) classifier showing the root, intermediate, and the leaf nodes.

Table 6.1: A summary of symbols used in INDENT.

Symbol	**Description of the Symbol**
$\mathcal{D}^o, \mathcal{D}^i$	Original training dataset ($\mathcal{D}^o$) and incremental dataset ($\mathcal{D}^i$)
$\mathcal{S}$ $\mathcal{S}^o, \mathcal{S}^i$	Subset of dataset that propagates to the node of interest in the tree Original ($\mathcal{S}^o$) and incremental ($\mathcal{S}^i$) subsets that propagate to tree node
$\mathcal{S}_{true}$ $\mathcal{S}_{false}$	Split of $\mathcal{S}$ using a (feature-threshold) pair into true branch ($\mathcal{S}_{true}$) and false branch ($\mathcal{S}_{false}$) at a tree node
M, N	Number of input features (M) and output classes (N)
$G, G(\mathcal{S})$	Gini index (G) and $G(\mathcal{S})$: Gini index of a sample set $\mathcal{S}$
IG	Information gain
p_i	Probability of classifying sample(s) to class i
P	Set of p_i, where $1 \leqslant i \leqslant N$
F_j	j^{th} feature, $j \in \mathbb{Z}^+, 1 \leqslant j \leqslant M$
$\mathcal{U}(F_j, \mathcal{S})$	Set of unique values of F_j in $\mathcal{S}$

defines the important terms used in the literature and an overview of DTs, while Table 6.1 summarizes the symbols used in this work.

Root Node: The first decision point in the DT is the root node, as shown in Figure 6.1.

Leaf Node: A leaf node in the tree is defined as the node that contains the final prediction (or) output class.

Intermediate Node: The remaining nodes in the tree, apart from the root and leaf nodes. They contain a decision condition and lead to another intermediate node or a leaf node, as shown in Figure 6.1.

Suppose the number of output classes for the DT classifier is N. We denote the probability that any given sample belongs to class i as $p_i \ \forall i \in \mathbb{Z}^+, 1 \leqslant i \leqslant N$. M denotes the number of input features to the classifier.

Gini Index: Assume that a data sample is randomly chosen and classified to one of the N classes. The *Gini index*, a.k.a. Gini impurity, measures the probability that this classification will be *incorrect* [199]. The probability of selecting a sample from class i and the incorrect classification probability are p_i and $(1 - p_i)$, respectively. Hence, the Gini index (G) is expressed as:

$$G = \sum_{i=1}^{N} p_i(1 - p_i) = 1 - \sum_{i=1}^{N} p_i^2 \tag{6.1}$$

As an example, consider a dataset that contains 20 samples categorized into 3 output classes. Assume that 10 samples are labeled with class 1, 8 with class 2, and 2 with class 3.

$$p_1 = 10/20 = 0.5; p_2 = 8/20 = 0.4; p_3 = 2/20 = 0.1$$
$$G = 1 - [(0.5)^2 + (0.4)^2 + (0.1)^2] = 0.58$$

The Gini index ranges between 0 and 1. Suppose all the samples at a particular node in the tree belong to the same class, i.e., $p_i = 1$; then the Gini index is $G = 0$, denoting the purity of the classification. A low Gini index is preferred, especially 0, such that all samples at the particular node can be classified into the same class.

Information Gain (IG): This term, obtained from Kullback–Leibler divergence, is a measure of the information gained when a particular node is split based on a specific feature and a threshold value [149]. Let $\mathcal{U}(F_j, \mathcal{S})$ be the set of *unique values* of *feature* F_j $(1 \leqslant j \leqslant M)$ in sample set $\mathcal{S}$. Suppose the corresponding node is split based on feature F_j and threshold value $V_k \in \mathcal{U}(F_j, \mathcal{S})$. Assume that the samples in $\mathcal{S}$ are split into $\mathcal{S}_{true}$ under the *true* branch and $\mathcal{S}_{false}$ under the *false* branch, hence, $|\mathcal{S}_{true}| + |\mathcal{S}_{false}| = |\mathcal{S}|$. The resulting information gain (IG) is defined as the difference between the Gini impurity of the parent node and the weighted sum of the Gini impurities of the *true* and *false* branches based on (F_j, V_k):

$$IG(F_j, V_k) = G(\mathcal{S}) - [\frac{|\mathcal{S}_{left}|}{|\mathcal{S}|} \cdot G(\mathcal{S}_{true}) + \frac{|\mathcal{S}_{right|}}{|\mathcal{S}|} \cdot G(\mathcal{S}_{false})] \tag{6.2}$$

DT training algorithms aim to choose (F_j, V_k) at each node such that the information gain (IG) is maximized. Figure 6.1 shows an illustration of the Gini indices and information gains at each node for the chosen (F_j, V_k) computed as per Equation 6.1 and Equation 6.2.

6.2.2 Assumptions

We assume that a system-level framework (e.g., a system scheduler or a runtime framework) invokes INDENT when there are variations in the set of applications or processing clusters. Once invoked, the system *temporarily* switches from using the DT for task scheduling to a baseline scheduler, such as a sophisticated heuristic. Sophisticated schedulers include complex heuristics such as the earliest task first [51] and heterogeneous earliest finish time (HEFT) [192]. Given a new task and the system state, the goal of the baseline scheduler is to search for the best scheduling decision. Then, the *system state* and *this decision* are stored as *features* (shown in Table 6.2) and *labels*, respectively. We run the baseline scheduler for a specific amount of time (e.g., 200 new jobs in our experiments) to obtain the incremental dataset ($\mathcal{D}^i$). After the incremental data collection is complete, the proposed INDENT algorithm updates the DT policy, as described in Section 6.3, and switches back to it.

In this work, we use the earliest task first (ETF) [51] heuristic as the baseline since it iterates over all ready tasks and processing elements to determine the best action. However, the INDENT algorithm can work with *any* scheduler since it only uses the scheduling decisions as labels without relying on the internal operation.

Table 6.2: The list of all input features for each task and the specific order of features enforced by INDENT to allow incremental updates when new clusters are added.

List of Input Features	**Order of Features Enforced by INDENT**
Execution times on M clusters	Depth of task in the DAG
Power consumption in M clusters	Job ID
Earliest availability of M clusters	Application type
Depth of task in the DAG	Execution time, power consumption in Cluster-1
Job ID	Earliest availability time of Cluster-1
Application type	Execution time, power consumption in Cluster-2
	Earliest availability time of Cluster-2
	Execution time, power consumption in Cluster-M
	Earliest availability time of Cluster-M

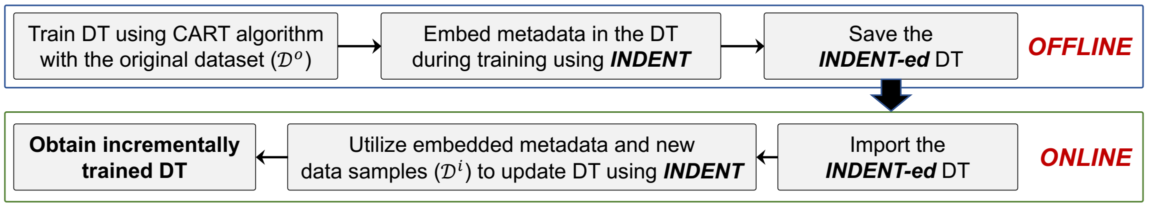

Train DT using CART algorithm with the original dataset ($\mathcal{D}^o$)
Embed metadata in the DT during training using ***INDENT***
Save the ***INDENT-ed*** DT
OFFLINE
Import the ***INDENT-ed*** DT
Utilize embedded metadata and new data samples ($\mathcal{D}^i$) to update DT using ***INDENT***
Obtain incrementally trained DT
ONLINE

Figure 6.2: An overview of the INDENT flow to incrementally train DTs online.

We emphasize that using the baseline scheduler indefinitely is prohibitive since searching for high-quality decisions can lead to significantly long execution times. As our hardware measurements in Section 5.5 demonstrate, DT schedulers trained by INDENT make decisions within 42 ns, in strong contrast to 370 ns – 3.6 µs (for 1–20 ready tasks) execution time of ETF. Finally, we note that the online updates to schedulers will be very infrequent, e.g., new applications or hardware configurations typically appear in the order of months. Consequently, the incremental data collection and runtime overheads of INDENT are negligible.

6.3 INDENT: Incremental Online DT Training

This section presents the proposed INDENT algorithm that incrementally trains DTs online. The algorithm consists of two primary steps as illustrated in Figure 6.2:

1. An *offline phase* that uses the original training data to construct a DT and embed metadata into it,
2. An *online phase* that updates the DT incrementally to adapt to changes in hardware and new applications.

Algorithm 3: INDENT Algorithm to Embed Metadata into the Initial Decision Tree (built OFFLINE)

```
1  build_CART_tree ($\mathcal{D}_o$) /* where $\mathcal{D}_o \leftarrow$ original dataset */
2  Function build_CART_tree ($\mathcal{S}$) begin
3      Features in $\mathcal{S}$, $F \leftarrow \{F_1, F_2, ..., F_M\}$ /* M features */
4      G = compute_Gini ($\mathcal{S}$) as per Equation 6.1
5      best_IG = 0 ; best_feature = 0
6      best_$\mathcal{S}_{true}$ = {} ; best_$\mathcal{S}_{false}$ = {} ; metadata = {}
7      foreach $F_j \in F$ do
8          foreach $V_k \in \mathcal{U}(F_j, \mathcal{S})$ do
9              $\mathcal{S}_{true}, \mathcal{S}_{false}$ = divide_data($\mathcal{S}, F_j, V_k$)
10             IG = G−{[ $|\mathcal{S}_{true}| \cdot G(\mathcal{S}_{true}) + |\mathcal{S}_{true}| \cdot (\mathcal{S}_{false})$ ] / $|\mathcal{S}|$}
               /* Get probability of frequency of classes in true and false
                  branches */
11             $P_{true}$ = get_prob_dist_labels ($\mathcal{S}_{true}$)
12             $P_{false}$ = get_prob_dist_labels ($\mathcal{S}_{false}$)
13             metadata.add ($F_j, V_k$, IG, $|\mathcal{S}_{true}|, |\mathcal{S}_{false}|, P_{true}, P_{false}$)
14             if IG > best_IG then
15                 best_feature = ($F_j, V_k$) ; best_IG = IG
16                 best_$\mathcal{S}_{true}$ = $\mathcal{S}_{true}$
17                 best_$\mathcal{S}_{false}$ = $\mathcal{S}_{false}$
       /* If IG at a node is 0, all samples belong to same class */
18     if best_IG == 0 then
19         metadata = get_histogram ($\mathcal{S}$)
20     P = get_prob_dist_labels ($\mathcal{S}$) ; class_label = argmax (P)
       /* Create a leaf node at stopping criteria */
21     if (best_IG == 0) or (depth == max_depth) then
22         create_DT_node (class_label, metadata)
       /* Build true branch (left child) */
23     true_branch = build_CART_tree (best_$\mathcal{S}_{true}$)
       /* Build false branch (right child) */
24     false_branch = build_CART_tree (best_$\mathcal{S}_{false}$)
25     return create_DT_node(best_feature, true_branch, false_branch, metadata)
```

This section describes these phases and how INDENT adapts to new applications, processing cluster and other unseen scenarios.

6.3.1 The Offline Phase of INDENT

The first step of the offline phase is designing the DT using the recursive CART algorithm with the original dataset. Algorithm 3 outlines the CART training algorithm in *black* typeface, and our modifications to embed metadata into the DT using *blue* typeface. We call the algorithm using the original dataset $\mathcal{D}_o$ (line 1). It first uses the input dataset and features to compute the Gini index at the root node (line 3). Then, it loops through each feature (line 7) and each unique feature value (line 8) to determine the best feature-value pair (F_j, V_k) which maximizes IG at the current node (lines 7–17). The input dataset is split based on the best feature-value pair (F_j, V_k) to divide the samples into the true $(\mathcal{S}_{true})$ and false $(\mathcal{S}_{false})$ branches respectively (line 9). Then, the child's true and false branches are built recursively using the corresponding samples in $\mathcal{S}_{true}$ and $\mathcal{S}_{false}$ (lines 23 & 24), respectively. This recursive training procedure terminates when the maximum depth or leaf node is reached (line 21) on all branches of the DT.

The metadata we embed into the tree is crucial to help reconstruct the information provided by the original dataset. The Gini index and information gain are used to compute the feature and threshold at each decision node (e.g., lines 4 & 10). Since the Gini index depends on the frequency of classes of data arriving at a particular node (Equation 6.1), we store the probability distribution of the labels (lines 11 & 12) as part of the metadata for feature and threshold pair combinations (as shown in line 13 of Algorithm 3). In addition, we also store the information gain and the number of samples in the true and false branch splits. If the information gain at a node is zero, it implies that all samples belong to the same class, and we now store the histogram of the data samples at the node, which will help reconstruct information in the online phase.

6.3.2 The Online Phase of INDENT

The primary challenge in the online training phase is the lack of access to the original training data. However, online algorithms can only use the new training samples and the metadata embedded during the offline phase. Hence, the critical differences between the two phases is the computation of class distribution probabilities (p_i), Gini indices, and IG (lines 4, 10–12, & 20) in Algorithm 3. One of the novel contributions of INDENT is computing these quantities using *only the metadata and the new data samples*, as outlined in Figure 6.3. This key contribution enables the update to the DT without requiring access to the entire original training data.

The online update begins at the root node and follows the steps outlined in Figure 6.3 for all nodes, similar to the offline phase. At the root node, INDENT uses the new training sample set ($\mathcal{S}^i$) and extracts all unique feature-value pairs (F_j, V_k) $\forall j$ for $1 \leqslant j \leqslant M$ and $\forall\ V_k \in \mathcal{U}(F_j, \mathcal{S}^i)$. If F_j is a new feature that has not been

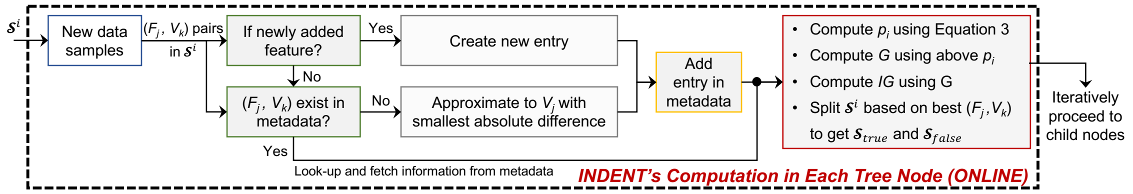

$\mathcal{S}^i$
New data samples
(F_j, V_k) pairs in $\mathcal{S}^i$
If newly added feature?
Yes
No
Create new entry
(F_j, V_k) exist in metadata?
No
Yes
Approximate to V_j with smallest absolute difference
Add entry in metadata
Look-up and fetch information from metadata
Compute p_i using Equation 3
Compute G using above p_i
Compute IG using G
Split $\mathcal{S}^i$ based on best (F_j,V_k) to get $\mathcal{S}_{true}$ and $\mathcal{S}_{false}$
Iteratively proceed to child nodes
INDENT's Computation in Each Tree Node (ONLINE)

Figure 6.3: Block diagram describing the computation in INDENT's online phase to incrementally update a DT node.

encountered before, there is no prior information about F_j in the original training data. Hence, INDENT creates a new entry in the metadata of this node. If F_j is a pre-existing feature, INDENT checks if information to corresponding (F_j, V_k) is available in the metadata. If the information is present, INDENT looks up the information. Otherwise, we approximate the information corresponding to (F_j, V_k) with that of (F_j, V_x) such that $|V_x - V_k|$ is minimum $\forall\, V_x \in \mathcal{U}(F_j, \mathcal{S})$ in the metadata and create a corresponding entry.

The metadata provides the probability distribution of the labels $(p_{i,metadata})$ and number of samples $(n_{i,metadata})$ in the original data for $1 \leqslant i \leqslant N$. Similarly, the corresponding probabilities and number of samples in the new training samples $(\mathcal{S}^i)$ are $p_{i,inc}$ and $n_{i,inc}$ respectively. Using these inputs, INDENT updates the probability distribution of the labels as:

$$p_{i,inc} = \frac{(p_{i,metadata} \times n_{metadata}) + n_{i,inc}}{n_{metadata} + n_{total,inc}} \tag{6.3}$$

This equation enables us to obtain a unified probability for each class. Then, INDENT uses these probabilities and Equation 6.1 to compute the Gini indices. Like in the offline phase, the Gini indices are used to identify the best (F_j, V_k) pair such that IG is maximized. Finally, $\mathcal{S}^i$ is split into the true and false branch samples, $\mathcal{S}_{true}$ and $\mathcal{S}_{false}$ respectively. Then, these two subsets are propagated to the child nodes, and the above computation is recursively repeated until the maximum depth of the tree is reached. As a result, INDENT updates the DT incrementally through this procedure to adapt it to unseen scenarios.

The INDENT framework adapts DT schedulers to a wide variety of scenarios, such as the addition of new processing clusters and the addition of new applications, as discussed in the following sections.

6.3.3 Handling New Processing Clusters

The number of features increases with the number of processing clusters, but the metadata embedded offline does not contain any information about the new features. For example, the additional input features that describe the new PEs include the availability time of a PE, expected execution time of the task on the PE, and power consumption of the task on the PE, as listed in Table 6.2. To address this challenge and update the DT online, we employ the following *three* strategies.
Strategy 1: To retain the significance of existing features and allow scalability for new processing clusters, INDENT enforces an ordering that places the cluster independent features at the beginning (see Table 6.2). Then, the features of each processing cluster are grouped together and appended to the feature set to the end without affecting the significance of the original features in the DT.
Strategy 2: If the feature being parsed is *available* in the original dataset, then INDENT only updates the values of the entries for that (feature, threshold) combination in the metadata.
Strategy 3: If the feature being parsed is a *newly introduced* feature, then we compute the Gini score, information gain, and other metrics, and append them to the metadata.

Combined with these strategies, the techniques presented in Section 6.3.1 and 6.3.2 update DTs online effectively, as illustrated next.

6.3.4 Illustration: Adapting to New Scenarios

This section illustrates how INDENT adapts a DT trained using Algorithm 3 to new scenarios encountered online. Suppose that INDENT is invoked after a new application, which was not known during training, arrives. The new application may have tasks that show different characteristics than the existing ones, while some others may be identical to them. The DT classifier can predict the execution resource accurately for the tasks in the new applications that are identical to the ones in the original training data. However, it may need one of the following two updates for the tasks that *do not* resemble existing tasks:

1. Adding new nodes or branches,
2. Updates to the decision criteria in the existing nodes.

We explain both scenarios using a canonical example shown in Figure 6.4, which assumes that the dataset uses *two* features and contains *three* output classes. The

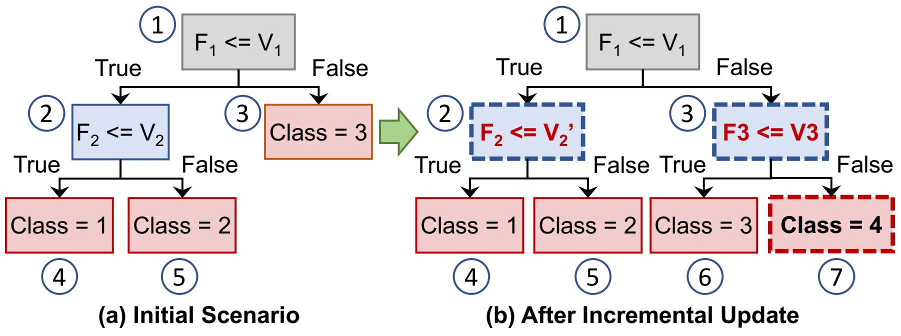


Figure 6.4: A canonical example showing the (a) initial and (b) incrementally updated DT. The incremental update may change decision criteria in nodes (node 2), expand nodes (node 3) and introduce new leaf nodes (node 7).

DT for the applications in the initial workload is shown in Figure 6.4(a). The introduction of new applications in the workload expands the DT by creating new nodes, highlighted by rectangles with *red bold dashed* borders in Figure 6.4(b). Another example of an unseen scenario is newer processing clusters, which typically improve the power and performance characteristics. DTs may require updating the decision criteria in the existing nodes or adding new ones.

INDENT effectively updates the decision criteria in the nodes (rectangles in *blue bold dashed* borders) and modifies the structure of the tree, as illustrated in Figure 6.4(b). The quality of its results is evaluated in the next section.

6.4 Experimental Evaluation

This section first presents the applications and the DSSoC configurations. Then, Section 6.4.2 evaluates the proposed INDENT algorithm with the addition of new processing clusters, while Section 6.4.3 presents the results when new applications are introduced. Section 6.4.4 illustrates INDENT with three different optimization objectives. Finally, Section 6.4.5 compares INDENT to state-of-the-art RL, Hoeffding trees, and DT ensemble (random forest) techniques.

6.4.1 Experimental Setup

Domain Applications: We use the following six wireless communications and radar systems applications to evaluate INDENT: WiFi transmitter and receiver (W-TX and W-RX), single-carrier transmitter and receiver (S-TX and S-RX), lag detection (LD), and temporal mitigation (TM). Jobs from these applications are mixed to construct complex workloads (see Table 6.3). Since the jobs stream into the system, they observe varying system dynamics in the states of the system and processing elements in the DSSoC.

DSSoC Description: The processing clusters are chosen to provide programming flexibility and maximize the energy efficiency for domain applications. To this end, the DSSoC has two general-purpose core clusters. The first one has four energy-efficient LITTLE Arm cores, while the other has four high-performance (energy-intensive) big Arm cores. In addition, the DSSoC integrates hardware accelerators for complex matrix multiplication (MM), fast Fourier transform (FFT), inverse FFT, and Viterbi decoding (VD), as shown in Table 6.3. We use 4 cores each in the LITTLE, big and FFT clusters, and 2 cores each in the FFT and VD cluster, resulting in a total of 16 cores.

Evaluation Hardware Platform and Simulator: The proposed INDENT framework is rigorously evaluated with the DS3 simulator [51]. DS3 supports plug-and-play scheduling algorithms, analytical and logical models for the scheduling environment, processing elements, interconnect, and memory. It is calibrated and validated with Xilinx Zynq ZCU102 and Exynos 5422 platforms. The DTs used in our evaluations are *implemented* on the Tegra SoC in the Jetson Xavier NX platform and profiled for their inference latencies.

Decision Tree Specification: The maximum DT depth and the number of input features critically influence the inference latency, which must be low for resource management applications. Therefore, we choose a small set of 19 features and a

Table 6.3: Table summarizing the workload and SoC configuration scenarios used for the evaluation of INDENT. The bold text denotes the new processing clusters or applications.

SoC Config.	**Clusters in Initial Config**	**Clusters in New Config**	**Workload**	**Apps in Initial Workload**	**Apps in New Workload**
Config-1	LITTLE, big MM, VD	LITTLE, big MM, VD, ***FFT***	**Workload-1**	W-TX, S-TX, LD, TM	W-TX, S-TX, LD, TM, ***W-RX,S-RX***
Config-2	LITTLE, big FFT, MM	LITTLE, big FFT, MM, ***VD***	**Workload-2**	W-RX, W-TX, S-RX, S-TX	W-RX, W-TX, S-RX, S-TX, ***LD,TM***
Config-3	LITTLE, big FFT, VD	LITTLE, big FFT, VD, ***MM***	**Workload-3**	W-RX, W-TX, LD, TM	W-RX, W-TX, LD, TM, ***S-TX,S-RX***

maximum depth of 6 to avoid overfitting. The depth of 6 is enforced for INDENT and all other prior approaches that we use for comparison in this work.

In our experiments, the DTs are *first* trained with the original data $\mathcal{D}^o$ offline using Algorithm 3. *Then*, new processing clusters are added to the SoC or the workload is expanded with the new applications. *Finally*, INDENT incrementally updates the offline DT to adapt it to the unseen scenarios. INDENT's performance is compared to the baseline approach that trains a new DT from scratch using both the original ($\mathcal{D}^o$) and incremental ($\mathcal{D}^i$) data. We strongly emphasize that *this baseline is not practical, but it is chosen here to produce a theoretical comparison point.*

6.4.2 Evaluating INDENT with New PE Clusters

To evaluate the INDENT framework with new processing clusters, we consider the three SoC configurations presented in Table 6.3.

Adding an FFT accelerator cluster: Wireless communication and radar applications

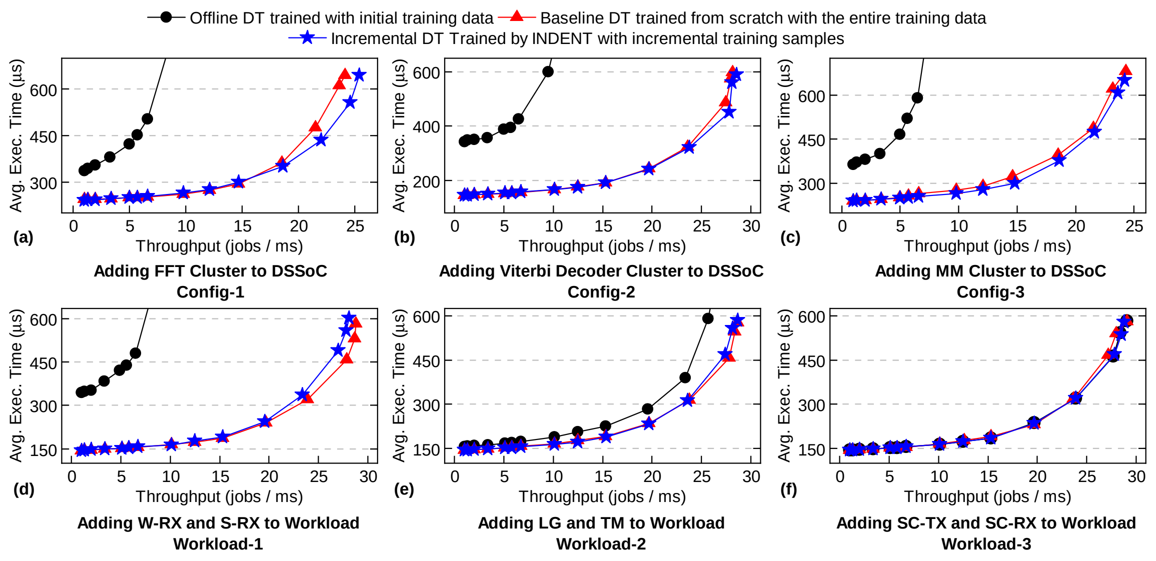


Figure 6.5: The incremental DT trained using INDENT (⋆ marker) by updating the initial DT (● marker) performs very close to the DT trained from scratch with the entire training data (▲ marker) .

extensively utilize the FFT and inverse-FFT operations [200]. In the absence of an FFT cluster, the initial DT policy schedules the FFT tasks to the general-purpose cores, leading to significantly higher execution times, as shown in Figure 6.5(a). This study adds an FFT hardware accelerator that performs the direct and inverse-FFT operations at least 10× faster than general-purpose cores [49]. Then, INDENT adapts the DT online with the new training samples to learn using the FFT processing cluster. Figure 6.5(a) shows that INDENT (⋆ marker) achieves 2% better performance than the baseline tree trained from scratch (▲ marker) using only 7.4% of the original data as metadata. Furthermore, storing only the entries with the highest information gains reduces the metadata size to 6.6% with negligible performance impact.

Adding a Viterbi decoder accelerator cluster: The W-RX and S-RX receiver applications use the Viterbi decoding (VD) task to decode the incoming bitstream. The original DT was trained without this accelerator and hence, schedules the Viterbi decoding task to the general-purpose cluster with significant execution time (~2 ms). As a result, the overall system performance is significantly poor, as shown in Figure 6.5(b). In contrast, INDENT successfully adapts the DT and takes advantage of the new Viterbi decoder accelerator (with ~10 µs execution time). INDENT improves the original DT significantly using metadata as little as 1.1% of the original training data. Figure 6.5(b) shows that its performance is almost the same as the baseline DT.

Adding a Matrix Multiplication accelerator cluster: The lower task latency with the matrix multiplication accelerator significantly improves the average execution time per job. After adding the matrix multiplication accelerator, updating the DT incrementally using as little as 1.7% of the original training data achieves a performance within 3% of a DT trained from scratch, as shown in Figure 6.5(c). These results demonstrate that INDENT performs as well as training DTs from scratch after adding new processing clusters.

6.4.3 Evaluating INDENT with New Applications

To demonstrate INDENT's ability to update a DT classifier with new applications (described in Section 6.3.4), we consider the three workload scenarios presented in Table 6.3 (columns 4–6). Workload-1 initially comprises only the transmitter (W-TX and S-TX) and radar (LD and TM) applications. Then, we introduce the WiFi and single-carrier receiver applications (W-RX and S-RX). Likewise, the second scenario introduces the radar (LD and TM) applications to the initial workload. Finally, the third scenario adds two single-carrier (S-TX and S-RX) applications into the workload after offline training.

Workload Scenario-1: Adding W-RX and S-RX receiver applications into the workload: The newly added applications have a compute-intensive VD task, not included in the original dataset. Since the original DT does not know how to handle these tasks, it schedules them to the general-purpose cluster which is almost 100$\times$ slower in execution compared to the Viterbi decoder accelerator cluster, as described in Section 6.4.2. Hence, the overall system performance degrades significantly, as shown in Figure 6.5(d). In contrast, INDENT ($\star$ marker) successfully adapts the offline trained DT ($\bullet$ marker) to these new applications (as conceptually illustrated in Figure 6.4) and exploits the Viterbi decoder accelerator. As a result, the DT incrementally trained using INDENT performs within 3.7% of the DT trained from scratch ($\blacktriangle$ marker) and metadata that is 4% of the original training data, as shown in Figure 6.5(d).

Workload Scenario-2: Adding two radar applications (LD and TM) into the workload: Introducing these two applications into the workload causes performance degradation of 16% on average (and 54% at the highest throughput) when the initial DT policy is used. Similar to the previous scenario, INDENT adapts the offline DT to learn the new radar applications in the workload, as shown in Figure 6.5(e) and performs within 2% of the baseline DT by utilizing metadata that sizes to 4.7% of the original data.

Workload Scenario-3: Adding single-carrier applications into the workload:

These two new applications do not contain any new tasks that are missing in the original dataset. Thus, the initial DT performs as well as the fully trained tree with the addition of S-TX and S-RX applications. Figure 6.5(f) shows that INDENT successfully maintains the accuracy without any adverse effects. This corner case confirms that INDENT will improve the DT when there is room for improvement without impacting correct decisions.

6.4.4 Illustration with Other Objectives

This section illustrates INDENT with schedulers that optimize other objectives, such as energy and energy delay^2 product (ED^{2}P). For demonstration, we consider a scenario when W-RX and S-RX applications are added to the workload. Figure 6.6(a) shows that the original DT (● marker) performs poorly after the new applications are added. In contrast, INDENT successfully adapts the DTs to the new applications

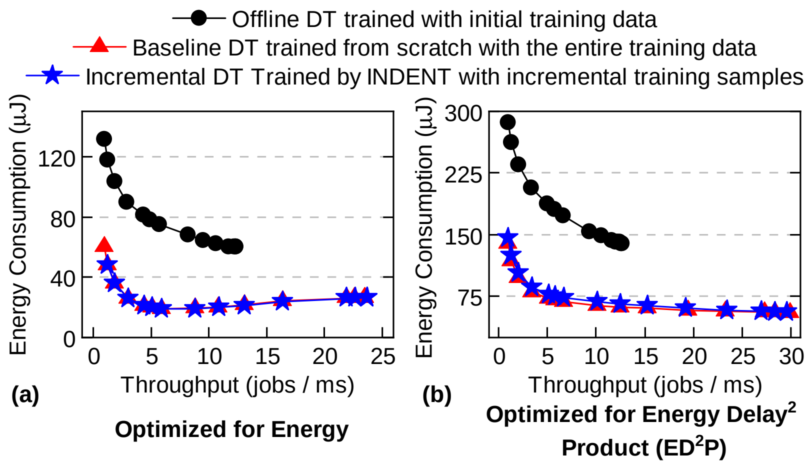


Figure 6.6: Energy comparison of INDENT DTs optimized for energy (a) and ED^{2}P (b) for workload-1 against a baseline DT.

with energy consumption within 2% of the baseline DT using only 5% of the original training samples as metadata. Similarly, Figure 6.6(b) shows that INDENT works equally well with the ED^{2}P metric (emphasizes performance more than delay), achieving almost identical performance to the DT trained from scratch with entire training data.

6.4.5 Comparing INDENT with Prior Work

This section compares the performance of INDENT to alternative approaches using RL, Hoeffding trees, and DT ensembles.

Reinforcement Learning: The RL approach used for comparison is based on a differentiable decision tree (DDT) approach proposed in [3]. We use the proximal policy optimization (PPO) based RL method for evaluation [201]. The negative value of the makespan of each job is used as the reward to update the DT using RL [33].

Hoeffding Trees: We employ the Hoeffding tree classifier implementation from the River library [202, 2], with a maximum depth of 6. Other parameters such as splitting confidence and tie threshold are retained at the default recommended values.

Decision Tree Ensembles: Random forests (RF) are highly popular DT ensemble techniques, and hence, we use the scikit-learn library implementation to compare against INDENT [4]. Our implementation uses 100 weak estimators with a depth of 6.

Performance Comparison: Figure 6.7(a) presents a comparison of the loss in the

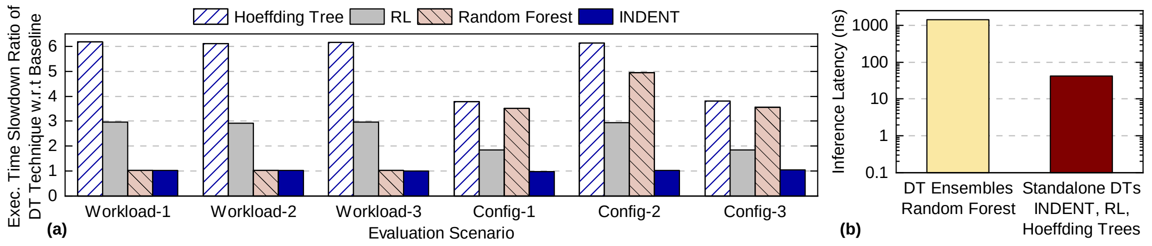


Figure 6.7: (a) Comparison of average execution time slowdown ratio of Hoeffding Tree [2], RL [3], Random Forest (DT Ensemble) [4] and INDENT (our proposed approach) with respect to the baseline DT trained from scratch with the entire training data. (b) Comparison of inference latencies measured on the Jetson Xavier NX hardware platform of DT ensemble (random forest [4]) and standalone DTs (such as INDENT, RL [3] and Hoeffding trees [2]).

performance of the RL-based approach, Hoeffding trees, random forest, and proposed INDENT techniques normalized to the execution time of the baseline DT. RL suffers from poor convergence due to a weak reward function and a small number of features, thereby limiting its ability to learn the scheduling decisions effectively. Furthermore, RL consumes excessively large computation resources (multi-threaded CPUs and GPU acceleration), takes more than 4 hours to train a policy (versus 20 minutes for INDENT and other prior techniques), and encounters severe convergence issues, making it impractical. The performance degradation with Hoeffding trees is substantial since they are suited for ultra-large data streams and require large tree depths to grow the tree with incoming data samples. A maximum depth of 6 is highly limiting for Hoeffding trees, resulting in poor performance. Random forests perform similar to INDENT when new applications are introduced, but fail to learn the decisions when new processing clusters are introduced since new features are added. Hence, they are unable to adapt to these changes. In contrast, INDENT performs well in all cases and achieves similar performance (within 5%)

to the baseline DT. Finally, INDENT achieves an average application execution time speedup of 5.4×, 2.6×, and 2.5× compared to RL, Hoeffding trees, and DT ensembles, respectively.

Inference Latency: We implemented the DT classifiers on the Nvidia Jetson Xavier NX platform running at 1.9 GHz. The inference latency measured on the platform is 42 ns for the standalone classifiers such as INDENT, RL, and Hoeffding trees. On the contrary, the random forest ensemble classifier experiences a higher inference latency of 1.4 µs (due to the evaluation of 100 weak learners and combining their predictions), making them a less suitable choice for resource management applications, as shown in Figure 6.7(b).

Summary: Out of the approaches we evaluated (RL, Hoeffding Trees, DT ensembles and INDENT), *INDENT is the only approach* that achieves ultra-low latency, outperforms other approaches, and also performs very similar (within 5%) to hypothetical DTs that are trained with both the original and incremental training samples.

7 FPGA-BASED EMULATION FRAMEWORK FOR DOMAIN-SPECIFIC ARCHITECTURES

7.1 Background, Motivation and Contributions

With the slowdown of Moore's Law, the ability of traditional homogeneous processors and single-ISA heterogeneous multicore architectures to satisfy the power and performance requirements has saturated [21]. Graphics processing units (GPUs), digital signal processors (DSPs), and hardware accelerators significantly improve the efficiency metrics at the cost of user programmability. Domain-specific system-on-chip (DSSoC) architectures, which are a specific realization of heterogeneous architectures, bridge the gap between programmability and energy efficiency by smartly combining general-purpose, special-purpose, and hardware accelerator cores. The special-purpose and hardware accelerator cores strive to maximize the energy efficiency of applications in a targeted domain, and the general-purpose processors provide programming flexibility [87].

SoC architectures, particularly DSSoCs, face monumental design and verification efforts due to rapidly increasing design sizes and complexities and pose critical threats to the design and verification life cycle, planning, cost, man effort, tools, and time to market [203]. Functional and performance bugs in these complex chips post-fabrication result in unprecedented costs. Therefore, stringent pre-silicon verification techniques such as RTL simulation, gate-level simulation, formal verification, FPGA emulation, and prototyping frameworks are used to detect and rectify bugs in the early design stages. Specifically, FPGA prototyping offers the following advantages [86]: (1) enables execution of real-world workloads on the full system (significantly faster than simulation), (2) allows early firmware and software development, and (3) facilitates faster time-to-market. In literature, FPGAs have extensively been used for network-on-chip (NoC) emulation, prototyping, and performance evaluation of novel special-purpose architectures [204, 205, 20, 206]. However, end-to-end frameworks do not exist for the emulation and prototyping DSSoCs on a Linux-based operating system.

This chapter proposes FALCON, an end-to-end FPGA-based emulation framework, to prototype DSSoCs for rapid design, pre-silicon functional validation, and performance evaluation. FALCON provides an accelerator sandbox, which uses standard AMBA-based interfaces to the rest of the SoC. The accelerator sandbox improves developers' productivity by providing a plug-and-play environment to include, remove, and modify hardware accelerators. FALCON also allows designers to develop drivers for non-standard ISA designs before the chip is available. The framework enables software and firmware development, including boot firmware, operating system bundles, and device management. Finally, FALCON interfaces with CEDR [87], a software runtime framework, to allow applications to be seamlessly executed in a DSSoC. *The contributions in this chapter are summarized as follows:*

- An FPGA-based DSSoC emulation framework,
- An accelerator sandbox that provides a plug-and-play interface for hardware accelerators,

- An environment for hardware accelerator driver development and validation, and
- Experiments to demonstrate FALCON's capabilities using radar/signal processing domain applications.

7.2 The FALCON Architecture

This section describes FALCON's full-system architecture for DSSoC design and emulation, as outlined in Figure 7.1. FALCON is composed of the hardware platform and the software stack. While these components are typical in any SoC, DSSoCs are highly customized to maximize the energy efficiency of domain applications. The hardware platform integrates general-purpose cores that offer programma-

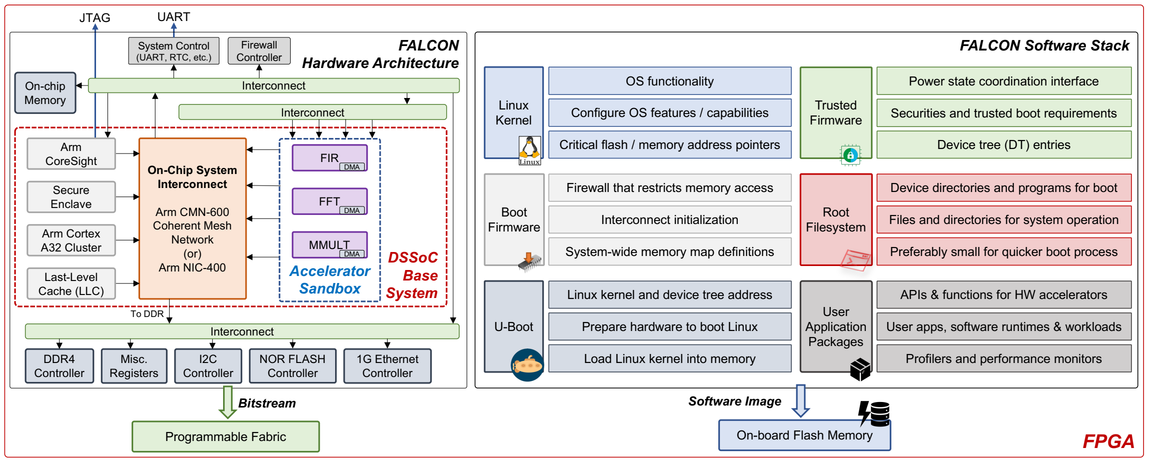

JTAG
UART
System Control (UART, RTC, etc.)
Firewall Controller
FALCON Hardware Architecture
On-chip Memory
Interconnect
Interconnect
Arm CoreSight
Secure Enclave
Arm Cortex A32 Cluster
Last-Level Cache (LLC)
On-Chip System Interconnect
Arm CMN-600 Coherent Mesh Network (or) Arm NIC-400
FIR DMA
FFT DMA
MMULT DMA
Accelerator Sandbox
DSSoC Base System
To DDR
Interconnect
DDR4 Controller
Misc. Registers
I2C Controller
NOR FLASH Controller
1G Ethernet Controller
Bitstream
Programmable Fabric
FALCON Software Stack
Linux Kernel
OS functionality
Configure OS features / capabilities
Critical flash / memory address pointers
Trusted Firmware
Power state coordination interface
Securities and trusted boot requirements
Device tree (DT) entries
Boot Firmware
Firewall that restricts memory access
Interconnect initialization
System-wide memory map definitions
Root Filesystem
Device directories and programs for boot
Files and directories for system operation
Preferably small for quicker boot process
U-Boot
Linux kernel and device tree address
Prepare hardware to boot Linux
Load Linux kernel into memory
User Application Packages
APIs & functions for HW accelerators
User apps, software runtimes & workloads
Profilers and performance monitors
Software Image
On-board Flash Memory
FPGA

Figure 7.1: An overview of the key components and organization of the FALCON framework for DSSoC emulation on FPGAs. The hardware architecture and design (shown on the left) is programmed as a bitstream onto the programmable fabric. The software image (shown on the right) is programmed to the on-board flash memory.

bility, hardware accelerators and specialized processors for energy efficiency, a high-speed interconnect for low-latency on-chip data movement, last-level cache (LLC), peripherals, and debug logic. After synthesis and automatic place-and-route, the entire hardware architecture is packaged into a bitstream to program the programmable logic on the FPGA. The software stack comprises the Linux OS kernel, file system, and embedded system software components such as the boot firmware and U-boot. All components in the software stack are integrated into a software image, which is programmed into the FPGA flash memory. Then, applications run on the underlying hardware of the DSSoC with the use of software runtime environments, such as CEDR and SPARTA [87, 124].

7.2.1 Hardware Architecture

The FALCON hardware architecture is constructed using three major components: (1) DSSoC base system, (2) DSSoC accelerator sandbox, and (3) miscellaneous hardware, controllers, and peripherals in the FPGA. The framework organizes the energy-efficient processors into the accelerator sandbox and the general-purpose processors with the on-chip system-level interconnect into the base system. In addition to the base system and the sandbox, FALCON includes peripherals, controllers, and other hardware, as shown in Figure 7.1.

7.3 Demonstrations using FALCON

FALCON aids chip developers in performing functional design validation, identifying the optimal data flow for hardware accelerators, analyzing performance bottlenecks using hardware performance counters, and even performing early pre-silicon power evaluations. This chapter demonstrates the capabilities of FALCON for the radar/signal processing application domain. We pick two representative examples: pulse Doppler and temporal mitigation, to showcase the need for hardware acceleration using fast Fourier Transform (FFT) and matrix multiplications. FALCON is evaluated using three Xilinx FPGA devices, Zynq UltraScale+ ZCU102, Virtex UltraScale+ VCU128, and Virtex UltraScale+ VU19P. Only the accelerator sandbox is deployed on the ZCU102 since it includes pre-built Arm Cortex-A53 cores (Zynq base system). We strongly emphasize that FALCON is generic and can be seamlessly used for any application domain using the hardware architecture and software stack principles described in this work.

7.3.1 Enabling Software Development and Functional Validation

We developed software drivers for the hardware accelerators in the sandbox, which send and receive data from the system using DMA units. We generate random stimuli as inputs to the hardware accelerators. The outputs of the accelerators are compared with a reference software implementation for functional validation and precision evaluation. For the FFT accelerator, we evaluated transform sizes from 32 to 2048 (in multiples of 2) as shown in Figure 7.2. The number of precision mis-

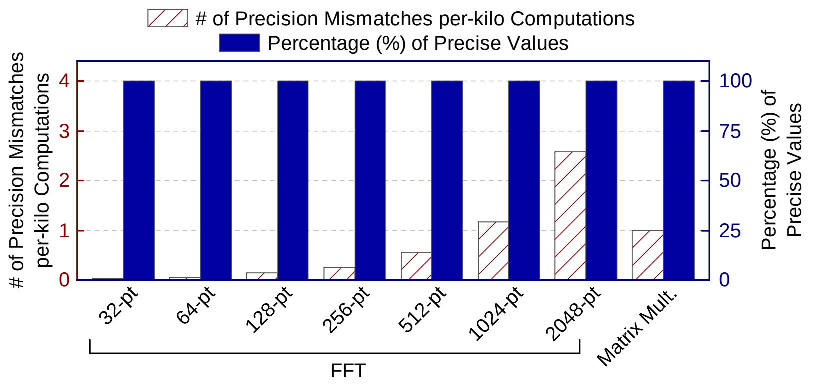


Figure 7.2: The number of precision mismatches (primary axis) per-kilo computations and the percentage of precise values (secondary axis) of the FFT and matrix multiplication hardware accelerators with respect to a reference software implementation in VCU128.

matches remains fewer than three per-kilo computations. A larger transform size experiences higher mismatches due to a higher number of floating-point multiplication and accumulation operations. The fixed-size complex matrix multiplication accelerator experiences an average of one precision mismatch per-kilo computations. On average, more than 99.9% of the values in the computations match the software implementation within a 0.1% difference. Therefore, FALCON enables the development of software drivers, functional validation of the drivers and accelerators when exercised with the full system, and evaluation of the precision requirements.

7.3.2 Optimizing DSSoC Configuration for Domain Applications

Hardware accelerators introduce the notorious double-copy problem where the data must be explicitly transferred from/to them using DMA units. While fetching the data from the main memory involves significant latency overheads, the

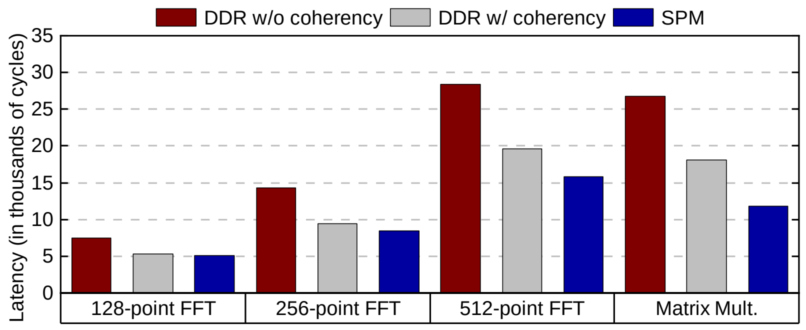


Figure 7.3: A comparison of the hardware accelerator latencies (in thousands of cycles) when data is transferred through DDR memory (with and without coherency) and the on-chip scratchpad memory (SPM) in VCU128.

on-chip scratchpad memory have limited capacity. Figure 7.3 presents the latency (in thousands of equivalent CPU clock cycles) for 128-, 256-, and 512-point FFT operations and complex matrix multiplication computation. The latency improves substantially with the use of scratchpad memory for larger computations (larger transform sizes). Larger computations require more memory and experience significant conflicts in the cache. Therefore, they experience better speedup when data is transferred using the on-chip scratchpad. Through such analyses, FALCON allows us to optimize the number of processors, cache sizes, and memory hierarchies for the processing elements for specific domain applications to maximize energy efficiency.

7.3.3 Illustration of Hardware Performance Counters

Table 7.1 presents the hardware counters (described in Section 7.4.2) for two implementations of a matrix multiplication operation. The two implementations use different loop ordering to improve data locality in the caches, reflecting fewer cache

Table 7.1: Comparison of Hardware Performance Counter Values between Two Different Software Implementations of Matrix Multiplication in VCU128 operating at a frequency of 32 MHz.

HW Perf. Counter	Implementation 1	Implementation 2
Task-clock	640.2 seconds	495.6 seconds
Context-switches	687	164
Cpu-migrations	0	1
Page faults	751	996
Cycles	20.5 B	15.8 B
Instructions	14.8 B	9.5 B
Branches	1.0 B	0.14 B
Branch-misses	5.9 M (0.58%)	2.2 M (1.59%)
Cache-references	1.5 B	1.9 B
Cache-misses	125 M (8.6%)	5.3 M (0.274%)

misses and, consequently, fewer computation cycles and wall time (Table 7.1). The ability to utilize performance counters in FALCON enables users and developers to systematically analyze the effects of code optimization and their impact in terms of microarchitectural events.

7.3.4 Enabling Pre-Silicon Power Evaluations

Figure 7.4 presents the power consumption in the processing system (PS) and programmable logic (PL) in a ZCU102 FPGA using four Arm Cortex-A53 cores at 1.2GHz, one FFT, and one complex matrix multiplication accelerator operating at 100 MHz. The power consumption is captured using TI INA226 power monitors on the board. We run 100 jobs each of *pulse Doppler* and *temporal mitigation* compiled for heterogeneous execution and managed using CEDR [87]. Figure 7.4 presents the idle and active PS and PL power for 100 seconds. At time 0, the PS consumes 1.6 W, while PL consumes 1.322 W when the system is idle. The PS power increases

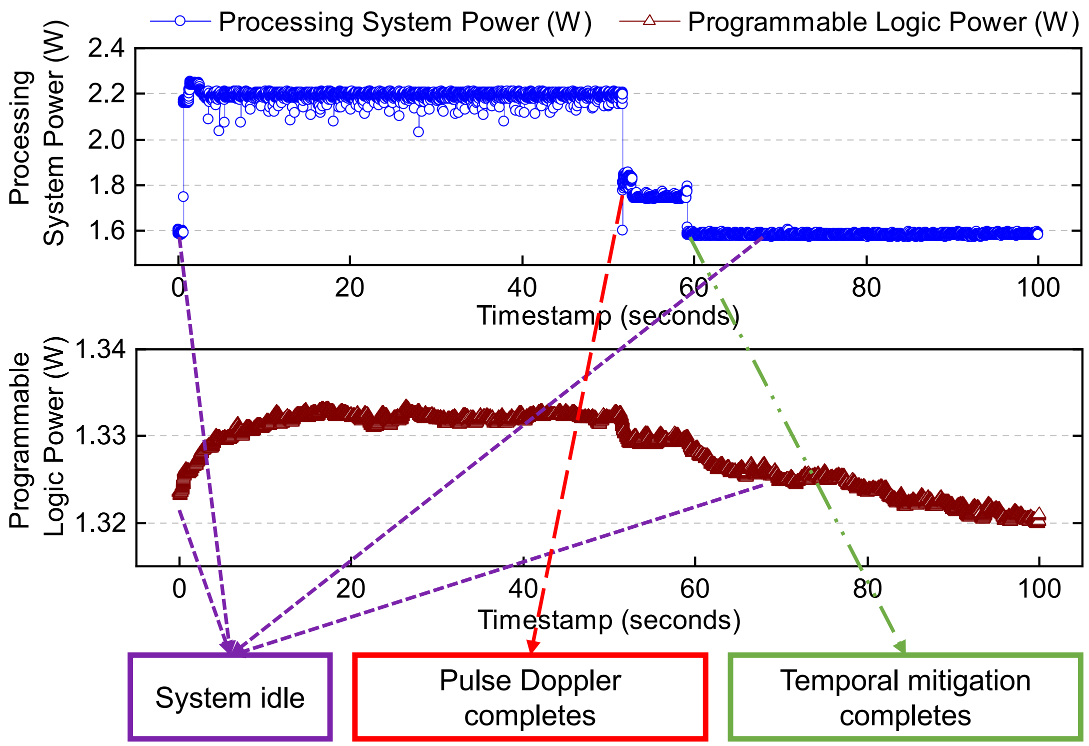


Figure 7.4: An illustration of power consumption in the processing system (Arm Cortex-A53 cores operating at 1.2 GHz) and programmable fabric (hardware accelerators operating at 100 MHz) in a Zynq UltraScale+ ZCU102 FPGA.

to 2.2 W when the applications start execution. The change in PL power remains insignificant because of its lower operating frequency and high static/idle power in the FPGA logic. The PS power reduces at 52 seconds when *pulse Doppler* completes and reduces to idle power of 1.6 W at 58 seconds when the *temporal mitigation* also ends. This capability allows developers to obtain early estimates and observe power consumption trends for preliminary analysis before the chip is taped out.

7.3.5 FPGA Implementation strategy

Although FALCON shares a similar design as the actual chip, there are a few critical aspects that must be addressed. *First*, a failure to generate the intended gated clocks in the design causes the tool to use cascaded clocks instead of generating a clock tree on the FPGA. This failure can also cause long placement and routing runtimes. While the tool must automatically generate gated clocks, developers must carefully check the gated clock conversion reports after synthesis and manually specify the clock relationships as required. *Second*, the design may instantiate memory modules from a specific process technology library. We should replace the memory instantiations with the FPGA memory primitive or a generic memory model. *Third*, the synthesis, placement and routing strategy should be tuned to the specific FPGA and the size of SoC. We use the *AlternateRoutability* for the accelerator sandbox synthesis and the *SpreadLogic* at the top-level due to the Finally, we achieve timing closure at 32MHz main clock frequency on the VCU128 and VU19P FPGAs.

Base System: The base system forms the general-purpose subsystem of the DSSoC. FALCON utilizes Arm's Corstone-700 as the base system. Corstone-700 is a flexible and configurable subsystem that houses the 32-bit Arm Cortex-A32 cores as the processing cluster. It also provides easy and flexible interfaces to integrate other system components and peripherals. The number of Cortex-A32 cores is configurable between 1–4. An AXI-based interrupt controller distributes the interrupts to the different on-chip components. The secure enclave and CoreSight unit address the system's security and debug services. We note that the ratios of all clock frequencies are maintained to maintain accuracy with the final tapeout. The flexibility of the architecture allows the base subsystem to be easily swapped with other Corstone subsystems or potentially with different types of host systems.

Accelerator Sandbox: The design and integration of components in a DSSoC are highly complex due to the large number and diverse processing elements. Therefore, the exploration phase involves frequent addition, modification, and removal of accelerators in a DSSoC. To address this concern, FALCON employs a modularized implementation of the interfacing of hardware accelerators with the base system. The accelerator sandbox is an independent module that uses standard AXI interfaces to connect to the system-level interconnect. The sandbox approach allows the rest of the system to observe only the AXI interfaces from the sandbox. It is oblivious to its internal architecture, providing a plug-and-play mechanism for integrating hardware accelerators. This architecture assumes each accelerator can master the memory bus or work closely with other accelerators or DMA engines to transfer data to the system. The sandbox provides interrupt lines to the accelerators to indicate control transfer to the base subsystem. The sandbox uses four AXI initiator-responder channels. Designers map the chosen number of channels among the different hardware accelerators and their data streams based on the latency, bandwidth, and throughput requirements. A DSSoC that targets wireless communication and radar applications may integrate accelerators for a finite impulse response (FIR-filter), fast Fourier transform (FFT), and matrix multiplication to accelerate frequently encountered tasks as illustrated in Figure 7.1. The plug-and-play mechanism allows designers to provide intra-sandbox communication between accelerators to improve data movement latencies. The sandbox can easily be extended to support multiple clocks and resets if the accelerators are required to operate at different frequencies.

On-Chip System Interconnect:

With the diverse processing elements on the chip, data movement is critical to ensure that the hardware has the necessary inputs to perform the required computation. Developers may choose to integrate low-latency mesh network-on-chip (NoC) interconnects (such as Arm CMN-600) or low-power crossbar based interconnects (such as Arm NIC-400) [207].
While the base system and system-level interconnect use Arm-based components, FALCON is not limited to Arm-based systems, and developers are free to integrate processing elements and interconnects of their choice. We note that the software stack (described in Section 7.3.6) would need appropriate updates to support the hardware choices.

7.3.6 Software Stack

DSSoCs demand an extensive software stack to exploit the full potential of the hardware architecture and provide comprehensive programming support to end-users and developers. FALCON is based on the Arm Corstone-700 base system. Hence, it utilizes the Arm reference platforms to produce the software stack [208]. The Arm reference platforms are based on the Yocto project to build customized Linux distributions. While this section describes configuring the Arm reference platforms for FALCON, the methodology is generic since the Yocto project is widely used to produce Linux distributions and software stacks. We emphasize that this

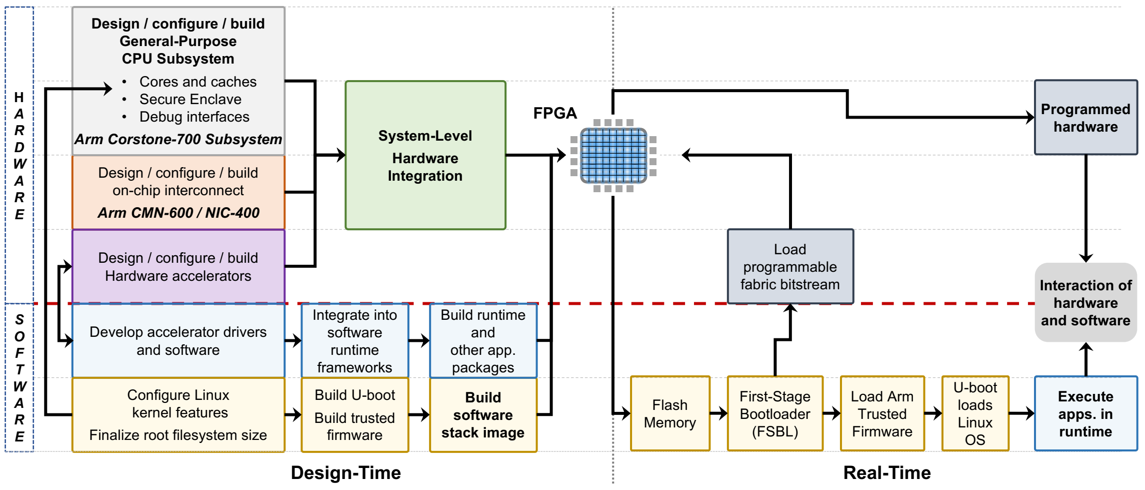

HARDWARE
SOFTWARE
Design / configure / build General-Purpose CPU Subsystem
Cores and caches
Secure Enclave
Debug interfaces
Arm Corstone-700 Subsystem
Design / configure / build on-chip interconnect
Arm CMN-600 / NIC-400
Design / configure / build Hardware accelerators
System-Level Hardware Integration
FPGA
Develop accelerator drivers and software
Integrate into software runtime frameworks
Build runtime and other app. packages
Configure Linux kernel features
Finalize root filesystem size
Build U-boot
Build trusted firmware
Build software stack image
Programmed hardware
Load programmable fabric bitstream
Interaction of hardware and software
Flash Memory
First-Stage Bootloader (FSBL)
Load Arm Trusted Firmware
U-boot loads Linux OS
Execute apps. in runtime
Design-Time
Real-Time

Figure 7.5: A timeline of the hardware development process and the boot sequence in software for a DSSoC emulated by the proposed FALCON framework.

software stack is fully deployable with standard security, virtualization, and the guarantees of a full-fledged Linux-based embedded system. This section focuses more on the specific configurations for the DSSoC configuration prototyped in this work.

The software stack integrates the following components to produce the entire software stack (as shown in Figure 7.1): (1) Linux kernel, (2) boot firmware, (3) trusted firmware, (4) U-boot, (5) root filesystem, and (6) application packages. The interactions between these components are captured in Figure 7.5.

Linux Kernel: The primary responsibilities of the kernel include (1) memory management, (2) process management, (3) device drivers, and (4) system calls and security. To support the hardware described in Section 7.2.1, FALCON makes the following modifications to the base configuration [208] for the Linux kernel:

- Enable multicore support through symmetric multiprocessing feature

- Configure input-output memory management unit (IOMMU) for multiple cores
- Enable kernel debugging capabilities
- Configure power state coordination interface (PSCI) for multiple cores
- Modify address pointers and image size of the software stack image stored in the flash memory

Boot Firmware: The boot firmware is the software for the secure enclave in the hardware architecture [208]. From the user perspective, the components that should be modified are the firewall access, system-wide memory map definitions and the interconnect initialization. The firewall determines the accessible/restricted memory regions of: (1) root file system, (2) Linux kernel, and (3) DDR. The boot firmware is converted into compiled binary code that is then built into the secure enclave hardware. Embedding the boot firmware into the hardware has a major implication in the DSSoC validation process, and this is precisely where early software development supported by FALCON plays a crucial role in developing bug-free and fully-functional SoCs (described in Section 7.4).
U-Boot: This software comprises the first-stage boot loader (FSBL) and a second-stage boot loader (U-boot). It is the primary component that handles hardware initialization and control hand-off to the OS for the booting process. FALCON modifies the Linux kernel address based on its size in the software image and the device tree address in U-Boot.

Trusted Firmware: The trusted firmware in FALCON comprises the critical security software for Arm-based processor systems. The default Corstone-700 stack boots only one Arm core. One of the most critical components is the power state coordination interface (PSCI) which is the interface for managing the idle cores, booting the secondary cores, and system shutdown/reset. FALCON modifies the PSCI firmware to power on the secondary cores and enables real-time access to multiple cores. FALCON also adds helper threads with assembly code to initialize and boot the secondary cores. The secondary core information is also specified in the device trees as entries in (1) the PSCI interfaces and (2) CPU cores. The device tree binary is built as part of the trusted firmware in FALCON's software stack.
Root Filesystem: The operating system's root filesystem (rootFS) contains the files and directories critical to the system's operation. By default, the Corstone-700 reference software stack provides a read-only file system. This requirement forces all the critical packages and features to be built into the rootFS during the build process. The packages to be integrated into the rootFS determine its size. It is also critical to reduce the rootFS size to minimize the boot time.
User Application Packages: The user applications range from libraries that include APIs to exercise the hardware accelerators to workloads, benchmarks, profilers, and performance monitors. The domain workload- and benchmark-source codes are cross-compiled for the specific Arm architecture (32-bit Arm v8 architecture in FALCON) and packaged into the software stack. Additionally, performance monitoring tools, such as *perf* that uses the performance monitoring unit to monitor the CPU pipelines and the interconnect, can be integrated to enable runtime performance

monitoring and evaluation. The FALCON emulation framework utilizes the CEDR runtime framework, which is also deployed as a user application package.

7.3.7 Software Runtimes

In this study for our experimental evaluations, we utilize the Compiler integrated Extensible DSSoC Runtime (CEDR) [87] ecosystem to conduct a design space exploration over the heterogeneous architecture emulated on the FPGA. This system allows end users to compile their applications for execution on a heterogeneous architecture and then interact with hardware by launching a workload composed of any number and combination of different applications with user-specified arrival rates. We choose CEDR over other runtime frameworks [209, 210] since it enables the compilation and development of user applications for heterogeneous SoCs, evaluating the performance of pre-silicon heterogeneous hardware configurations based on dynamically arriving workload scenarios through distinct plug-and-play integration points in a unified workflow. Furthermore, CEDR offers a rich set of integrated scheduling policies, allows integration of new policies through its distinct plug-and-play interfaces, offers collecting performance counter-based performance evaluation through "perf" utility, and, more importantly, enables conducting design space exploration in the trade space of hardware composition, workload complexity, scheduling policy over the user-defined performance metrics.

7.4 Enabling Software and Driver Development

The efforts involved in software design and driver development for DSSoCs are substantially higher due to the presence of hardware accelerators and specialized cores. Software development after fabrication significantly delays the time-to-market. To this end, pre-silicon FPGA-based emulation frameworks serve as a platform for software development and hardware-software codesign cycle.

7.4.1 Accelerator Drivers

While general-purpose cores have a well-established programming methodology in terms of programming languages and compilation toolchains, the hardware accelerator interfaces are mainly ad-hoc. They may not follow pre-defined protocols and languages. The interface to a hardware accelerator involves the following aspects: (1) configuration interface that allows the user to configure the accelerator based on the application parameters, (2) control interface that manages its initialization, starting, and completion, (3) data interface for the inputs and outputs. It is critical to validate these interfaces and data transfer protocols in the design stage. Another aspect involves determining the optimal burst size for input and output data transfers and the memory hierarchy for the data interfaces. Current approaches in the literature include analytical and performance models to estimate these effects, but the modeling accuracy limits them. Moreover, they are often evaluated with only portions of the system, resulting in estimation inaccuracies. FALCON enables evaluation in a full-system real-platform-like environment, providing highly accurate performance estimates. The precise evaluation allows designers to redesign the hardware and software architecture and interfaces as necessary to maximize metrics, including performance, throughput, bandwidth, and energy efficiency.

7.4.2 Enabling Performance Monitoring Unit (PMU)

The performance monitoring unit (PMU) records architectural and microarchitectural events and provides key performance indicators (KPIs). KPIs allow users to profile the applications and fine-tune the system parameters and architecture to maximize performance. Enabling the PMU in FALCON requires several changes to the software stack. First, the following features are enabled in the Linux kernel: (1) CONFIG_PROFILING, (2) CONFIG_PERF_EVENTS, (3) CONFIG_ARM_PMU, and (4) CONFIG_HW_PERF_EVENTS. The size of the Linux kernel in the software image increases when the PMU is enabled. This increase changes the address offsets and the size parameters in the boot firmware and the U-boot, as described in Section 7.3.6.

While the modifications described above are to the software stack, they strongly affect the hardware design. As described in Section 7.3.6, the boot firmware includes addresses, offsets, and sizes of the Linux kernel, rootFS, and the DDR memory, which are used in the firewall of the secure enclave. This information is packaged into the hardware design, making it infeasible to update these parameters after the chip is fabricated. To this end, FALCON enables all these hardware-software codesign aspects to ensure that the fabricated chip supports all intended features and functionality.

8 CONCLUSIONS AND FUTURE DIRECTIONS

The slowdown of Moore's Law and Dennard scaling has limited the power and performance gains obtained with the evolution of technology process nodes over the years. Beyond the conventional approaches to address this challenge, DSAs promise to achieve superior energy efficiency by combining general-purpose cores and hardware accelerators for applications in a target domain. This dissertation addressed several critical challenges in DSA design and development to fully exploit their potential and improve their performance and energy efficiency. First, we developed DS3, a high-level discrete-event full-system simulation framework that facilitates rapid design space exploration and extensive evaluation of resource management algorithms. Then, we posed scheduling as a classification problem and proposed the use of imitation learning and decision tree-based machine learning classifiers to perform runtime task scheduling. The proposed imitation learning based scheduling policy performs within 1% of an Oracle generated Oracle with minimal scheduling overheads. Next, the software optimization approaches proposed in this dissertation achieve a latency of less than 50 nanoseconds for decision trees of up to depth 12 on hardware platforms such as the Xilinx Zynq UltraScale+ ZCU102 and Nvidia Jetson Xavier NX. Scheduling policies generated using offline Oracles can quickly be rendered ineffective when new applications are introduced or new processing clusters are introduced. To this end, we proposed an online and lightweight training algorithm to adapt decision tree-based scheduling policies to changes in the workload and SoC configurations. The online algorithm utilizes merely 1-8% of the original training dataset and yet achieves a performance that is within 5% of a baseline implementation that trains trees from scratch using the entire training data. Finally, we developed end-to-end full-system FPGA-based emulation framework for DSAs that integrate general-purpose processors, hardware accelerators, memory hierarchies, and on-chip interconnects. It's hardware architecture and software stack, coupled with a runtime environment enable realistic application execution in a Linux-based operating system and allow full-system functional validation and early performance estimates.

In summary, this dissertation addressed several critical gaps in the design and development of DSAs by making the following contributions:

- A detailed and comprehensive literature review on DSAs and corresponding research directions [50],
- DS3, a domain-specific system-on-chip simulation framework to perform rapid design space exploration and evaluate resource management algorithms [51],
- An imitation learning (IL) based task scheduling approach [49],
- Optimization techniques for decision tree classifiers [52],
- An incremental and online decision tree training framework [53], and
- An FPGA-based emulation framework for domain-specific architectures.

8.1 Future Directions

High-Level Simulation Framework: The DS3 simulator presented in this dissertation is calibrated to real-world wireless communication and radar applications and to commercially available hardware platforms such as the Xilinx Zynq UltraScale+ ZCU102 FPGA and Odroid-XU3. As the fabrication and academic/commercial availability of DSAs become prominent, fine-tuning the models in DS3 based on power and performance measurements from real DSA silicon will improve its fidelity. Furthermore, exploring the applicability of DS3 to other application domains would improve DS3's scope and applicability.

Imitation learning based task scheduling: Machine learning (ML) based approaches rely heavily on the distribution of the training data to perform predictions. Real systems present runtime variations due to changes in system workload, memory, and congestion. The variations manifest as imperfections in the input data and result in outliers. However, the IL-based policies presented in this work must be robust to such outliers. In other words, the policies must ensure that they do not make illegal decisions that violate the constraints of the system. Incorporating robustness and other ancillary aspects such as risk awareness and security are interesting future directions.

This dissertation proposes use of the IL-based scheduling approach to perform task scheduling and our voltage-frequency scaling technique [113] to adjust the V-F levels and avoid deadline violations. However, future directions can consider building the deadline constraints directly into the task scheduling policies to consider deadlines at a finer granularity.

Incremental and online decision tree training framework: The online training technique proposed in this work uses only 1-8% of the original training data by storing its synopsis. Several compression algorithms such as arithmetic encoding and dictionary-based methods can aid in further reducing the memory footprint of the metadata. Future directions can consider such compression algorithms and efficient online Oracle generation approaches to minimize the impact of generating the ground truth and the memory footprint at runtime.

FPGA-based System-Level Emulation: This work illustrated the optimization of the memory hierarchy of domain applications between the on-chip cache and scratchpad memory based on the kernel sizes. While the hardware cache controller is responsible for selecting the data stored in the cache, the scratchpad memory management relies on the application programmer or the runtime framework to manage the data. In the future, compilers, runtime frameworks, and hardware interactions can automatically partition the scratchpad memory based on the domain applications and kernel sizes, thereby relieving the programmer's burden. Finally, updating the high-level simulators with models that describe the data partitioning between caches and scratchpad memory improves their power and performance estimates.

BIBLIOGRAPHY

[1] Haluk Topcuoglu, Salim Hariri, and Min-You Wu. Performance-Effective and Low-Complexity Task Scheduling for Heterogeneous Computing. *IEEE Transactions on Parallel and Distributed Systems*, 13(3):260–274, 2002.

[2] Albert Bifet, Geoff Holmes, Bernhard Pfahringer, and Eibe Frank. Fast Perceptron Decision Tree Learning from Evolving Data Streams. In *Pacific-Asia Conference on Knowledge Discovery and Data Mining*, pages 299–310. Springer, 2010.

[3] Andrew Silva, Matthew Gombolay, Taylor Killian, Ivan Jimenez, and Sung-Hyun Son. Optimization Methods for Interpretable Differentiable Decision Trees Applied to Reinforcement Learning. In *International Conference on Artificial Intelligence and Statistics*, pages 1855–1865. PMLR, 2020.

[4] Fabian Pedregosa, Gaël Varoquaux, Alexandre Gramfort, Vincent Michel, Bertrand Thirion, Olivier Grisel, Mathieu Blondel, Peter Prettenhofer, Ron Weiss, Vincent Dubourg, et al. Scikit-learn: Machine Learning in Python. *Journal of Machine Learning Research*, 12:2825–2830, 2011.

[5] Thomas N Theis and H-S Philip Wong. The End of Moore's Law: A New Beginning for Information Technology. *Computing in Science & Engineering*, 19(2):41–50, 2017.

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

[7] Hadi Esmaeilzadeh, Emily Blem, Renee St Amant, Karthikeyan Sankaralingam, and Doug Burger. Dark Silicon and the End of Multicore Scaling. In *38th Annual International Symposium on Computer Architecture (ISCA)*, pages 365–376. IEEE, 2011.

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

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

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

[11] James Jeffers, James Reinders, and Avinash Sodani. *Intel Xeon Phi Processor High Performance Programming: Knights Landing Edition*. Morgan Kaufmann, 2016.

[12] Pawel Gepner and Michal Filip 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, 2006.

[13] Edson Luiz Padoin, Laércio Lima Pilla, Márcio Castro, Francieli Z Boito, Philippe Olivier Alexandre Navaux, and Jean-François 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.

[14] Hardkernel. ODROID-XU3. `https://wiki.odroid.com/old_product/odroid-xu3/odroid-xu3`. [Online; last accessed 01-Dec-2022.].

[15] Vinay Hanumaiah and Sarma Vrudhula. Energy-Efficient Operation of Multicore Processors by DVFS, Task Migration, and Active Cooling. *IEEE Transactions on Computers*, 63(2):349–360, 2012.

[16] Microprocessor Report: PC Processors Adopt Hybrid CPUs. `https://www.techinsights.com/blog/year-review-pc-processors-adopt-hybrid-cpus`. [Online; last accessed 01-Dec-2022.].

[17] Sparsh Mittal and Jeffrey S Vetter. A Survey of CPU-GPU Heterogeneous Computing Techniques. *ACM Computing Surveys (CSUR)*, 47(4):1–35, 2015.

[18] Yao Xiao, Shahin Nazarian, and Paul Bogdan. Self-optimizing and Self-programming Computing Systems: A Combined Compiler, Complex Networks, and Machine Learning Approach. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, 27(6):1416–1427, 2019.

[19] Alcides Fonseca and Bruno Cabral. Prototyping a GPGPU Neural Network for Deep-learning Big Data Analysis. *Big Data Research*, 8:50–56, 2017.

[20] Santanu Sarma and Nikil Dutt. FPGA Emulation and Prototyping of a Cyberphysical-System-on-Chip (CPSoC). In *IEEE International Symposium on Rapid System Prototyping*, pages 121–127, 2014.

[21] John Hennessy and David Patterson. A New Golden Age for Computer Architecture: Domain-Specific Hardware/Software Co-Design, Enhanced. In *ACM/IEEE 45th Annual International Symposium on Computer Architecture (ISCA)*, 2018.

[22] John L Hennessy and David A Patterson. A New Golden Age for Computer Architecture. *Communications of the ACM*, 62(2):48–60, 2019.

[23] Aporva Amarnath, Subhankar Pal, Hiwot Tadese Kassa, Augusto Vega, Alper Buyuktosunoglu, Hubertus Franke, John-David Wellman, Ronald Dreslinski, and Pradip Bose. Heterogeneity-Aware Scheduling on SoCs for Autonomous Vehicles. *IEEE Computer Architecture Letters*, 20(2):82–85, 2021.

[24] RF Convergence: From the Signals to the Computer by Dr. Tom Rondeau (Microsystems Technology Office, DARPA). `https://futurenetworks.ieee.org/images/files/pdf/FirstResponder/Tom-Rondeau-DARPA.pdf`. [Online; last accessed 01-Dec-2022.].

[25] Yigit Tuncel, Anish Krishnakumar, Aishwarya Lekshmi Chithra, Younghyun Kim, and Umit Ogras. A Domain-Specific System-On-Chip Design for Energy Efficient Wearable Edge AI Applications. In *Proceedings of the ACM/IEEE International Symposium on Low Power Electronics and Design*, pages 1–6, 2022.

[26] Ganapati Bhat, Kunal Bagewadi, Hyung Gyu Lee, and Umit Y Ogras. REAP: Runtime Energy-Accuracy Optimization for Energy Harvesting IoT Devices. In *56th ACM/IEEE Design Automation Conference (DAC)*, pages 1–6, 2019.

[27] Ranadeep Deb, Ganapati Bhat, Sizhe An, Holly Shill, and Umit Y Ogras. Trends in Technology Usage for Parkinson's Disease Assessment: A Systematic Review. *MedRxiv*, 2021.

[28] NK Anish, Ganapati Bhat, Jaehyun Park, Hyung Gyu Lee, and Umit Y Ogras. Sensor-classifier Co-optimization for Wearable Human Activity Recognition Applications. In *IEEE International Conference on Embedded Software and Systems (ICESS)*, pages 1–4, 2019.

[29] Nagadastagiri Challapalle, Sahithi Rampalli, Linghao Song, Nandhini Chandramoorthy, Karthik Swaminathan, John Sampson, Yiran Chen, and Vijaykrishnan Narayanan. GaaS-X: Graph Analytics Accelerator Supporting Sparse Data Representation using Crossbar Architectures. In *ACM/IEEE 47th Annual International Symposium on Computer Architecture (ISCA)*, pages 433–445, 2020.

[30] DW Bliss, T Ajayi, A Akoglu, I Aliyev, T Basaklar, L Belayneh, D Blaauw, J Brunhaver, C Chakrabarti, L Chang, et al. Enabling Software-Defined RF Convergence with a Novel Coarse-Scale Heterogeneous Processor. In *IEEE International Symposium on Circuits and Systems (ISCAS)*, pages 443–447, 2022.

[31] Niall O'Mahony, Sean Campbell, Anderson Carvalho, Suman Harapanahalli, Gustavo Velasco Hernandez, Lenka Krpalkova, Daniel Riordan, and Joseph Walsh. Deep Learning vs. Traditional Computer Vision. In *Science and Information Conference*, pages 128–144. Springer, 2019.

[32] Gokul Krishnan, Sumit K Mandal, Manvitha Pannala, Chaitali Chakrabarti, Jae-Sun Seo, Umit Y Ogras, and Yu Cao. SIAM: Chiplet-based Scalable In-Memory Acceleration with Mesh for Deep Neural Networks. *ACM Transactions on Embedded Computing Systems (TECS)*, 20(5s):1–24, 2021.

[33] Hongzi Mao, Mohammad Alizadeh, Ishai Menache, and Srikanth Kandula. Resource Management with Deep Reinforcement Learning. In *ACM Workshop on Hot Topics in Networks*, pages 50–56, 2016.

[34] Ivens Portugal, Paulo Alencar, and Donald Cowan. The Use of Machine Learning Algorithms in Recommender Systems: A Systematic Review. *Expert Systems with Applications*, 97:205–227, 2018.

[35] Jenna Wiens and Erica S Shenoy. Machine Learning for Healthcare: On the Verge of a Major Shift in Healthcare Epidemiology. *Clinical Infectious Diseases*, 66(1):149–153, 2018.

[36] Nagadastagiri Challapalle, Sahithi Rampalli, Makesh Chandran, Gurpreet Kalsi, Sreenivas Subramoney, John Sampson, and Vijaykrishnan Narayanan. PSB-RNN: A Processing-in-Memory Systolic Array Architecture using Block Circulant Matrices for Recurrent Neural Networks. In *Design, Automation & Test in Europe Conference & Exhibition (DATE)*, pages 180–185, 2020.

[37] Nagadastagiri Challapalle, Makesh Chandran, Sahithi Rampalli, and Vijaykrishnan Narayanan. X-VS: Crossbar-Based Processing-in-Memory Architecture for Video Summarization. In *IEEE Computer Society Annual Symposium on VLSI (ISVLSI)*, pages 592–597, 2020.

[38] Nagadastagiri Challapalle and Vijaykrishnan Narayanan. Performance Evaluation of Video Analytics Workloads on Emerging Processing-In-Memory Architectures. In *IEEE Computer Society Annual Symposium on VLSI (ISVLSI)*, pages 158–163, 2022.

[39] Nagadastagiri Challapalle, Karthik Swaminathan, Nandhini Chandramoorthy, and Vijaykrishnan Narayanan. Crossbar based Processing in Memory Accelerator Architecture for Graph Convolutional Networks. In *IEEE/ACM International Conference On Computer Aided Design (ICCAD)*, pages 1–9, 2021.

[40] Nagadastagiri Challapalle, Sahithi Rampalli, Nicholas Jao, Akshaykrishna Ramanathan, John Sampson, and Vijaykrishnan Narayanan. FARM: A Flex-

ible Accelerator for Recurrent and Memory Augmented Neural Networks. *Journal of Signal Processing Systems*, 92(11):1247–1261, 2020.

[41] Thomas Norrie, Nishant Patil, Doe Hyun Yoon, George Kurian, Sheng Li, James Laudon, Cliff Young, Norman Jouppi, and David Patterson. The Design Process for Google's Training Chips: TPUv2 and TPUv3. *IEEE Micro*, 41(2):56–63, 2021.

[42] David Patterson. 50 Years of Computer Architecture: From the Mainframe CPU to the Domain-Specific TPU and the Open RISC-V Instruction Set. In *IEEE International Solid-State Circuits Conference-(ISSCC)*, pages 27–31. IEEE, 2018.

[43] Yu Emma Wang, Gu-Yeon Wei, and David Brooks. Benchmarking TPU, GPU, and CPU Platforms for Deep Learning. *arXiv preprint arXiv:1907.10701*, 2019.

[44] Norman P Jouppi, Doe Hyun Yoon, Matthew Ashcraft, Mark Gottscho, Thomas B Jablin, George Kurian, James Laudon, Sheng Li, Peter Ma, Xiaoyu Ma, et al. Ten Lessons from Three Generations Shaped Google's TPUv4i: Industrial Product. In *ACM/IEEE 48th Annual International Symposium on Computer Architecture (ISCA)*, pages 1–14, 2021.

[45] Idan Burstein. Nvidia Data Center Processing Unit (DPU) Architecture. In *IEEE Hot Chips 33 Symposium (HCS)*, pages 1–20, 2021.

[46] Brad Burres, Dan Daly, Mark Debbage, Eliel Louzoun, Christine Severns-Williams, Naru Sundar, Nadav Turbovich, Barry Wolford, and Yadong Li. Intel's Hyperscale-Ready Infrastructure Processing Unit (IPU). In *IEEE Hot Chips 33 Symposium (HCS)*, pages 1–16, 2021.

[47] Yvan Tortorella, Luca Bertaccini, Davide Rossi, Luca Benini, and Francesco Conti. RedMulE: A Compact FP16 Matrix-Multiplication Accelerator for Adaptive Deep Learning on RISC-V-Based Ultra-Low-Power SoCs. *arXiv preprint arXiv:2204.11192*, 2022.

[48] Augusto Vega, John-David Wellman, Hubertus Franke, Alper Buyuktosunoglu, Pradip Bose, Aporva Amarnath, Hiwot Kassa, Subhankar Pal, and Ronald Dreslinski. STOMP: Agile Evaluation of Scheduling Policies in Heterogeneous Multi-Processors. In *DOSSA-3 Workshop@ HPCA*, 2021.

[49] Anish Krishnakumar, Samet E Arda, A Alper Goksoy, Sumit K Mandal, Umit Y Ogras, Anderson L Sartor, and Radu Marculescu. Runtime Task Scheduling using Imitation Learning for Heterogeneous Many-core Systems. *IEEE Transactions on CAD of Integrated Circuits and Systems*, 39(11):4064–4077, 2020.

[50] Anish Krishnakumar, Umit Y Ogras, Radu Marculescu, Michael Kishinevsky, and Trevor Mudge. Domain-Specific Architectures (DSAs): Research Problems and Promising Approaches. *ACM Transactions on Embedded Computing Systems (TECS)*, 2022.

[51] Samet Arda, Anish Krishnakumar, Ahmet Alper Goksoy, Joshua Mack, Nirmal Kumbhare, Anderson Luiz Sartor, Ali Akoglu, Radu Marculescu, and Umit Y Ogras. DS3: A System-Level Domain-Specific System-on-Chip Simulation Framework. *IEEE Transactions on Computers*, 69(8):1248–1262, 2020.

[52] Anish Krishnakumar and Umit 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.

[53] Anish Krishnakumar, Radu Marculescu, and Umit Ogras. INDENT: Incremental Online Decision Tree Training for Domain-Specific Systems-on-Chip. In *Proceedings of the 41st IEEE/ACM International Conference on Computer-Aided Design*, pages 1–9, 2022.

[54] Paolo Mantovani, Davide Giri, Giuseppe Di Guglielmo, Luca Piccolboni, Joseph Zuckerman, Emilio G Cota, Michele Petracca, Christian Pilato, and

Luca P Carloni. Agile SoC Development with Open ESP. In *IEEE/ACM International Conference On Computer Aided Design (ICCAD)*, pages 1–9, 2020.

[55] Davide Giri, Kuan-Lin Chiu, Guy Eichler, Paolo Mantovani, and Luca P Carloni. Accelerator Integration for Open-Source SoC Design. *IEEE Micro*, 41(4):8–14, 2021.

[56] Richard Uhrie, Daniel W Bliss, Chaitali Chakrabarti, Umit Y Ogras, and John Brunhaver. Machine understanding of domain computation for Domain-Specific System-on-Chips (DSSoC). In *Open Architecture/Open Business Model Net-Centric Systems and Defense Transformation 2019*, volume 11015, pages 180 – 187. International Society for Optics and Photonics, SPIE, 2019.

[57] Richard Uhrie, Chaitali Chakrabarti, and John Brunhaver. Automated Parallel Kernel Extraction from Dynamic Application Traces. *arXiv preprint arXiv:2001.09995*, 2020.

[58] Jeronimo Castrillon, Andreas Tretter, Rainer Leupers, and Gerd Ascheid. Communication-Aware Mapping of KPN Applications onto Heterogeneous MPSoCs. In *Proceedings of the 49th Annual Design Automation Conference*, pages 1266–1271. ACM, 2012.

[59] Lodewijk T Smit, Johann L Hurink, and Gerard JM Smit. Run-time Mapping of Applications to a Heterogeneous SoC. In *International Symposium on System-on-Chip*, pages 78–81. IEEE, 2005.

[60] Ewerson Luiz de Souza Carvalho, Ney Laert Vilar Calazans, and Fernando Gehm Moraes. Dynamic Task Mapping for MPSoCs. *IEEE Design & Test of Computers*, 27(5):26–35, 2010.

[61] Chen-Ling Chou, Umit Y Ogras, and Radu Marculescu. Energy-and Performance-Aware Incremental Mapping for Networks on chip with Multiple Voltage Levels. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 27(10):1866–1879, 2008.

[62] Jurn-Gyu Park, Nikil Dutt, Hoyeonjiki Kim, and Sung-Soo Lim. HiCAP: Hierarchical FSM-based Dynamic Integrated CPU-GPU Frequency Capping Governor for Energy-Efficient Mobile Gaming. In *Proceedings of the International Symposium on Low Power Electronics and Design*, pages 218–223. ACM, 2016.

[63] Dominik Brodowski et al. Linux CPUFreq Governors. `https://www.kernel.org/doc/Documentation/cpu-freq/governors.txt`.

[64] Ayse Kivilcim Coskun, Tajana Simunic Rosing, and Kenny C Gross. Temperature Management in Multiprocessor SoCs Using Online Learning. In *45th ACM/IEEE Design Automation Conference*, pages 890–893. IEEE, 2008.

[65] Ashish Kumar Maurya and Anil Kumar Tripathi. On Benchmarking Task Scheduling Algorithms for Heterogeneous Computing Systems. *The Journal of Supercomputing*, 74(7):3039–3070, 2018.

[66] Nima Khalilzad, Kathrin Rosvall, and Ingo Sander. A Modular Design Space Exploration Framework for Multiprocessor Real-Time Systems. In *Forum on Specification and Design Languages (FDL)*, pages 1–7. IEEE, 2016.

[67] Kai Neubauer, Philipp Wanko, Torsten Schaub, and Christian Haubelt. Exact Multi-Objective Design Space Exploration using ASPmT. In *Design, Automation & Test in Europe Conference & Exhibition (DATE)*, pages 257–260. IEEE, 2018.

[68] Nikola Trčka, Martijn Hendriks, Twan Basten, Marc Geilen, and Lou Somers. Integrated Model-Driven Design-Space Exploration for Embedded Systems. In *International Conference on Embedded Computer Systems: Architectures, Modeling and Simulation*, pages 339–346. IEEE, 2011.

[69] Bart Kienhuis, Ed Deprettere, Kees Vissers, and Pieter Van Der Wolf. An Approach for Quantitative Analysis of Application-Specific Dataflow Architectures. In *International Conference on Application-Specific Systems, Architectures and Processors*, pages 338–349. IEEE, 1997.

[70] Twan Basten, Emiel van Benthum, Marc Geilen, Martijn Hendriks, Fred Houben, Georgeta Igna, Frans Reckers, Sebastian de Smet, Lou Somers, Egbert Teeselink, et al. Model-Driven Design-Space Exploration for Embedded Systems: The Octopus Toolset. In *International Symposium On Leveraging Applications of Formal Methods, Verification and Validation*, pages 90–105. Springer, 2010.

[71] Andy D Pimentel, LO Hertzbetger, Paul Lieverse, Pieter Van Der Wolf, and EE Deprettere. Exploring Embedded-Systems Architectures with Artemis. *Computer*, 34(11):57–63, 2001.

[72] Andy D Pimentel, Cagkan Erbas, and Simon Polstra. A Systematic Approach to Exploring Embedded System Architectures at Multiple Abstraction Levels. *IEEE Transactions on Computers*, 55(2):99–112, 2006.

[73] Giovanni Beltrame, Luca Fossati, and Donatella Sciuto. ReSP: A Nonintrusive Transaction-Level Reflective MPSoC Simulation Platform for Design Space Exploration. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 28(12):1857–1869, 2009.

[74] Cédric Augonnet, Samuel Thibault, Raymond Namyst, and Pierre-André Wacrenier. StarPU: A Unified Platform for Task Scheduling on Heterogeneous Multicore Architectures. *Concurrency and Computation: Practice and Experience*, 23(2):187–198, 2011.

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

[76] Nicolas Genko, David Atienza, Giovanni De Micheli, and Luca Benini. Feature-NoC Emulation: A Tool and Design Flow for MPSoC. *IEEE Circuits and Systems Magazine*, 7(4):42–51, 2007.

[77] John R Mashey. Interactions, Impacts, and Coincidences of the First Golden Age of Computer Architecture. *IEEE Micro*, 41(6):131–139, 2021.

[78] Wen Chen, Sandip Ray, Jayanta Bhadra, Magdy Abadir, and Li-C Wang. Challenges and Trends in Modern SoC Design Verification. *IEEE Design & Test*, 34(5):7–22, 2017.

[79] Amir Charif, Gabriel Busnot, Rania Mameesh, Tanguy Sassolas, and Nicolas Ventroux. Fast Virtual Prototyping for Embedded Computing Systems Design and Exploration. In *Proceedings of the Rapid Simulation and Performance Evaluation: Methods and Tools*, pages 1–8, 2019.

[80] Fabrice Bellard. QEMU, A Fast and Portable Dynamic Translator. In *USENIX Annual Technical Conference, FREENIX Track*, volume 41, pages 10–5555, 2005.

[81] Daniel Bartholomew. QEMU: A Multihost, Multitarget Emulator. *Linux Journal*, 2006(145):3, 2006.

[82] Prashant Varanasi and Gernot Heiser. Hardware-Supported Virtualization on ARM. In *Proceedings of the Second Asia-Pacific Workshop on Systems*, pages 1–5, 2011.

[83] Fabrizio Mulas, David Atienza, Andrea Acquaviva, Salvatore Carta, Luca Benini, and Giovanni De Micheli. Thermal Balancing Policy for Multiprocessor Stream Computing Platforms. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 28(12):1870–1882, 2009.

[84] David Atienza, Pablo G Del Valle, Giacomo Paci, Francesco Poletti, Luca Benini, Giovanni De Micheli, Jose M Mendias, and Roman Hermida. HW-SW Emulation Framework for Temperature-aware Design in MPSoCs. *ACM Transactions on Design Automation of Electronic Systems (TODAES)*, 12(3):1–26, 2008.

[85] Mostafa Khamis, Sameh El-Ashry, Ahmed Shalaby, Mohamed AbdElsalam, and M Watheq El-Kharashi. A Configurable RISC-V for NoC-based MPSoCs:

A Framework for Hardware Emulation. In *11th International Workshop on Network on Chip Architectures (NoCArc)*, pages 1–6, 2018.

[86] Andreas Kurth, Pirmin Vogel, Alessandro Capotondi, Andrea Marongiu, and Luca Benini. HERO: Heterogeneous Embedded Research Platform for Exploring RISC-V Manycore Accelerators on FPGA. *arXiv preprint arXiv:1712.06497*, 2017.

[87] Joshua Mack, Nirmal Kumbhare, Anish NK, Umit Y Ogras, and Ali Akoglu. User-Space Emulation Framework for Domain-Specific SoC Design. In *IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW)*, pages 44–53, 2020.

[88] Joshua Mack, Serhan Gener, Ali Akoglu, Jacob Holtom, Alex Chiriyath, Chaitali Chakrabarti, Daniel Bliss, Anish Krishnakumar, Alper Goksoy, and Umit Ogras. GNU Radio and CEDR: Runtime Scheduling to Heterogeneous Accelerators. In *Proceedings of the GNU Radio Conference*, volume 7, 2022.

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

[90] Umair Ullah Tariq, Hui Wu, and Suhaimi Abd Ishak. Energy-aware Scheduling of Conditional Task Graphs on NoC-based MPSoCs. In *Proceedings of the 51st Hawaii International Conference on System Sciences*, 2018.

[91] Junlong Zhou, Jin Sun, Peijin Cong, Zhe Liu, Xiumin Zhou, Tongquan Wei, and Shiyan Hu. Security-critical Energy-aware Task Scheduling for Heterogeneous Real-time MPSoCs in IoT. *IEEE Transactions on Services Computing*, 13(4):745–758, 2019.

[92] Andy D Pimentel, Cagkan Erbas, and Simon Polstra. A Systematic Approach to Exploring Embedded System Architectures at Multiple Abstraction Levels. *IEEE Transactions on Computers*, 55(2):99–112, 2006.

[93] Amit Kumar Singh, Muhammad Shafique, Akash Kumar, and Jörg Henkel. Mapping on Multi/Many-Core Systems: Survey of Current and Emerging Trends. In *50th ACM/EDAC/IEEE Design Automation Conference (DAC)*, pages 1–10, 2013.

[94] Andreas Gerstlauer, Christian Haubelt, Andy D Pimentel, Todor P Stefanov, Daniel D Gajski, and Jürgen Teich. Electronic System-Level Synthesis Methodologies. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 28(10):1517–1530, 2009.

[95] Joonho Kong, Sung Woo Chung, and Kevin Skadron. Recent Thermal Management Techniques for Microprocessors. *ACM Computing Surveys (CSUR)*, 44(3):1–42, 2012.

[96] Luca Benini, Alessandro Bogliolo, and Giovanni De Micheli. A Survey of Design Techniques for System-Level Dynamic Power Management. *IEEE Transactions on Very Scale Integration (VLSI) Systems*, 8(3):299–316, 2000.

[97] Chandandeep Singh Pabla. Completely Fair Scheduler. *Linux Journal*, 2009(184), 2009.

[98] Tobias Beisel, Tobias Wiersema, Christian Plessl, and André 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.

[99] Mihaela-Andreea Vasile, Florin Pop, Radu-Ioan Tutueanu, Valentin Cristea, and Joanna Kołodziej. Resource-Aware Hybrid Scheduling Algorithm in Heterogeneous Distributed Computing. *Future Generation Computer Systems*, 51:61–71, 2015.

[100] Rizos Sakellariou and Henan Zhao. A Hybrid Heuristic for DAG Scheduling on Heterogeneous Systems. In *International Parallel and Distributed Processing Symposium*, page 111. IEEE, 2004.

[101] Yu-Kwong Kwok and Ishfaq Ahmad. Dynamic Critical-Path Scheduling: An Effective Technique for Allocating Task Graphs to Multiprocessors. *IEEE Transactions on Parallel and Distributed Systems*, 7(5):506–521, 1996.

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

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

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

[105] Guoqi Xie, Gang Zeng, Liangjiao Liu, Renfa Li, and Keqin Li. Mixed Real-Time Scheduling of Multiple DAGs-based Applications on Heterogeneous Multi-core Processors. *Microprocessors and Microsystems*, 47:93–103, 2016.

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

[107] Yuan Wen, Zheng Wang, and Michael FP O'Boyle. Smart Multi-Task Scheduling for OpenCL Programs on CPU/GPU Heterogeneous Platforms. In *International Conference on High Performance Computing*, pages 1–10, 2014.

[108] Azalia Mirhoseini, Hieu Pham, Quoc V Le, Benoit Steiner, Rasmus Larsen, Yuefeng Zhou, Naveen Kumar, Mohammad Norouzi, Samy Bengio, and Jeff Dean. Device Placement Optimization with Reinforcement Learning. In *International Conference on Machine Learning-Volume 70*, pages 2430–2439. JMLR. org, 2017.

[109] Hongzi Mao, Malte Schwarzkopf, Shaileshh Bojja Venkatakrishnan, Zili Meng, and Mohammad Alizadeh. Learning Scheduling Algorithms for Data Processing Clusters. In *Special Interest Group on Data Communication*, pages 270–288. ACM, 2019.

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

[111] Ryan Gary Kim, Wonje Choi, Zhuo Chen, Janardhan Rao Doppa, Partha Pratim Pande, Diana Marculescu, and Radu Marculescu. Imitation Learning for Dynamic VFI Control in Large-Scale Manycore Systems. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, 25(9):2458–2471, 2017.

[112] Sumit K Mandal, Ganapati Bhat, Janardhan Rao Doppa, Partha Pratim Pande, and Umit 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.

[113] Anderson L Sartor, Anish Krishnakumar, Samet E Arda, Umit Y Ogras, and Radu Marculescu. HiLITE: Hierarchical and Lightweight Imitation Learning For Power Management Of Embedded SoCs. *IEEE CAL*, 19(1):63–67, 2020.

[114] Ching-Chi Lin, You-Cheng Syu, Chao-Jui Chang, Jan-Jan Wu, Pangfeng Liu, Po-Wen Cheng, and Wei-Te Hsu. Energy-efficient Task Scheduling for Multi-core Platforms with Per-core DVFS. *Journal of Parallel and Distributed Computing*, 86:71–81, 2015.

[115] Ujjwal Gupta, Manoj Babu, Raid Ayoub, Michael Kishinevsky, Francesco Paterna, and Umit Y Ogras. STAFF: Online Learning with Stabilized Adaptive Forgetting Factor and Feature Selection Algorithm. In *55th ACM/ESDA/IEEE Design Automation Conference (DAC)*, pages 1–6, 2018.

[116] Ganapati Bhat, Gaurav Singla, Ali K Unver, and Umit Y Ogras. Algorithmic Optimization of Thermal and Power Management for Heterogeneous Mobile

Platforms. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, 26(3):544–557, 2017.

[117] Vinay Hanumaiah, Digant Desai, Benjamin Gaudette, Carole-Jean Wu, and Sarma Vrudhula. STEAM: A Smart Temperature and Energy Aware Multicore Controller. *ACM Transactions on Embedded Computing Systems (TECS)*, 13(5s):1–25, 2014.

[118] Zhuo Tang, Ling Qi, Zhenzhen Cheng, Kenli Li, Samee U Khan, and Keqin Li. An Energy-Efficient Task Scheduling Algorithm in DVFS-enabled Cloud Environment. *Journal of Grid Computing*, 14(1):55–74, 2016.

[119] Sodam Han, Yonghee Yun, Young Hwan Kim, and Seokhyeong Kang. Proactive Scenario Characteristic-aware Online Power Management on Mobile Systems. *IEEE Access*, 8:69695–69711, 2020.

[120] Basireddy Karunakar Reddy, Amit Kumar Singh, Dwaipayan Biswas, Geoff V Merrett, and Bashir 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.

[121] Kasra Moazzemi, Biswadip Maity, Saehanseul Yi, Amir M Rahmani, and Nikil Dutt. HESSLE-FREE: Heterogeneous Systems Leveraging Fuzzy Control for Runtime Resource Management. *ACM Transactions on Embedded Computing Systems (TECS)*, 18(5s):1–19, 2019.

[122] Raid Ayoub, Umit Ogras, Eugene Gorbatov, Yanqin Jin, Timothy Kam, Paul Diefenbaugh, and Tajana Rosing. OS-Level Power Minimization under Tight Performance Constraints in General Purpose Systems. In *International Symposium on Low Power Electronics and Design*, pages 321–326, 2011.

[123] Sumit K Mandal, Ganapati Bhat, Chetan Arvind Patil, Janardhan Rao Doppa, Partha Pratim Pande, and Umit Y Ogras. Dynamic Resource Management of Heterogeneous Mobile Platforms via Imitation Learning. *Transactions on Very Large Scale Integration (VLSI) Systems*, 2019.

[124] Bryan Donyanavard, Tiago Mück, Santanu Sarma, and Nikil Dutt. SPARTA: Runtime Task Allocation for Energy Efficient Heterogeneous Manycores. In *International Conference on Hardware/Software Codesign and System Synthesis (CODES+ ISSS)*, pages 1–10, 2016.

[125] Onur Sahin and Ayse K Coskun. Providing Sustainable Performance in Thermally Constrained Mobile Devices. In *Proceedings of the 14th ACM/IEEE Symposium on Embedded Systems for Real-Time Multimedia*, pages 72–77, 2016.

[126] Somdip Dey, Amit Kumar Singh, and Klaus Dieter McDonald-Maier. P-edgecoolingmode: An Agent-based Performance Aware Thermal Management Unit for DVFS Enabled Heterogeneous MPSoCs. *IET Computers & Digital Techniques*, 13(6):514–523, 2019.

[127] Sudeep Pasricha, Raid Ayoub, Michael Kishinevsky, Sumit K Mandal, and Umit Y Ogras. A Survey on Energy Management for Mobile and IoT Devices. *IEEE Design & Test*, 37(5):7–24, 2020.

[128] Yen-Kuan Wu, Shervin Sharifi, and Tajana Simunic Rosing. Distributed Thermal Management for Embedded Heterogeneous MPSoCs with Dedicated Hardware Accelerators. In *IEEE 29th International Conference on Computer Design (ICCD)*, pages 183–189, 2011.

[129] Sandeep D'souza and Ragunathan Rajkumar. CycleTandem: Energy-Saving Scheduling for Real-Time Systems with Hardware Accelerators. In *IEEE Real-Time Systems Symposium (RTSS)*, pages 94–106, 2018.

[130] Arm TrustZone. `https://developer.arm.com/documentation/PRD29-GENC-009492/c/TrustZone-Hardware-Architecture`. [Online; last accessed 01-Dec-2022.].

[131] Apple Secure Enclave. `https://support.apple.com/guide/security/secure-enclave-sec59b0b31ff/web`. [Online; last accessed 01-Dec-2022.].

[132] Kit Murdock, David Oswald, Flavio D Garcia, Jo Van Bulck, Daniel Gruss, and Frank Piessens. Plundervolt: Software-based fault injection attacks against Intel SGX. In *IEEE Symposium on Security and Privacy (SP)*, pages 1466–1482, 2020.

[133] Zhixin Pan and Prabhat Mishra. Automated Test Generation for Hardware Trojan Detection using Reinforcement Learning. In *Proceedings of the 26th Asia and South Pacific Design Automation Conference*, pages 408–413, 2021.

[134] Chen Liu, Jeyavijayan Rajendran, Chengmo Yang, and Ramesh Karri. Shielding Heterogeneous MPSoCs from Untrustworthy 3PIPs through Security-driven Task Scheduling. *IEEE Transactions on Emerging Topics in Computing*, 2(4):461–472, 2014.

[135] Michael Hutter and Jörn-Marc Schmidt. The Temperature Side Channel and Heating Fault Attacks. In *International Conference on Smart Card Research and Advanced Applications*, pages 219–235. Springer, 2013.

[136] Somdip Dey, Amit Kumar Singh, and Klaus McDonald-Maier. ThermalAttackNet: Are CNNs Making it Easy to Perform Temperature Side-channel Attack in Mobile Edge Devices? *Future Internet*, 13(6):146, 2021.

[137] Yi Xiang and Sudeep Pasricha. Soft and Hard Reliability-aware Scheduling for Multicore Embedded Systems with Energy Harvesting. *IEEE Transactions on Multi-Scale Computing Systems*, 1(4):220–235, 2015.

[138] Vipin Kumar Kukkala, Sudeep Pasricha, and Thomas Bradley. SEDAN: Security-Aware Design of Time-Critical Automotive Networks. *IEEE Transactions on Vehicular Technology*, 69(8):9017–9030, 2020.

[139] Tajana Simunic Rosing, Kresimir Mihic, and Giovanni De Micheli. Power and Reliability Management of SoCs. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, 15(4):391–403, 2007.

[140] Lin Huang, Feng Yuan, and Qiang Xu. Lifetime Reliability-aware Task Allocation and Scheduling for MPSoC Platforms. In *Design, Automation & Test in Europe Conference & Exhibition*, pages 51–56, 2009.

[141] Jia Huang, Jan Olaf Blech, Andreas Raabe, Christian Buckl, and Alois Knoll. Analysis and Optimization of Fault-tolerant Task Scheduling on Multiprocessor Embedded Systems. In *Proceedings of the International Conference on Hardware/Software Codesign and System Synthesis*, pages 247–256, 2011.

[142] Laura A Rozo Duque, Jose M Monsalve Diaz, and Chengmo Yang. Improving MPSoC Reliability through Adapting Runtime Task Schedule based on Time-correlated Fault Behavior. In *Design, Automation & Test in Europe Conference & Exhibition (DATE)*, pages 818–823, 2015.

[143] Arturo Pérez, Alfonso Rodríguez, Andrés Otero, David González Arjona, Alvaro Jiménez-Peralo, Miguel Ángel Verdugo, and Eduardo De La Torre. Run-time Reconfigurable MPSoC-based On-board Processor for Vision-based Space Navigation. *IEEE Access*, 8:59891–59905, 2020.

[144] Farshad Firouzi, Ali Azarpeyvand, Mostafa E Salehi, and Sied Mehdi Fakhraie. Adaptive Fault-tolerant DVFS with Dynamic Online AVF Prediction. *Microelectronics Reliability*, 52(6):1197–1208, 2012.

[145] Saeed Parsa and Reza Entezari-Maleki. RASA: A New Grid Task Scheduling Algorithm. *International Journal of Digital Content Technology and its Applications*, 3(4):91–99, 2009.

[146] Sehrish Malik, Shabir Ahmad, Israr Ullah, Dong Hwan Park, and DoHyeun Kim. An Adaptive Emergency First Intelligent Scheduling Algorithm for Efficient Task Management and Scheduling in Hybrid of Hard Real-time and Soft Real-time Embedded IoT Systems. *Sustainability*, 11(8):2192, 2019.

[147] A Alper Goksoy, Anish Krishnakumar, Md Sahil Hassan, Allen J Farcas, Ali Akoglu, Radu Marculescu, and Umit Y Ogras. DAS: Dynamic Adaptive

Scheduling for Energy-Efficient Heterogeneous SoCs. *IEEE Embedded Systems Letters*, 2021.

[148] Zhe Lin, Wei Zhang, and Sinha Sharad. Decision Tree based Hardware Power Monitoring for Run Time Dynamic Power Management in FPGA. In *27th International Conference on Field Programmable Logic and Applications*, pages 1–8, 2017.

[149] Dongsheng Che, Qi Liu, Khaled Rasheed, and Xiuping Tao. Decision Tree and Ensemble Learning Algorithms with their Applications in Bioinformatics. *Software Tools and Algorithms for Biological Systems*, pages 191–199, 2011.

[150] Alberto Suárez and James F Lutsko. Globally Optimal Fuzzy Decision Trees for Classification and Regression. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 21(12):1297–1311, 1999.

[151] Peter Kontschieder, Madalina Fiterau, Antonio Criminisi, and Samuel Rota Bulo. Deep Neural Decision Forests. In *Proceedings of the IEEE International Conference on Computer Vision*, pages 1467–1475, 2015.

[152] Tianqi Chen, Tong He, Michael Benesty, Vadim Khotilovich, Yuan Tang, Hyunsu Cho, Kailong Chen, et al. XGBoost: Extreme Gradient Boosting. *R package version 0.4-2*, 1(4):1–4, 2015.

[153] Xianwei Gao, Chun Shan, Changzhen Hu, Zequn Niu, and Zhen Liu. An Adaptive Ensemble Machine Learning Model for Intrusion Detection. *IEEE Access*, 7:82512–82521, 2019.

[154] Juan José Rodriguez, Ludmila I Kuncheva, and Carlos J Alonso. Rotation Forest: A New Classifier Ensemble Method. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 28(10):1619–1630, 2006.

[155] Chaitanya Manapragada, Geoffrey I Webb, and Mahsa Salehi. Extremely Fast Decision Tree. In *Proceedings of the SIGKDD International Conference on Knowledge Discovery & Data Mining*, pages 1953–1962, 2018.

[156] Jiang Su and Harry Zhang. A Fast Decision Tree Learning Algorithm. In *AAAI*, volume 6, pages 500–505, 2006.

[157] Shaoshan Liu, Jie Tang, Zhe Zhang, and Jean-Luc Gaudiot. Computer Architectures for Autonomous Driving. *Computer*, 50(8):18–25, 2017.

[158] Jason Cong, Mohammad Ali Ghodrat, Michael Gill, Beayna Grigorian, and Glenn Reinman. Architecture Support for Domain-Specific Accelerator-Rich CMPs. *ACM Transactions on Embedded Computing Systems (TECS)*, 13(4s):131, 2014.

[159] Nathan Binkert, Bradford Beckmann, Gabriel Black, Steven K Reinhardt, Ali Saidi, Arkaprava Basu, Joel Hestness, Derek R Hower, Tushar Krishna, Somayeh Sardashti, et al. The gem5 simulator. *ACM SIGARCH Computer Architecture News*, 39(2):1–7, 2011.

[160] Andreas Sandberg, Nikos Nikoleris, Trevor E Carlson, Erik Hagersten, Stefanos Kaxiras, and David Black-Schaffer. Full Speed Ahead: Detailed Architectural Simulation at Near-native Speed. In *IEEE International Symposium on Workload Characterization*, pages 183–192, 2015.

[161] Michael Pellauer, Michael Adler, Michel Kinsy, Angshuman Parashar, and Joel Emer. HAsim: FPGA-based High-detail Multicore Simulation Using Time-division Multiplexing. In *IEEE International Symposium on High Performance Computer Architecture*, pages 406–417, 2011.

[162] Zynq ZCU102 Evaluation Kit. `https://www.xilinx.com/products/boards-and-kits/ek-u1-zcu102-g.html`. [Online; last accessed 01-Dec-2022.].

[163] Sumit K Mandal, Raid Ayoub, Michael Kishinevsky, and Umit Y Ogras. Analytical Performance Models for NoCs with Multiple Priority Traffic Classes. *ACM Transactions on Embedded Computing Systems (TECS)*, 18(5s):52, 2019.

[164] CFS Scheduler. `https://www.kernel.org/doc/html/latest/scheduler/sched-design-CFS.html`. [Online; last accessed 01-Dec-2022.].

[165] Sumit K Mandal, Anish Krishnakumar, Raid Ayoub, Michael Kishinevsky, and Umit 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.

[166] Paul Rosenfeld, Elliott Cooper-Balis, and Bruce Jacob. DRAMSim2: A Cycle Accurate Memory System Simulator. *IEEE Computer Architecture Letters*, 10(1):16–19, 2011.

[167] Tracy D Braun, Howard Jay Siegel, Noah Beck, Ladislau L Bölöni, Muthucumaru Maheswaran, Albert I Reuther, James P Robertson, Mitchell D Theys, Bin Yao, Debra Hensgen, et al. A Comparison of Eleven Static Heuristics for Mapping a Class of Independent Tasks onto Heterogeneous Distributed Computing Systems. *Journal of Parallel and Distributed computing*, 61(6):810–837, 2001.

[168] James Blythe, Sonal Jain, Ewa Deelman, Yolanda Gil, Karan Vahi, Anirban Mandal, and Ken Kennedy. Task Scheduling Strategies for Workflow-based Applications in Grids. In *Proceedings of the IEEE International Symposium on Cluster Computing and the Grid*, volume 2, pages 759–767, 2005.

[169] IBM ILOG CPLEX. IBM ILOG CPLEX Optimization Studio V12. 8. *Language User's Manual*, 2017.

[170] Xiaowen Chen, Yuanwu Lei, Zhonghai Lu, and Shuming Chen. A Variable-Size FFT Hardware Accelerator based on Matrix Transposition. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, 26(10):1953–1966, 2018.

[171] John D Tobola and James E Stine. Low-Area Memoryless optimized Soft-Decision Viterbi Decoder with Dedicated Parallel Squaring Architecture. In *Asilomar Conference on Signals, Systems, and Computers*, pages 203–207. IEEE, 2018.

[172] Samet E Arda, Anish NK, A Alper Goksoy, Joshua Mack, Nirmal Kumbhare, Anderson L Sartor, Ali Akoglu, Radu Marculescu, and Umit Y Ogras. A Simulation Framework for Domain-Specific System-on-Chips: Work-in-Progress. In *Proceedings of the International Conference on Hardware/Software Codesign and System Synthesis Companion*, pages 1–2, 2019.

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

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

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

[176] Francesca Rossi, Peter Van Beek, and Toby Walsh. *Handbook of Constraint Programming*. Elsevier, 2006.

[177] Ashwin M Aji, Antonio J Peña, Pavan Balaji, and Wu-chun Feng. MultiCL: Enabling Automatic Scheduling for Task-Parallel Workloads in OpenCL. *Parallel Computing*, 58:37–55, 2016.

[178] Jing-Jang Hwang, Yuan-Chieh Chow, Frank D Anger, and Chung-Yee Lee. Scheduling Precedence Graphs in Systems with Interprocessor Communication Times. *SIAM Journal on Computing*, 18(2):244–257, 1989.

[179] Richard S Sutton, David A McAllester, Satinder P Singh, and Yishay Mansour. Policy Gradient Methods for Reinforcement Learning with Function Approximation. In *Advances in Neural Information Processing Systems*, pages 1057–1063, 2000.

[180] Anish Nallamur Krishnakumar. Design and Run-Time Resource Management of Domain-Specific Systems-on-Chip (DSSoCs). Technical report, University of Wisconsin-Madison, 2022.

[181] Yuandou Wang, Hang Liu, Wanbo Zheng, Yunni Xia, Yawen Li, Peng Chen, Kunyin Guo, and Hong Xie. Multi-Objective Workflow Scheduling with Deep-Q-Network-based Multi-Agent Reinforcement Learning. *IEEE Access*, 7:39974–39982, 2019.

[182] Stéphane Ross, Geoffrey Gordon, and Drew Bagnell. A Reduction of Imitation Learning and Structured Prediction To No-Regret Online Learning. In *Proceedings of the International Conference on Artificial Intelligence and Statistics*, pages 627–635, 2011.

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

[184] Changjiu Xian, Yung-Hsiang Lu, and Zhiyuan 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.

[185] Rastislav Struharik. Decision Tree Ensemble Hardware Accelerators for Embedded Applications. In *IEEE International Symposium on Intelligent Systems and Informatics (SISY)*, pages 101–106, 2015.

[186] Xubin Tan, Jaume Bosch, Daniel Jiménez-González, Carlos Álvarez-Martínez, Eduard Ayguadé, and Mateo Valero. Performance Analysis of a Hardware Accelerator of Dependence Management for Task-based Dataflow Programming Models. In *IEEE International Symposium on Performance Analysis of Systems and Software*, pages 225–234, 2016.

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

[188] Darryl Lin, Sachin Talathi, and Sreekanth Annapureddy. Fixed Point Quantization of Deep Convolutional Networks. In *International Conference on Machine Learning*, pages 2849–2858, 2016.

[189] Gaurav Mitra, Beau Johnston, Alistair P Rendell, Eric McCreath, and Jun Zhou. Use of SIMD Vector Operations to Accelerate Application Code Performance on Low-powered ARM and Intel Platforms. In *IEEE International Symposium on Parallel & Distributed Processing, Workshops and PhD Forum*, pages 1107–1116, 2013.

[190] Joshua Mack, Sahil Hassan, Nirmal Kumbhare, Miguel Castro Gonzalez, and Ali Akoglu. CEDR-A Compiler-integrated, Extensible DSSoC Runtime. *ACM Transactions on Embedded Computing Systems (TECS)*, 2022.

[191] Vivy Suhendra, Chandrashekar Raghavan, and Tulika Mitra. Integrated Scratchpad Memory Optimization and Task Scheduling for MPSoC Architectures. In *Proceedings of the International Conference on Compilers, Architecture and Synthesis for Embedded Systems*, pages 401–410, 2006.

[192] Joshua Mack, Samet Arda, Umit Y Ogras, and Ali Akoglu. Performant, Multi-objective Scheduling of Highly Interleaved Task Graphs on Heterogeneous System on Chip Devices. *IEEE Transactions on Parallel and Distributed Systems*, 2021.

[193] Robert Khasanov, Julian Robledo, Christian Menard, Andrés Goens, and Jeronimo 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.

[194] Guolin Ke, Qi Meng, Thomas Finley, Taifeng Wang, Wei Chen, Weidong Ma, Qiwei Ye, and Tie-Yan Liu. LightGBM: A Highly Efficient Gradient Boosting Decision Tree. *Advances in Neural Information Processing Systems*, 30, 2017.

[195] Yusen Zhan, Haitham Bou Ammar, and Matthew E Taylor. Scalable Lifelong Reinforcement Learning. *Pattern Recognition*, 72:407–418, 2017.

[196] Miao He, Junshan Zhang, and Vijay Vittal. Robust Online Dynamic Security Assessment using Adaptive Ensemble Decision-Tree Learning. *IEEE Transactions on Power systems*, 28(4):4089–4098, 2013.

[197] Trevor Hastie, Robert Tibshirani, Jerome H Friedman, and Jerome H Friedman. *The Elements of Statistical Learning: Data Mining, Inference, and Prediction*, volume 2. Springer, 2009.

[198] Harsh H Patel and Purvi Prajapati. Study and Analysis of Decision Tree based Classification Algorithms. *International Journal of Computer Sciences and Engineering*, 6(10):74–78, 2018.

[199] Laura Elena Raileanu and Kilian Stoffel. Theoretical Comparison between the Gini Index and Information Gain Criteria. *Annals of Mathematics and Artificial Intelligence*, 41(1):77–93, 2004.

[200] Preeti Kumari, Junil Choi, Nuria González-Prelcic, and Robert W Heath. IEEE 802.11 Ad-based Radar: An Approach to Joint Vehicular Communication-Radar System. *IEEE Transactions on Vehicular Technology*, 67(4):3012–3027, 2017.

[201] John Schulman, Filip Wolski, Prafulla Dhariwal, Alec Radford, and Oleg Klimov. Proximal Policy Optimization Algorithms. *arXiv preprint arXiv:1707.06347*, 2017.

[202] River Library for Hoeffding Tree Classifiers. `https://riverml.xyz/dev/api/tree/HoeffdingTreeClassifier`. [Online; last accessed 01-Dec-2022.].

[203] Zhao Han, Keerthikumara Devarajegowda, Andreas Neumeier, and Wolfgang Ecker. IP-Coding Style Variants in a Multi-layer Generator Framework. In *Design and Verification Conference and Exhibition (DVCon)*, 2020.

[204] Jialiang Zhang, Yue Zha, Nicholas Beckwith, Bangya Liu, and Jing Li. MEG: A RISCV-based System Emulation Infrastructure for Near-data Processing Using FPGAs and High-bandwidth Memory. *ACM Transactions on Reconfigurable Technology and Systems (TRETS)*, 13(4):1–24, 2020.

[205] Vikas Kumar, Mithun Mukherjee, and Jaime Lloret. Reconfigurable Architecture of UFMC Transmitter for 5G and Its FPGA Prototype. *IEEE Systems Journal*, 14(1):28–38, 2019.

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

[207] Sumit K Mandal, Anish Krishnakumar, and Umit Y Ogras. Energy-Efficient Networks-on-Chip Architectures: Design and Run-Time Optimization. *Network-on-Chip Security and Privacy*, page 55, 2021.

[208] Arm Reference Platform for Corstone-700 Subsystem. "`https://git.linaro.org/landing-teams/working/arm/arm-reference-platforms.git/tag/?h=CORSTONE-700-2020.12.10`". [Online; last accessed 01-Dec-2022.].

[209] Bryan Donyanavard, Tiago Mück, Amir M Rahmani, Nikil Dutt, Armin Sadighi, Florian Maurer, and Andreas Herkersdorf. SOSA: Self-Optimizing Learning with Self-Adaptive Control for Hierarchical System-on-Chip Management. In *Proceedings of the Annual IEEE/ACM International Symposium on Microarchitecture*, pages 685–698, 2019.

[210] Biswadip Maity, Bryan Donyanavard, Anmol Surhonne, Amir Rahmani, Andreas Herkersdorf, and Nikil Dutt. SEAMS: Self-Optimizing Runtime Manager for Approximate Memory Hierarchies. *ACM Transactions on Embedded Computing Systems*, 20(5):48:1–48:26, July 2021.