UNIVERSIDADE FEDERAL DO RIO GRANDE DO SUL
INSTITUTO DE INFORMÁTICA
PROGRAMA DE PÓS-GRADUAÇÃO EM COMPUTAÇÃO

MICHAEL GUILHERME JORDAN

Resource Provisioning Framework for CPU-FPGA Environments with Adaptive and Synergistic HLS-Versioning and DVFS

Thesis presented in partial fulfillment of the requirements for the degree of Doctor of Computer Science

Advisor: Prof. Dr. Antonio Carlos S. Beck
Coadvisor: Prof. Dr. Mateus Beck Rutzig

Porto Alegre
March 2023

CIP — CATALOGING-IN-PUBLICATION

Jordan, Michael Guilherme

Resource Provisioning Framework for
CPU-FPGA Environments with Adaptive
and Synergistic HLS-Versioning and DVFS / Michael Guilherme
Jordan. – Porto Alegre: PPGC da UFRGS, 2023.

150 f.: il.

Thesis (Ph.D.) – Universidade Federal do Rio Grande do Sul.
Programa de Pós-Graduação em Computação, Porto Alegre, BR–
RS, 2023. Advisor: Antonio Carlos S. Beck; Coadvisor: Mateus
Beck Rutzig.

1. Collaborative Execution. 2. CPU-FPGA Environments.
3. Cloud Computing. 4. Energy Efficiency. 5. HLS. I. Beck,
Antonio Carlos S.. II. Rutzig, Mateus Beck. III. Título.

“Success is not final, failure is not fatal: It is the courage to continue that counts.”

— WINSTON CHURCHILL

ACKNOWLEDGEMENTS

I would like to express my sincere appreciation to my parents, Silvia Inês Hoffmann Jordan and José Inácio Jordan, who have been my unwavering source of support and inspiration throughout my academic journey. Their love, guidance, and sacrifices have made it possible for me to pursue my academic dreams. I am forever indebted to them for their tireless efforts and unwavering belief in me. To my girlfriend, Tamires Dolores Santos Pereira, I cannot thank you enough for your unconditional love and support. You have been my pillar of strength, my sounding board, and my biggest cheerleader. Your presence in my life has made this journey all the more enjoyable and fulfilling.

I would also like to extend my deepest gratitude to my thesis advisors Professor Antonio Carlos Schneider Beck Filho and Professor Mateus Beck Rutzig, for their guidance, patience, and invaluable insights throughout this research project. Their expertise and support were critical to the successful completion of this thesis. I am particularly grateful for the encouragement they provided during challenging moments of the research process. Their empathy and kindness were a constant source of comfort and motivation, and I cannot thank them enough for going above and beyond their duties as advisors.

To Guilherme dos Santos Korol, Rafael Fão de Moura, Tiago Knorst, and all my colleagues who have provided me with invaluable assistance and support throughout my research journey. Their expertise, insights, and constructive criticism have played a crucial role in shaping my research project. I feel fortunate to have had the opportunity to work alongside such talented and dedicated individuals, and I am grateful for their camaraderie and friendship. I would like to acknowledge my friends for their constant encouragement, humor, and support. Their unwavering support and camaraderie have been a source of inspiration and motivation. Finally, thank you all for your contributions to this project and for your unwavering support throughout this journey.

ABSTRACT

Cloud companies have been exploiting CPU-FPGA collaborative environments to accelerate multi-tenant task requests with scalability and maximize resource utilization. In this scope, tasks may be dispatched to CPU and FPGA concurrently in a scenario with highly variant workloads and target architectures. The main challenge is having a well-balanced resource provisioning that handles these heterogeneous workloads efficiently. In addition to smart provisioning, both architectures offer particular optimization techniques to leverage execution benefits. On the CPU side, the Dynamic Voltage and Frequency Scaling (DVFS) technique is a complementary alternative to boost energy savings. On the FPGA side, High-Level Synthesis (HLS) offers a simple exploration of hardware optimizations through code annotations, resulting in multiple design versions with the same functionality, each with variant latency, power, and area. We call this property *HLS-Versioning*, which opens up space to explore designs optimized for specific warehouse status. Despite the widespread use of DVFS and HLS-Versioning, these have never been synergistically exploited to improve resource provisioning advantages. For that, this thesis proposes RAHD, a framework that bridges the gap between these techniques to achieve maximum performance and energy savings in CPU-FPGA Cloud. RAHD uses HLS-Versioning to select optimized task designs to cover clients' requests at runtime (i.e., either for performance or energy optimization). Then, it adopts an arbiter that automatically selects the most suitable provisioning strategy to distribute the tasks based on the workload/architecture properties. Finally, it uses DVFS without affecting the workload's makespan. All the optimizations are employed in an adaptive fashion and are end-user transparent (i.e., no intervention by the end-user is required). Our experiments show that RAHD outperforms a standard provisioning strategy, delivering, on average, 15.11x performance and 50.05x energy improvements. Compared to an Oracle that always selects the best provisioning strategies, RAHD shows, at most, 4% degradation in performance and 7% in energy.

Keywords: Collaborative Execution. CPU-FPGA Environments. Cloud Computing. Energy Efficiency. HLS.

Framework de provisionamento de recursos para ambientes CPU-FPGA com uso adaptativo e sinérgico de HLS-Versioning e DVFS.

RESUMO

Empresas da Nuvem têm explorado ambientes colaborativos CPU-FPGA para acelerar solicitações de tarefas de vários inquilinos com escalabilidade e maximizar a utilização de recursos. Nesse escopo, tarefas podem ser despachadas para a CPU e FPGA simultaneamente em um cenário com cargas de trabalho e arquiteturas alvo altamente heterogêneas. Diante disso, o principal desafio é ter um provisionamento de recursos bem equilibrado que lide eficientemente com essas cargas de trabalho heterogêneas. Além do provisionamento inteligente, ambas as arquiteturas oferecem técnicas de otimização específicas para aproveitar os benefícios de execução. Do lado da CPU, a técnica de Escalonamento Dinâmico de Voltagem e Frequência (DVFS do Inglês *Dynamic Voltage and Frequency Scaling*) é uma alternativa complementar para impulsionar a economia de energia. Do lado da FPGA, a Síntese de Alto Nível (HLS do Inglês *High-Level Synthesis*) oferece uma exploração simples de otimizações de hardware por meio de anotações de código, resultando em várias versões de design com a mesma funcionalidade, cada uma com latência, consumo de energia e área variantes. Chamamos essa propriedade de *HLS-Versioning*, que abre espaço para explorar designs otimizados para estados específicos da Nuvem. Apesar do uso generalizado de DVFS e HLS-Versioning, estes nunca foram explorados de forma sinérgica para melhorar as vantagens do provisionamento de recursos. Para isso, esta tese propõe o RAHD, um *framework* que une estas técnicas para alcançar o máximo desempenho e economia de energia em ambientes CPU-FPGA da Nuvem. O RAHD usa o HLS-Versioning para selecionar designs de tarefas otimizados para atender as solicitações dos clientes em tempo de execução (ou seja, para otimização de desempenho ou energia). Em seguida, adota um árbitro que seleciona automaticamente a estratégia de provisionamento mais adequada para distribuir as tarefas com base nas propriedades de carga de trabalho/arquitetura. Por fim, usa o DVFS sem afetar o tempo de conclusão da carga de trabalho. Todas as otimizações são empregadas de forma adaptativa e são transparentes para o usuário final (ou seja, nenhuma intervenção do usuário final é necessária). Nossos experimentos mostram que o RAHD supera uma estratégia de provisionamento padrão, entregando, em média, 15,11 vezes de desempenho e 50,05 vezes de melhorias de energia. Em comparação com um oráculo que sempre seleciona as melhores estratégias de provisionamento, o RAHD mostra, no máximo, uma degradação de 4% no desempenho e 7% na energia.

LIST OF FIGURES

LIST OF TABLES

LIST OF ABBREVIATIONS AND ACRONYMS

AaaS	Acceleration-as-a-Service
BRAM	Block Random Access Memory
CLB	Configurable Logic Blocks
CMOS	Complementary Metal-oxide-semiconductor
CPU	Central Processing Unit
DPR	Dynamic Partial Reconfiguration
DRAM	Dynamic Random-Access Memory
DSE	Design Space Exploration
DSP	Digital Signal Processor
DVFS	Dynamic Voltage and Frequency Scaling
EDP	Energy-Delay Product
FCFS	First-Come First-Served
FF	Flip-Flop
FPGA	Field-Programmable Gate Array
GMK	Genetic Multidimensional Knapsack
GPU	Graphic Processing Unit
HDL	Hardware Description Language
HLS	High-Level Synthesis
HW	Hardware
ILP	Integer Linear Programming
IP	Intellectual Property
LUT	Look-up Table
MTFR	Multi-Task Full Reconfiguration
OS	Operating System

PCIe	Peripheral Component Interconnect Express
PE	Processing Element
PRR	Partial Reconfigurable Regions
RLP	Request-Level Parallelism
RR	Round-Robin
SIMD	Single Instruction Multiple Data
SoC	System-on-Chip
SRAM	Static Random-Access Memory
SW	Software
VHDL	Very High Speed Integrated Circuits Hardware Description Language
VM	Virtual Machine
WRR	Weighted Round-Robin

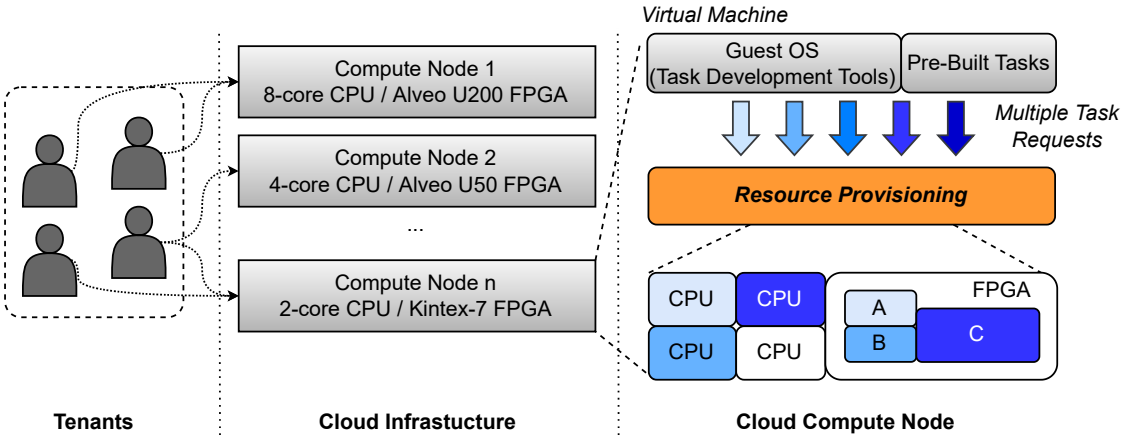
CONTENTS

1 INTRODUCTION

The growing offloading and software-as-a-service demands have increased warehouse infrastructures, pushing up power costs. To satisfy this demand, companies like Microsoft, Amazon, Alibaba, and Huawei (WANG et al., 2020; SKHIRI et al., 2019) are investing heavily in FPGA accelerators due to their ability to achieve high throughput and predictable latency into multiple application domains while providing easy programmability through High-Level Synthesis. Many types of workloads, e.g., neural networks, big data analytics, and high-performance computing, can be and have been accelerated by FPGAs. These accelerators are usually deployed alongside CPU devices, further expanding software optimization possibilities through collaborative computing, which allows both the CPU and FPGA to work together to perform tasks, leveraging the strengths of each device to achieve efficient execution.

In modern collaborative Cloud infrastructures, multi-tenancy has been employed to optimize resource usage, where several clients (tenants) share the infrastructure resources (CHEN et al., 2014; VAISHNAV; PHAM; KOCH, 2018; WANG et al., 2020; MAJUMDER et al., 2021). Therefore, the highly variant workload requires simultaneous optimization of multiple tasks from different applications, fully exploiting Request-Level Parallelism (RLP). Figure 1.1 depicts a multi-tenant Cloud environment where tenants request a CPU-FPGA node. Each tenant has access to their own Virtual Machine, which allows them to build and execute tasks by using the tools provided within the guest Operating System. Additionally, tenants may also request pre-built tasks that are made available by the Cloud. This scenario results in multiple heterogeneous tasks being requested

Figure 1.1: Resource provisioning in a multi-tenant Cloud environment.



Source: The Author.

for execution. To ensure efficient use of resources, a resource provisioning method is implemented to distribute these task requests among CPU and FPGA devices.

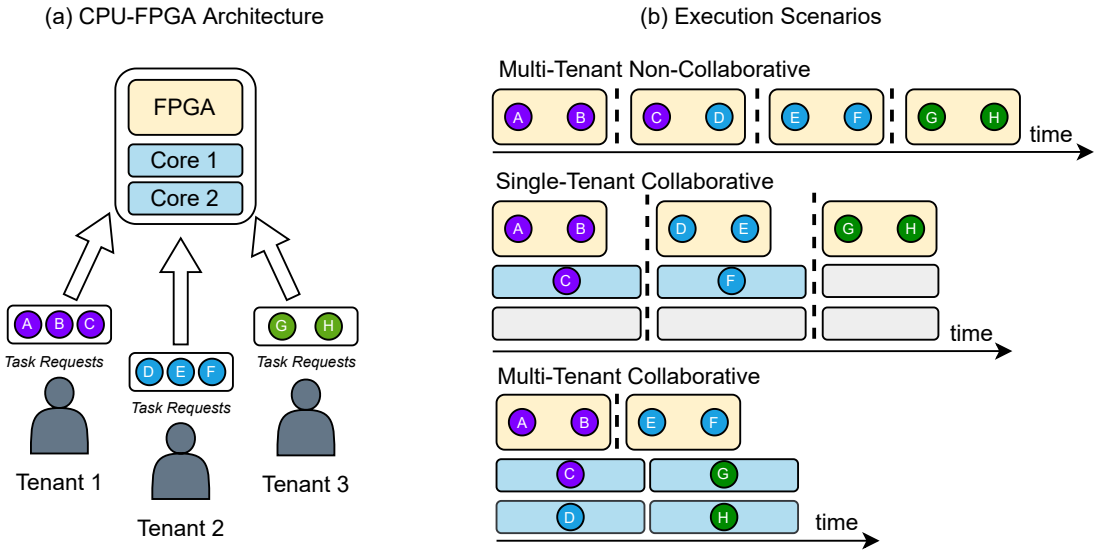
In this context, the main challenge is having a well-balanced resource provisioning that explores the optimization capabilities offered by both architectures to optimize performance and energy consumption. On the CPU side, power management techniques, like Dynamic Voltage and Frequency Scaling (DVFS), are complementary alternatives to boost energy savings. DVFS exploits the processor idleness (usually due to I/O operations) to decrease, at runtime, operating frequency and supply voltage, being widely present in CPU-only Cloud infrastructures. On the FPGA side, HLS brings the possibility of easily exploring the benefits of different hardware optimizations. Each hardware optimization and their combinations result in different versions of the same design in the same way CPU binaries are generated from the same source code when using particular compiler flags. We call this property *HLS-Versioning*. These versions vary not only in performance and energy but also in terms of resource consumption.

Even though both standalone techniques have been extensively used, they have never been cooperatively exploited to further improve resource provisioning efficiency. Therefore, this thesis proposes a framework that bridges the gap between DVFS, HLS-Versioning, and CPU-FPGA collaborative environments. *Our approach is designed for the Acceleration-as-a-Service Cloud model (SIDIROPOULOS et al., 2018; LI et al., 2018; BACIS; BRONDOLIN; SANTAMBROGIO, 2020; DAMIANI et al., 2022), where clients request the execution of pre-designed tasks over target devices (in our case, CPU-FPGA architectures).* As our object of study is Cloud scenarios, workload and available resources are highly varied. In this way, we consider the premise that our approach must have two essential characteristics: *adaptability* and *end-user transparency*. In our scope, we define **adaptability** as the ability to adapt resource provisioning and CPU/FPGA techniques according to parameters only known at run-time, such as available CPU and FPGA resources, workload, and cloud service. Offline solutions will not suffice in these cases since the analysis must be re-executed once any of these parameters change. We define an approach as **end-user transparent** when the end-user is unaware of the optimizations provided by either resource provisioning or CPU/FPGA techniques. Next, we discuss the optimization opportunities brought by collaborative resource provisioning, HLS-Versioning, and DVFS.

1.1 Resource Provisioning in Multi-Tenant Environments

Resource provisioning plays a vital role in CPU-FPGA environments, as the collaborative distribution of tasks over both devices affects the performance and energy of the system. At the same time, multi-tenancy maximizes the resource utilization of both devices if well performed. The advantages of multi-tenant collaborative environments are illustrated in Figure 1.2, where three tenants request the execution of tasks over a system that comprises an FPGA and a dual-core CPU (Figure 1.2.a). The task colors (i.e., purple, blue, and green) are used to distinguish tasks from different tenants. For instance, tasks with a purple color represent tasks from Tenant 1. These tasks are provisioned considering three distinct execution scenarios depicted in Figure 1.2.b. In the *Multi-Tenant Non-Collaborative* scenario (first timeline), all tasks are assigned to the FPGA. Unlike a CPU execution, the context switch over the FPGA is provided through FPGA reconfigurations (illustrated by the dashed bars), which are more time-consuming. The indistinct allocation of resource-hungry tasks provokes many FPGA reconfigurations, consequently increasing the overall execution time. In the *Single-Tenant Collaborative* scenario (second timeline), requests from different tenants do not execute in parallel (i.e., single-tenancy), but both devices are used for tasks' execution (i.e., collaborative execution). In this scenario, fewer FPGA reconfigurations are necessary since more devices are available. However, the execution of tenants' workloads in a sequential fashion under-utilizes the resources. Finally,

Figure 1.2: Comparison between Single-Tenant, Multi-Tenant, and Multi-Tenant Collaborative scenarios.



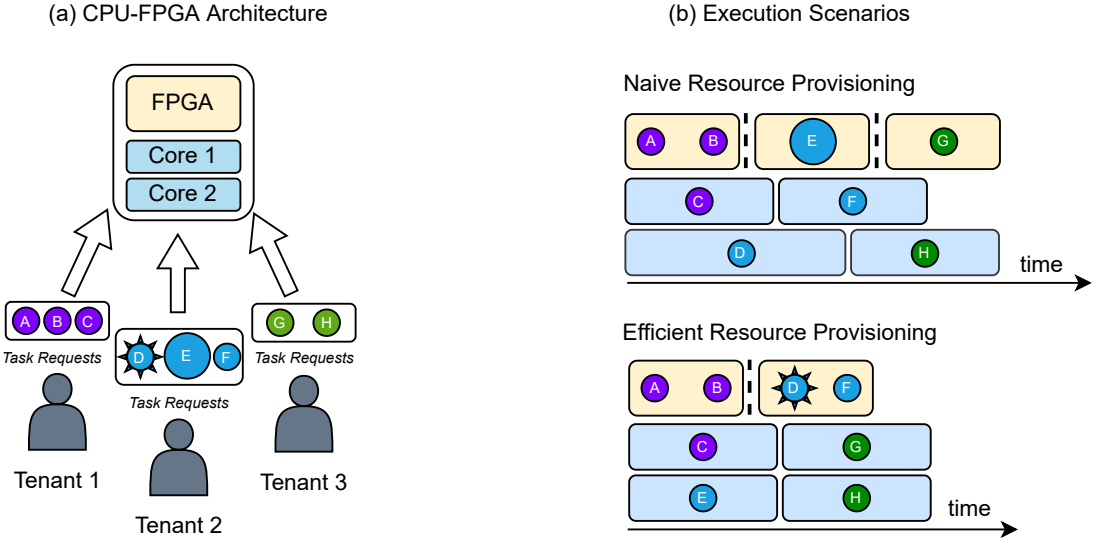
Source: The Author.

the combination of multi-tenancy and collaborative execution (*Multi-tenant Collaborative* scenario - third timeline) enables a better distribution across the CPU and FPGA, providing acceleration of more tasks, maximizing resource utilization, and reducing the overall execution time/energy consumption, which makes this execution model progressively adopted (DAMIANI et al., 2022; NGUYEN; KUMAR, 2020; MAJUMDER et al., 2021; ZHOU et al., 2021).

Although the multi-tenant collaborative approach shows advantages, this provisioning practice is complex. Several works have already shown that the diverse computing requirements and workload characteristics make the resource provisioning on FPGA-only devices an NP-complete problem (NP - non-deterministic polynomial time) (MINHAS et al., 2021). In CPU-FPGA environments, this is further aggravated, as tasks may present variant characteristics when executed on each device.

Figure 1.3 considers three tenants requesting the execution of tasks over the same system from the former example (depicted by Figure 1.3.a). Figure 1.3.b illustrates naive and efficient provisioning scenarios. Larger circles indicate higher FPGA resource consumption, while spiked circles have higher FPGA acceleration. In the *Naive Resource Provisioning*, tasks are assigned following their arriving order to the FPGA (until they fit), and the next tasks are dispatched to be executed in parallel over the CPU. By using this method, a task that demands a high number of FPGA resources was allocated over the FPGA (blue task E), affecting the overall execution time, as no other task could be addressed to the FPGA in parallel, also leading to additional FPGA reconfigurations. At the

Figure 1.3: Comparison between naive and efficient resource provisioning.



Source: The Author

same time, a task with high FPGA acceleration was addressed to the CPU (blue task D), which also increased the overall execution time. On the other hand, the *Efficient Resource Provisioning* considers workload characteristics when distributing tasks, allocating the high acceleration task to the FPGA (blue task D) and the FPGA resource-hungry task to the CPU (blue task E).

The example shows us the importance of efficient provisioning in multi-tenant collaborative environments, which must consider task characteristics on their allocations to achieve high-performance and energy-efficient execution. However, efficient provisioning may vary depending on several parameters, such as - 1) the architecture, given by the model of the FPGA and CPU and their available resources; 2) the intrinsic characteristics of the workload; 3) the demand, given by the number of requests per workload; and, 4) the provisioning strategy processing time (i.e., time to converge). Therefore, the challenge lies in enabling provisioning that dynamically adapts to the subtle change of these variables (JORDAN et al., 2021a).

Even though efficient provisioning advantages are clear, CPU-FPGA execution can be further improved by using optimization techniques offered by both devices. However, some of them may negatively impact the provisioning effectiveness or have their benefits reduced due to provisioning decisions. The next Section discusses two optimization techniques and how these may impact or may be impacted by resource provisioning.

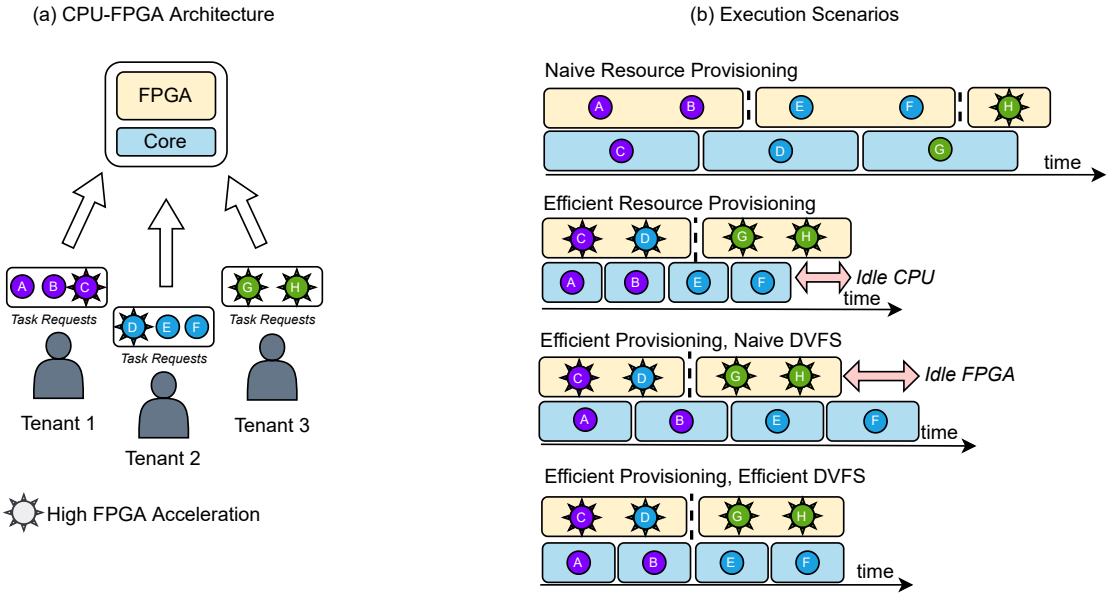
1.2 Exploiting DVFS and HLS Optimizations in CPU-FPGA Environments

This Section discusses the importance of efficient DVFS and HLS-Versioning exploitation in CPU-FPGA environments.

Dynamic Voltage and Frequency Scaling: DVFS has already been proven an effective way of reducing power consumption in CPU-based Cloud. However, if naively employed, it can result in energy/performance degradation. Therefore, we must provide efficient resource provisioning strategies for variant workloads and dynamically adapt the DVFS configurations so that the provisioning solutions are not affected.

Figure 1.4 illustrates the effects of exploiting resource provisioning alone and with the use of DVFS. In the example, three tenants request the execution of tasks with different FPGA acceleration levels - high (spiked circles) or low (circles). For the sake of clarity, let us consider a CPU-FPGA environment composed of an FPGA and a single-core CPU. First, tasks are distributed over CPU and FPGA without DVFS using naive

Figure 1.4: Resource provisioning and power optimization (DVFS) benefits.

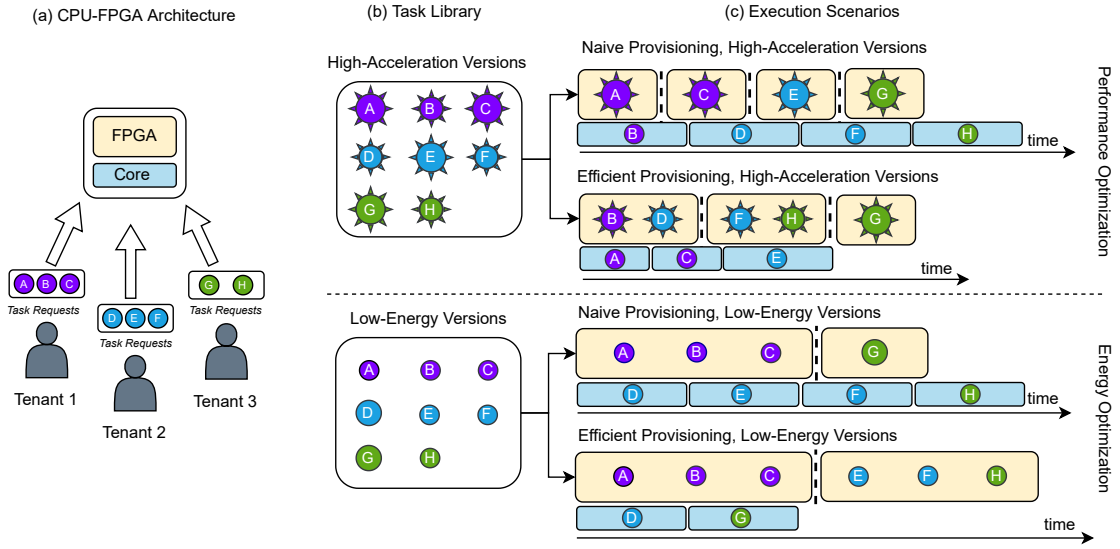


Source: The Author.

and efficient provisioning. The *Naive Resource Provisioning* assigns tasks following their arriving order (the same naive method from the former example). This strategy results in some tasks of low FPGA acceleration being dispatched to the FPGA, increasing execution time. RLP was also affected as one red task was not executed in parallel, generating three FPGA reconfigurations. Conversely, the *Efficient Resource Provisioning* properly allocates the high FPGA acceleration tasks to the FPGA, maximizing performance and RLP. Even though advantageous, efficient resource provisioning still opens up space to DVFS when FPGA and CPU loads are unbalanced, as the CPU keeps idle for some time while the FPGA is still running (as shown in Figure 1.4.b second timeline). In the *Efficient Provisioning, Naive DVFS* scenario, the naive use of DVFS (e.g., reducing voltage and frequency as much as possible) could not balance FPGA and CPU loads, resulting in performance penalties as the CPU execution time was increased. Finally, in the *Efficient Provisioning, Efficient DVFS* scenario, DVFS was employed in a manner that both FPGA and CPU workloads could be balanced, and the overall execution time (i.e., time to finish all tasks) was not affected. Therefore, DVFS must be aware of the cooperative execution to reduce power while not significantly harming the performance.

High-Level Synthesis: High-Level Synthesis has been employed as an effective way of abstracting the complexity of hardware description languages like VHDL and Verilog. By adding code annotations, one can design tasks with variant performance, power, and area. Such HLS-Versioning is usually employed to extract designs optimized

Figure 1.5: Resource provisioning and HLS-Versioning benefits.



Source: The Author.

for maximum performance or energy efficiency. These optimizations usually come at variant area costs, which can limit the RLP offered by the FPGA device. Therefore, to achieve the full potential of these designs, efficient provisioning must be adopted.

Figure 1.5 illustrates the benefits of exploiting HLS-Versioning, where three tenants request the execution of eight tasks (Figure 1.5.a) over an environment composed of an FPGA and a single-core CPU. In Figure 1.5.b, we can observe a Task Library, which comprises variant designs for each available task. In the example, each task has two design versions for FPGA execution, one optimized for performance (spiked circles - *High-Acceleration Versions*) and the other optimized for energy (*Low-Energy Versions*). The larger the symbols, the higher the area costs. For instance, considering the high-acceleration versions, tasks A, B, E, and G consume more resources than B, D, F, and H. These design versions are explored in the execution scenarios present in Figure 1.5.c, which comprises four timelines. The first two timelines use high-acceleration versions to provide a performance-optimized execution (*Performance Optimization*). The third and fourth timelines adopt low-energy versions to provide an energy-optimized execution (*Energy Optimization*). In both optimization scenarios, we show the impact of naive (assigns tasks by arriving order) and efficient provisioning.

Let us start by evaluating the performance optimization execution scenarios (first and second timelines). As seen in the first timeline (*Naive Provisioning, High-Acceleration Versions*), naive provisioning can not extract the potential of high-acceleration task versions. In this scenario, the naive provisioning distributed the tasks with higher area costs

(i.e., A, C, E, G) to the FPGA, hugely reducing RLP - only one task could be assigned to the FPGA at a time. In contrast, the second timeline shows us the combination of efficient provisioning with high acceleration versions (*Efficient Provisioning, High-Acceleration Versions*). By addressing tasks that consume fewer resources for FPGA execution (i.e., B, D, F, H), efficient provisioning could achieve both RLP and acceleration benefits.

We provide a similar example for the energy optimization execution scenarios (third and fourth timelines). In this study, let us consider that the FPGA execution using low-energy versions always consumes less energy than executing the tasks over the CPU. In the third timeline (*Naive Provisioning, Low-Energy Versions*), we observe that the naive provisioning ended up addressing several tasks to the CPU, not exploiting the benefits of low-energy versions over the FPGA. For instance, a high resource consumption task (task G) was addressed to the FPGA, preempting the availability of the device for tasks F and H, which consequently were executed over the CPU. In the fourth timeline (*Efficient Provisioning, Low-Energy Versions*), the efficient provisioning, despite presenting a longer execution time than the scenario from the previous timeline, assigned more low-energy tasks to the FPGA, resulting in an energy-efficient allocation. Therefore, when the provisioning targets energy optimization, avoiding allocating tasks over the power-hungry device may be necessary, even if it results in performance penalties.

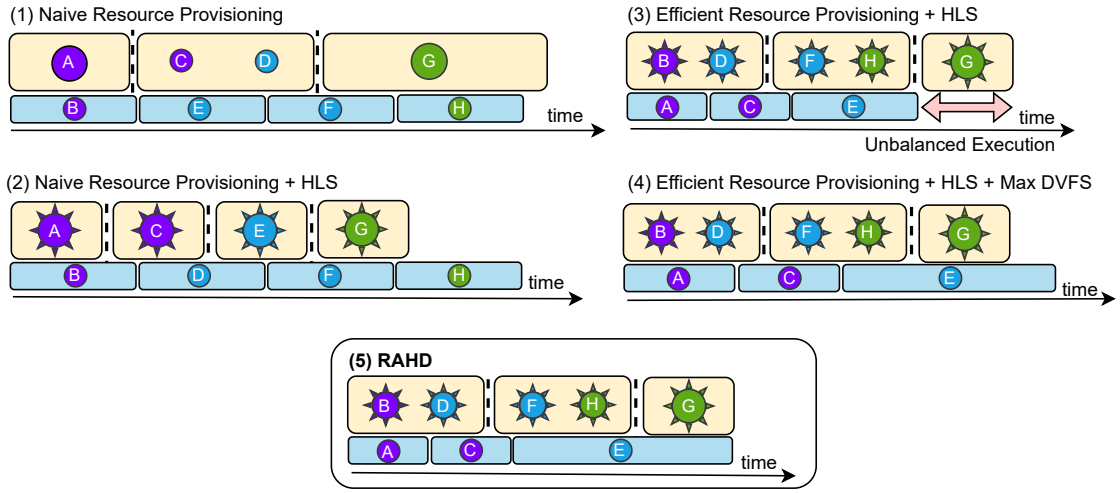
This example shows us that naively assigning specialized tasks will not always yield the best results. Additionally, selecting different task versions changes the properties of the workload, highlighting the importance of provisioning adaptability.

1.3 Thesis Contributions

Many works have already proposed improving the standalone use of CPU-FPGA collaborative computing, and others combine collaborative computing with CPU-only power optimization techniques (e.g., like DVFS) ***OR*** FPGA HLS-Versioning. However, there is a lack of studies that explore the huge design space exploration (DSE) provided by collaborative computing and optimization techniques over CPU-FPGA. To the best of our knowledge, no work exploits all these optimization axes. Therefore, this thesis provides efficient resource provisioning in CPU-FPGA environments by bridging the gap between collaborative computing, DVFS on the CPU, and HLS-Versioning techniques.

For that, this thesis introduces RAHD, a ***R***esource Provisioning Framework for CPU-FPGA Environments with ***A***daptive ***H***LS-Versioning and ***D***VFS. Figure 1.6 demon-

Figure 1.6: Benefits of efficient provisioning, HLS-Versioning, and DVFS in a collaborative execution.



Source: The Author

strates RAHD's advantages. It presents the following execution scenarios - (1) naive provisioning is considered; (2) naive provisioning and HLS are employed; (3) efficient provisioning and HLS are used; (4) efficient provisioning, HLS, and Max DVFS levels are considered; (5) our proposed solution.

In timeline (1), naive provisioning (i.e., which assigns tasks in incoming order) may lead to unprofitable tasks (e.g., that benefit more from CPU execution or resource-hungry tasks) being assigned to the FPGA device. To reduce the overall execution time, one could consider using HLS-Versioning to generate performance-optimized task designs for FPGA execution, as depicted by timeline (2), which incorporates HLS-optimized task designs with naive provisioning. However, even in this scenario, naive provisioning is prohibitive, as it may assign resource-hungry tasks to the FPGA (only one task is executed at a time), overloading the CPU and reducing RLP. In timeline (3), efficient provisioning is used alongside HLS-optimized task designs. By avoiding resource-hungry tasks over the FPGA, this provisioning leads to more tasks being executed in parallel over the FPGA, fully extracting the HLS-Versioning potential. Moreover, tasks that present good performance over the CPU were correctly assigned to the device (tasks A and C), drastically reducing the overall execution time. Still, the provisioning may result in an unbalanced allocation. In timeline 3, we can observe a gap between FPGA and CPU execution times, where the CPU keeps idle while the FPGA executes task G. In this scenario, one could consider using DVFS over the CPU to balance execution times while achieving energy efficiency. Therefore, timeline 4 employs the maximum DVFS levels (i.e., reduces the frequency to minimum levels for maximum power reduction) over the produced al-

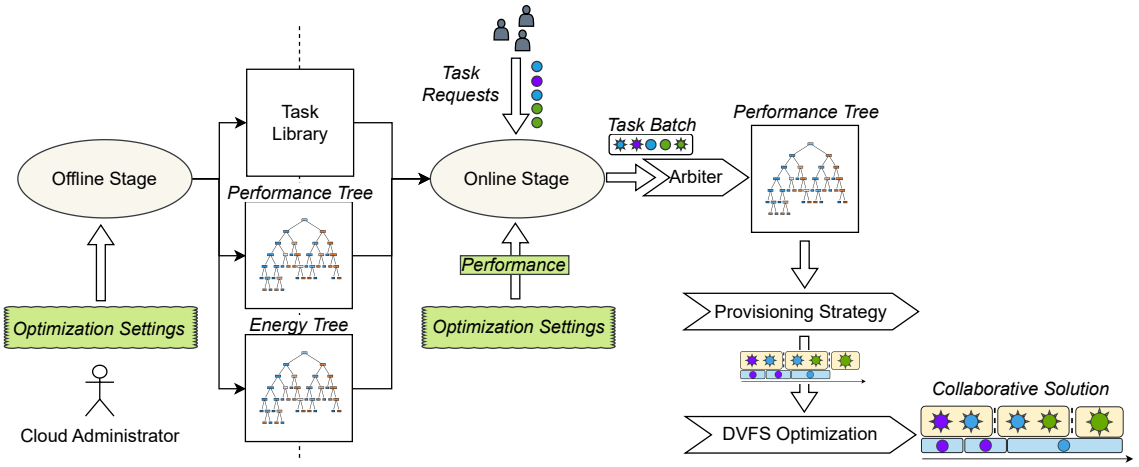
location. However, such a naive employment of DVFS may result in huge performance penalties, diminishing the benefits achieved by the other optimization fronts. Finally, our proposed approach (depicted by timeline (5)) exploits all techniques conversantly without reducing each other's benefits. For that, it uses provisioning methods that maximize HLS-Versioning potential while effectively using DVFS without increasing the overall execution time, achieving performance and energy advantages.

On top of that, *RAHD explores all optimization fronts presented in Figure 1.6 in a feasible time, which is a key challenge in Cloud scope*. For instance, a study provided over Alibaba Cloud workloads shows that 90% of task durations range from dozens of seconds to a few minutes (information taken from an Alibaba Cluster trace (HAN et al., 2022)). To cover workload with variant durations, RAHD comprises several provisioning strategies. These are classified into - short convergence time strategies (i.e., ms to converge), which minimally harm the execution of short-running workloads while still providing good solutions for workloads with specific characteristics; and long convergence time strategies, which cover a wider range of workload behaviors but with a long convergence time (i.e., seconds to converge). In this scope, the challenge lies in selecting the strategy that effectively handles the current workload behavior. To tackle this challenge, RAHD incorporates an arbiter that automatically chooses the most suitable provisioning strategy for incoming workloads by analyzing both the workload and the target architecture properties. The arbiter employs a configurable classification method that allows the selection of strategies that prioritize either performance or energy efficiency.

Figure 1.7 depicts RAHD's workflow, which includes the arbiter's functionality discussed in the previous paragraph. The framework comprises offline and online stages. *At the Offline Stage*, RAHD automatically generates both FPGA and CPU executable files, creating multiple design versions (i.e., HLS-Versioning) for each task to be executed on the FPGA. These designs include both performance- and energy-oriented options. Additionally, RAHD conducts a thorough profiling of each task's energy and performance characteristics when executed over both devices, also detecting their properties across a range of available CPU DVFS levels. All these executable files and tasks' information are then stored in a Task Library.

RAHD leverages the data from the Task Library to generate synthetic workload inputs for training the arbiter's classification logic, which employs decision trees to identify the most appropriate strategy for a given workload at the Online Stage. The framework generates two decision trees, one for detecting high-performance provisioning strategies

Figure 1.7: RAHD Offline and Online stages overview.



Source: The Author

and the other for detecting energy-efficient ones.

At the end of the stage, the administrator adjusts RAHD optimization settings, which offer the option to prioritize performance or energy optimization and can be changed dynamically. These settings affect the framework resource provisioning at the Online Stage, influencing the designs selected to serve the task requests on the FPGA and the arbiter's configuration. In the example, the administrator configured RAHD for performance optimization.

During the Online Stage, RAHD collects task requests (in the form of task IDs) and retrieves their properties by accessing the Task Library. The task properties include characteristics of the tasks when executed over CPU and FPGA, such as latency, power consumption, and resource usage. Regarding FPGA-related task properties, as RAHD is configured for performance optimization, only information regarding performance-oriented designs is collected. The task IDs, along with the aggregated information, are batched. Once the batch is formed, it serves as input to the arbiter, which, in our example, uses the performance tree (i.e., due to the cloud administrator configuration) to select the most suitable provisioning strategy based on the batch properties and target architecture.

Once the strategy is selected, the provisioning process is executed, resulting in a solution that comprises queues of tasks allocated to each device. RAHD then proceeds to perform the DVFS optimization, evaluating the solution in search of optimal DVFS levels that balance FPGA and CPU execution, allowing for energy savings without compromising performance (as shown in the example from Figure 1.4). The final outcome is a Collaborative Solution that includes task execution queues for both devices and additional information, including task IDs (to retrieve the corresponding CPU and FPGA executable

files from the Task Library) and details regarding the selected DVFS levels.

By experimentally evaluating our framework, we show that: 1) different resource provisioning strategies (with variant quality of solution and convergence time trade-offs) are needed depending on the architecture and workload characteristics; 2) HLS-Versioning is mandatory to either maximize performance or energy efficiency, also affecting provisioning strategies' potential, as it changes workload characteristics; 3) DVFS must be synergistically employed to bring energy gains without harming the performance/energy benefits brought by HLS-Versioning and resource provisioning optimization axes; 4) resource provisioning must be fully-adaptive as the properties of incoming workloads change due to the inherent heterogeneity of workloads and the use of HLS-Versioning.

Therefore, only RAHD can fully explore these points and extract the full potential of CPU-FPGA environments by provisioning tasks through a fully-adaptive approach (i.e., provisioning strategy arbiter) that leverages the benefits of HLS-Versioning and further improves energy efficiency through the use of DVFS.

The remainder of this thesis is organized as follows. Chapter 2 provides a background on the main concepts addressed by this thesis. Chapter 3 reviews the literature related to the techniques discussed in this thesis and highlights this thesis's contributions. Chapter 4 provides a detailed description of the proposed framework. Chapter 5 comprises our experiments. Chapter 6 presents the final considerations.

2 BACKGROUND

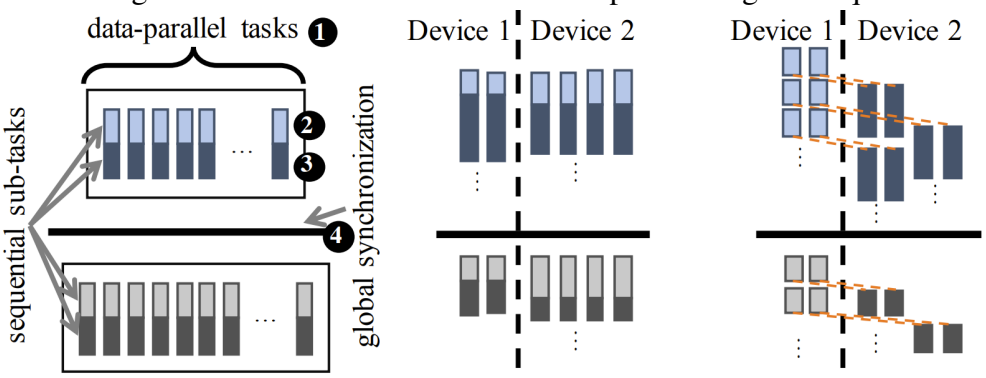
This chapter explains the concepts of this proposal. Section 2.1 presents the main concepts regarding collaborative computing. Section 2.2 highlights concepts regarding FPGA architectures and the exploration of FPGA High-Level Synthesis (HLS). Finally, in Section 2.3, we study the DVFS technique.

2.1 Collaborative Computing

Collaborative computing was proposed to leverage the efficiency of multiple device architectures (e.g., CPU-FPGA, CPU-GPU, and multi-GPU architectures) by orchestrating workload distribution among the diverse computing elements. This distribution can have a significant impact on the system's performance and energy consumption since some workloads may perform better in a specific architecture that can hugely differ in power dissipation. In heterogeneous architectures like the CPU-FPGA environment used in this work, the challenge lies in choosing which device a given workload must be executed according to a given optimization metric (e.g., performance or energy).

Applications benefit from collaborative execution by exploiting different partitioning strategies, such as task and data partitioning. Figure 2.1 illustrates how both partitioning techniques work over a program structure with data-parallel and sequential subtasks (Figure 2.1.a). Data partitioning (Figure 2.1.b) is a collaborative computing technique that balances applications' workload among different architectures by distributing the data to be processed among each computing device. In other words, both devices process the

Figure 2.1: Collaborative data and task partitioning techniques.



(a) Program Structure (b) Data Partitioning (c) Task Partitioning

Source: (HUANG et al., 2019)

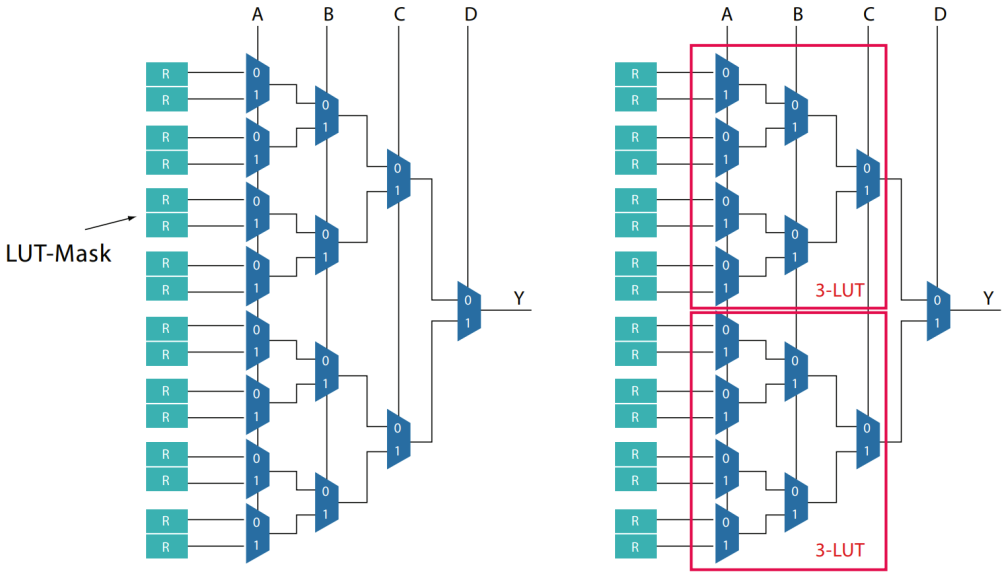
same tasks over different data. The main challenge with data partitioning is determining the optimal partitioning, i.e., the distribution of data-parallel tasks across devices that results in the highest performance. On the other hand, task partitioning (Figure 2.1.c) is a collaborative execution strategy wherein different devices execute different types of sub-tasks on the entire set of the data, i.e., within each data-parallel task, different types of devices perform different types of sub-tasks.

Applications can present different benefits when using the techniques. Let us consider the Random Sample Consensus, an algorithm for estimating parameter models. It takes random input samples and iteratively tests them until a successful model is found. The Random Sample Consensus comprises two stages: the first one is a model-fitting that uses the random samples, while the second one computes outliers and error values to evaluate the model's accuracy. This application can be processed in a data partitioning and task partitioning way. Considering a task partitioning approach, the first stage is executed in the CPU, as the model fitting is inherently sequential. The second stage is addressed to the FPGA, as it can be executed in a massively parallel fashion. By using data partitioning, both devices execute the two stages over a different set of iterations. In the experiments presented in (HUANG et al., 2019), the authors show that task partitioning is more efficient (i.e., shows higher performance) than data partitioning for the Random Sample Consensus application. When both stages are addressed to the FPGA (data-partitioning), the first stage stresses the use of DSP blocks, as it requires intensive floating-point operations, which reduces the data parallelism benefits as fewer compute units are implemented in parallel. On the other hand, with task partitioning, the first task is entirely executed in the CPU, enabling the full availability of the FPGA for the second task, which can better exploit the parallelism provided by the FPGA.

Apart from single applications, task partitioning can also be employed for multiple independent tasks from different applications. In this scenario, the challenge is to balance the workload to ensure that resources are allocated efficiently. This requires a careful analysis of the requirements of each task and application, as well as an understanding of the available resources and their limitations. Additionally, it may be necessary to implement mechanisms for monitoring and adjusting the workload in real time to ensure that the system remains balanced and responsive.

Section Remarks: In this work, we focus on collaboratively provisioning tasks using the task partitioning technique. We focus on this method as in a multi-tenant Cloud, many independent tasks from single or multiple applications are constantly requested for

Figure 2.2: 4-input look-up table.



Source: (ALTERA, 2006)

execution. More specifically, the technique is employed over batches of independent task requests.

2.2 Field Programmable Gate Array (FPGA)

An FPGA is a reconfigurable architecture composed of an array of programmable logic elements that can be configured to execute the logic of an application. The core elements of an FPGA are the lookup tables (LUTs), which are customizable truth tables that generate an output based on the inputs responsible for defining the behavior of the combinatorial logic of the hardware. LUTs are commonly composed of SRAM bits to hold the configuration memory (LUT mask) and a set of multiplexers to select the bit of the LUT mask that must be output. Figure 2.2 shows the implementation of a 4-input LUT (i.e., a LUT that can implement any function of four inputs). This LUT comprises 16 bits of SRAM (LUT mask) and 15 2:1 multiplexers. In fact, a generic k-LUT needs 2^k LUT-mask bits and $2^k - 1$ multiplexers. A 4-LUT can also be seen as two 3-LUTs connected by a 2:1 multiplexer, as is presented in Figure 2.2.

LUTs are parts of bigger units called Configurable Logic Blocks (CLB), which allow the user to implement any logical functionality within the chip. CLBs comprise k-LUTs, Muxes, Flip-Flops, and other logic elements (e.g., Full-Adders). Their precise composition depends on the target FPGA device. Apart from the CLBs, modern FPGAs

also include hard-logic components such as Digital Signal Processors (DSP), I/O controllers (DDR, PCI-E, network, etc.), and Block RAMs (BRAMs). These components implement specialized logic that would take up too many resources if implemented in LUTs.

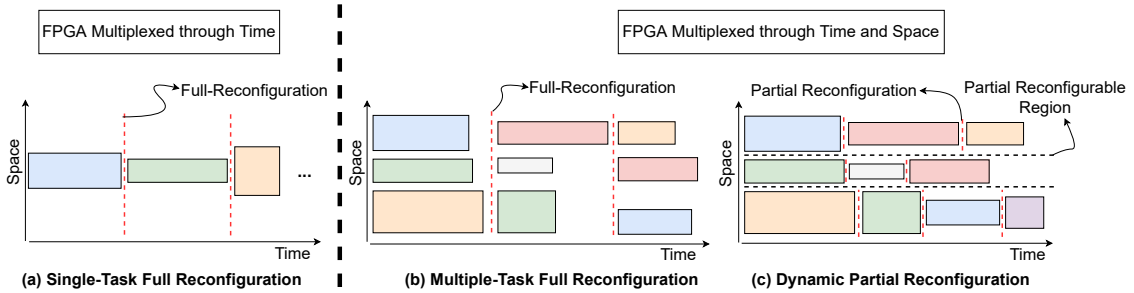
FPGA Synthesis. In FPGA systems, application tasks are designed using synthesis tools. Traditionally, the development of an FPGA design is performed by using Hardware Description Language (HDL) like VHDL and Verilog. In the past decade, High-Level Synthesis (HLS) tools (C/C++/Python) have also become an alternative. The final product of the synthesis is binaries in the format of bitstreams. We define the FPGA implementation of a particular task as a *task design* (also called an accelerator or kernel in the OpenCL programming model). When task designs are loaded to the FPGA, LUTs, and interconnections are configured to reflect the task behavior. Contrary to a standard software compilation, designing and generating bitstreams may demand a long time.

There are five common steps for generating a bitstream: synthesis, mapping, placement, routing, and bitstream generation. First, the hardware description is synthesized into a netlist (a textual description of a circuit made of components). Next, in the mapping step, the netlist functions are mapped to hard logic and LUTs. After that, the placement process selects which of the available instances of each function should be used on the target FPGA. Then, the routing determines the routing resources that must be used so that all functions are correctly connected, and all timing constraints are met. Finally, the FPGA bitstream is produced. In old FPGA devices, the bitstreams were transferred using a serial port, taking seconds for data transferring and loading. Recent FPGA devices use the PCIe connection, which hugely reduced the transference time (32 Gbps for PCIe 3.0, compared to 480Mbps for USB 2.0 used in old FPGA devices).

FPGA Reconfiguration: Allows changing the configuration of part or the entire FPGA fabric. This functionality enables multiple tasks to have their designs multiplexed over time on the same FPGA area.

The reconfiguration time depends on the system setup. Usually, the reconfiguration phase is given by pulling the bitstream from the off-chip memory to the processor's local memory, copying it from the local memory to the FPGA configuration controller, and sending it from the controller to the FPGA configuration memory. As stated by Sadeghi, Razavi and Zamani (2019), one of the largest overheads comes from the bitstream transfer from the external memory to the on-chip memory and from the processor to the configuration controller through the bus. In this way, one of the challenges when

Figure 2.3: FPGA configuring scenarios.



Source: The Author.

executing multiple task designs on FPGA is to reduce the number of reconfigurations. In the next Section, we present the methods for configuring task designs over FPGA devices.

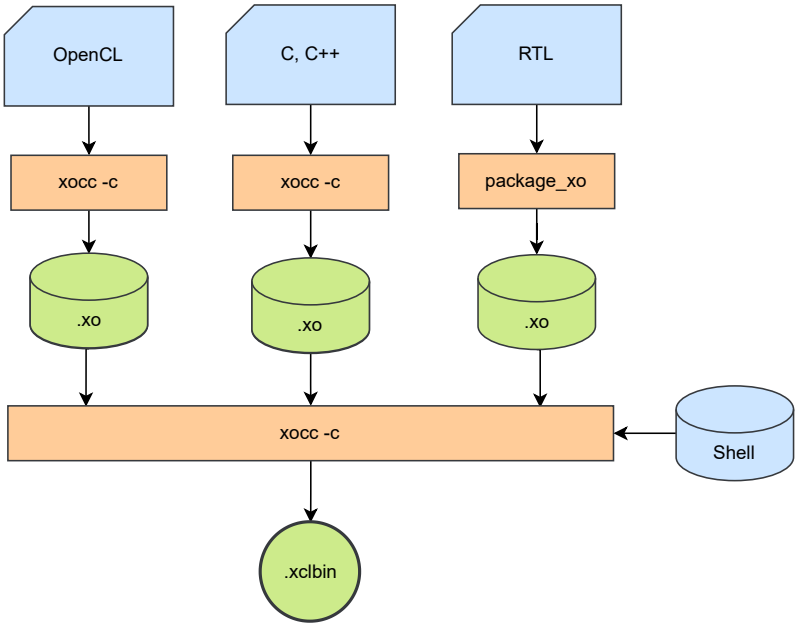
2.2.1 FPGA Multi-Task Configuration

Figure 2.3 illustrates how different FPGA configuring modes work. As can be seen, FPGAs can execute multiple task designs through time and space. In a single-task full reconfiguration, one task is configured at a time, resulting in resource underuse. On the other hand, in a multi-task full reconfiguration, multiple task designs can be addressed to the same configuration, sharing the FPGA space in a parallel manner. However, the whole FPGA must be reconfigured to execute a new set of tasks. In the third model, modern FPGAs support Dynamic Partial Reconfiguration (DPR), enabling only specific FPGA parts to be reconfigured.

Next, we will present more details regarding the *multi-task full reconfiguration technique*, which is this *thesis's target FPGA configuring mode*. We also present more details regarding the Partial Reconfiguration technique, as it is commonly employed in related works.

Multi-Task Full Reconfiguration (MTFR). With the emergent adoption of OpenCL by major FPGA vendors (Xilinx and Altera) and data centers (e.g., Amazon F1), the use of MTFR (i.e., OpenCL programming model) is becoming popular (JUNGBLUT; KRANZLMÜLLER, 2021; MINHAS et al., 2021; WIRBEL, 2014). As previously described, this method allows multiple tasks to be executed in parallel. However, to enable the execution of tasks that are not configured in the FPGA, the whole fabric must be reconfigured, preventing tasks from operating during the reconfiguration. To enable this process, one or multiple task designs must be gathered in FPGA task containers at the static time. Each

Figure 2.4: Task container generation.



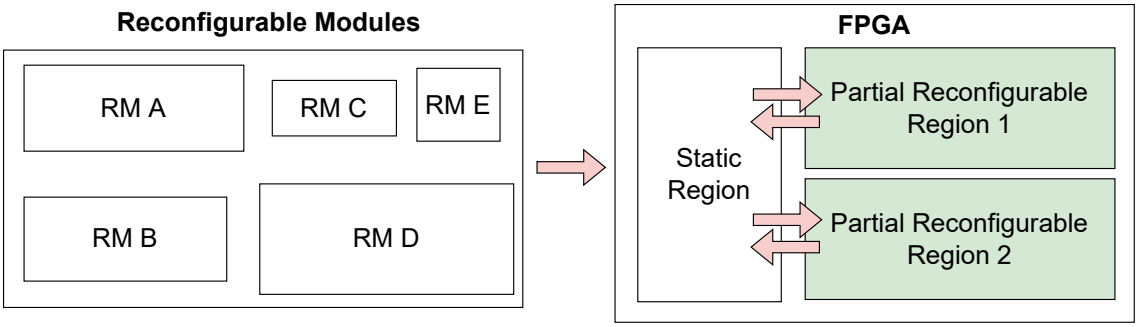
Source: Xilinx SDAccel Manual.

container is synthesized, producing bitstreams that comprise several task designs. At the dynamic time, the pre-generated containers can be loaded over the FPGA, enabling the parallel execution of the tasks present in the container. To allow the execution of tasks not comprised in the loaded container, a new one with the desired tasks must be loaded.

The process of generating an FPGA task container for Xilinx devices is called kernel/task linking and is depicted by Figure 2.4. Each task in an FPGA implementation (i.e., either OpenCL, C/C++, or register-transfer level - RTL descriptions) can be independently compiled to produce a Xilinx object file (.xo) using the Xilinx OpenCL flow. The .xo file is a binary file that contains the task compiled, along with information about the required resources and dependencies. To generate a .xo file, we can use a Xilinx OpenCL compiler, such as xocc, to compile C/C++ or OpenCL code into a .xo file. Alternatively, if we are working with RTL designs, we can use the package_xo utility to compile the design into a .xo file.

Once the individual .xo files for each task were generated, we can use the Xilinx SDAccel shell to link the files together into a single FPGA binary container, known as a .xclbin file. The .xclbin file is a binary file that contains the configuration information required to program the FPGA and implement the tasks described by the .xo files. At runtime, the host code (i.e., the software program that manages the FPGA's execution) is responsible for loading the .xclbin file onto the FPGA and executing the tasks contained within it. *More details concerning the MTFR technique and the OpenCL programming*

Figure 2.5: Partial Reconfigurable Region design.



Source: The Author.

model will be presented in our framework description (Section 4.2.4.1).

Partial Reconfiguration. Partial Reconfiguration (PR) brings extra flexibility to FPGAs by allowing an FPGA design to be loaded in parts of the FPGA without interrupting or compromising the integrity of an application running on other parts that are not being reconfigured. The process of dynamically loading tasks in PRRs is called Dynamic Partial Reconfiguration (DPR). Figure 2.5 illustrates a PR design. A partial reconfiguration design is divided into a static region, which is the portion of the device programmed at the startup and never changes, and one or more PR regions (PRRs), which can be dynamically modified to implement new logic. The static region is usually deployed by the interface logic and the reconfigurable regions controller. As illustrated, multiple reconfigurable modules (implemented task designs) can be dynamically loaded to the partial reconfigurable regions. The partial reconfiguration can be supported by the FPGA configuration controller, which is responsible for performing all the commands to access and modify the configuration memory and managing reconfigurable modules.

The process of identifying, creating, and placing specific hardware structures to achieve high performance within the available FPGA space is called Floorplanning. In this context, the Floorplanning step is responsible for designing the available PRRs. A PRR design flow differs from a standard design flow due to the Floorplan requirement and the application's bitstream generation, which must be specifically designed for the target PRRs. The following steps summarize the processing of a Xilinx PR design (taken from the Xilinx Dynamic Function Exchange (BENDOU, 2020) Manual):

1. Synthesize the static modules and task modules (also named reconfigurable modules) separately;
2. Create physical constraints to define the reconfigurable regions (based on the task modules - floorplan step);

3. Set the Xilinx reconfigurable property (i.e., setting to enable a PRR) on each Partial Reconfigurable Region (PRR);
4. Implement a complete design (static and one Task Module per PRR) in context;
5. Save a design checkpoint for the full routed design;
6. Remove Task Modules from this design and save a static-only design checkpoint;
7. Lock the static placement and routing;
8. Add new Task Designs to the static design and implement this new configuration, saving a checkpoint for the full routed design;
9. Repeat Step 8 until all Task Designs are implemented;
10. Run a verification utility on all configurations;
11. Create bitstreams for each configuration.

Section Remarks: As noticed, the entire PRR design process requires the designer's expertise and manual intervention. Moreover, the decisions made, especially at the floorplanning step, severely impact resource utilization and performance. Due to time constraints, although this technique brings huge flexibility for multiple-task execution, this work embraces only the MTFR model (i.e., OpenCL execution model), producing provisioning solutions for scenarios where the FPGA is fully reconfigured with bitstream containers, which can be composed of one or multiple task designs. Therefore, DPR will be explored in future works.

2.2.2 FPGAs in Multi-Task Environments

The presence of FPGA-based computing in warehouses and datacenters is emerging. Multiple applications must be mapped in this scope to improve scalability and FPGA resource utilization.

FPGAs in the Cloud. Warehouses provide three environments so the clients can exploit the available FPGAs: first, through access to Virtual Machines that support Operating Systems with FPGA development tools, where clients can design and run any task on FPGAs (FPGA-as-a-Service - FPGAaaS); second, through access to already designed tasks (i.e., ready-to-deploy accelerators), where clients require access to pre-synthesized libraries of task designs (Acceleration-as-a-Service - AaaS) (NGUYEN; KUMAR, 2020); third, the combination of both AaaS and FPGAaaS, where the tenant can opt for using already developed tasks or add their own projects to a library (CHEN et al., 2014; BYMA et

al., 2014; FAHMY; VIPIN; SHREEJITH, 2015). For example, companies like InAccel, rENIAC, and Falcon design these libraries of modules to be deployed in warehouse FPGAs. This model is attractive to cloud administrators because it enables easy management of multiple tenants to share the same resources, which improves cloud infrastructures' scalability and economic benefits.

FPGAs Virtualization and Sharing. Virtualization consists of running a virtual instance (e.g., an operating system, software, task) in a layer abstracted from the hardware. The most traditional example is running multiple operating systems on the same computer device simultaneously. In the FPGA virtualization context, it means running virtual instances of accelerators. The Survey proposed in (VAISHNAV; PHAM; KOCH, 2018) categorizes the existing virtualization works on FPGA at the following levels: Resource Level, Node Level, and Multi-node Level:

- Resource Level: It considers the virtualization of reconfigurable (e.g., LUTs and DSPs) and non-reconfigurable FPGA resources (e.g., I/O virtualization). For example, transparent I/O sharing in a multi-tenant system ((KNODEL; GENSSLER; SPALLEK, 2017));
- Node Level: Defined as the virtualization of a single FPGA. In this way, this level covers resource and infrastructure management techniques. Examples include runtime systems, VM management support, and shells to serve/optimize multi-tenant access to accelerators ((CHEN et al., 2014; BYMA et al., 2014; FAHMY; VIPIN; SHREEJITH, 2015; NGUYEN; KUMAR, 2020));
- Multi-node Level: Defined as a cluster of two or more FPGAs. This level covers techniques to optimize the scheduling and communication of tasks among several devices. Examples at this level include Catapult (PUTNAM et al., 2014) and MapReduce (WANG et al., 2016).

FPGA resources can be provided by using space-sharing models (MINHAS et al., 2021; BERTOLINO et al., 2020; STONE; GOHARA; SHI, 2010) and through dynamic partial reconfiguration (CHEN et al., 2014; BYMA et al., 2014; FAHMY; VIPIN; SHREEJITH, 2015; NGUYEN; KUMAR, 2020). Moreover, tasks can be processed in batch or interactive modes. In batch processing, a batch of tasks (or workloads) must be addressed for execution over computing resources. For instance, Amazon Batch dynamically provisions the optimal amount and type of computing resources based on the volume and specific resource requirements of batch tasks submitted. Interactive execution

is a processing mode where tasks must be distributed as soon as requested.

2.2.3 High-Level Synthesis (HLS)

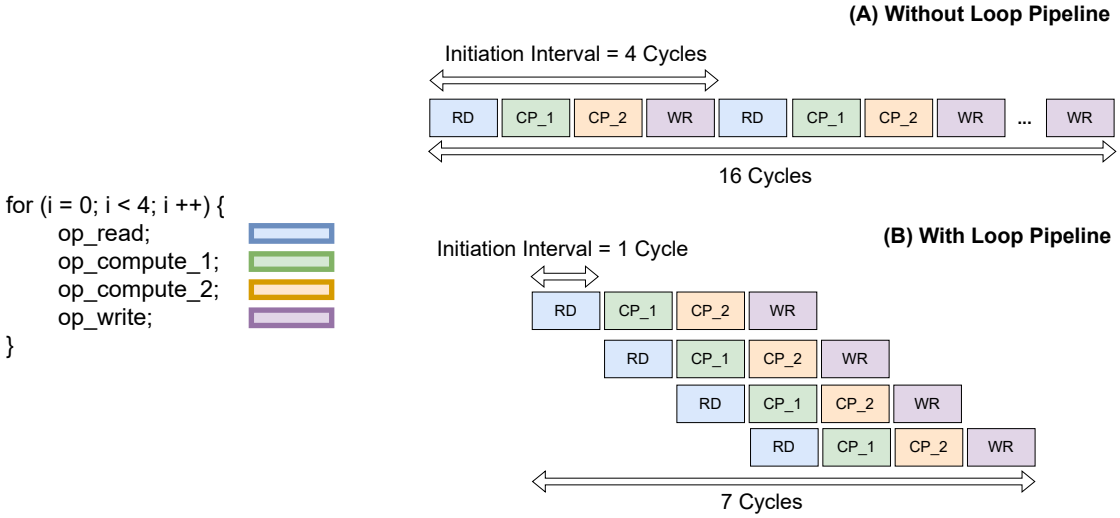
The HLS development flow is an alternative to ease hardware development efforts compared to traditional hardware description languages, like VHDL and Verilog. For that, the HLS flow provides the development of designs by using high-level description languages (e.g., such as C/C++). The high-level description is then automatically translated to the hardware description. This approach enables easy access to various hardware optimization possibilities. The use of different optimizations and their combinations result in different versions of the same design, in the same way as different binaries are generated from the same source code when using different compiler flags. These implementations result in accelerators with variant resource consumption, processing cycles, and power consumption.

The HLS optimizations are enabled through pragmas, which must be inserted in certain code regions that can be optimized. These optimizations aim at achieving: task-level parallelism, where multiple tasks can be executed in parallel over different compute units; data-level parallelism, where the data workload is split and addressed for execution over multiple compute units that execute the same task; resource optimization, which helps reducing or increasing specific use of certain hardware instances; and communication, providing efficient communication mechanisms. As it will be further presented in our Related Work, there are many works devoted to automatically providing design space exploration of pragmas over HLS designs, which can be used in future works.

We understand that there are many optimizations, and all of them can be effective for different designs. However, we focus on presenting the pragma optimizations that are promising in providing us with a trade-off between resource consumption and latency and have already shown to be the most critical in past works (ZHONG et al., 2016). Next, we present the pragmas that are already considered in our preliminary experiments and the ones that will be further employed in this thesis.

- *Loop Pipelining*: This pragma enables the concurrent execution of operations inside a loop in a pipeline fashion. The pipelined loop processes new data inputs every "n" clock cycles. The "n" factor is named the initiation interval. For instance, if the pipeline pragma is 2, new data input is processed every two clock cycles. Figure

Figure 2.6: Loop Pipelining example.

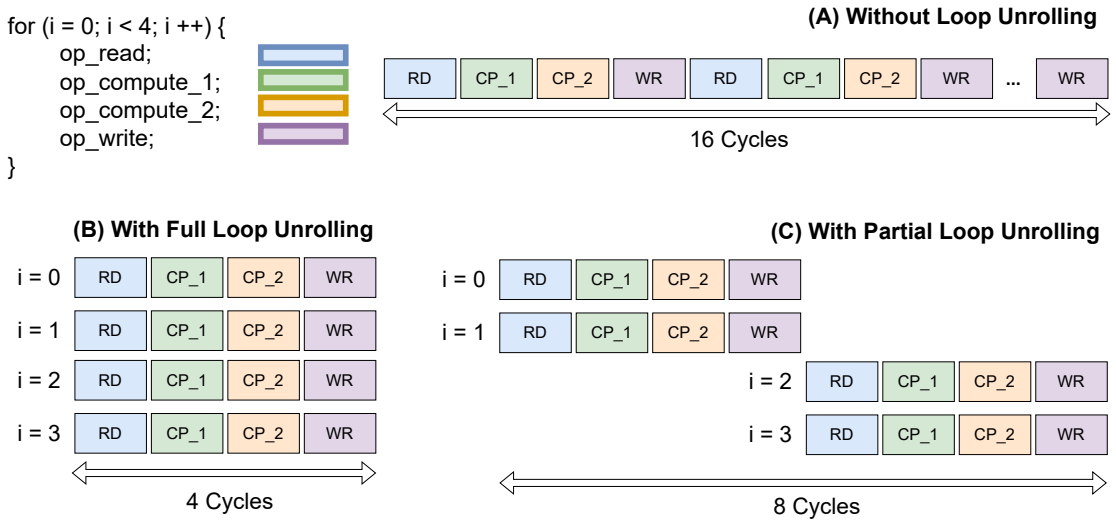


Source: The Author.

2.6 illustrates how loop pipelining works. The default sequential operation (Figure 2.6.A) takes four clock cycles between each input read, taking 16 clock cycles to perform all the operations inside the loop, which has four iterations. On the other hand, by using the loop pipelining (Figure 2.6.B) with the initiation interval set as 1, all the operations are finished in seven clock cycles.

- *Loop Unrolling*: This optimization creates multiple independent operations inside a loop that can be executed in parallel. Basically, it creates multiple copies of the loop operations inside the loop body, allowing some (partial unrolling) or all operations (full unrolling) to be concurrently performed, improving the data-level parallelism and throughput. When the loop is fully unrolled, a copy of the loop body is performed for each loop iteration. If the loop is partially unrolled, only N copies of the loop body are created. Figure 2.7 illustrates how the Loop Unrolling method works. The default sequential operation (Figure 2.7.A) takes 16 clock cycles. When the full loop unrolling is used (Figure 2.7.B), the loop is replicated four times (number of iterations). In this way, all iterations are executed in parallel, and the entire loop takes four cycles to finish its execution. On the other hand, with the partial loop unrolling configured with a factor 2 (Figure 2.7.C), the loop is replicated only two times, enabling two iterations to be executed concurrently. In this way, the entire loop takes eight cycles to finish its execution.
- *Array Partitioning*: Depending on the performance requirements, some or all elements of an array must be accessed in the same clock cycle. This technique is used to divide a large array of data into smaller sub-arrays (or partitions), which can be

Figure 2.7: Loop Unrolling example.



Source: The Author

processed in parallel by different resources on the FPGA. By partitioning the array, the FPGA can process multiple pieces of data at the same time. For example, if an FPGA is processing an array of 1,000 elements, it can be partitioned into 10 partitions of 100 elements each. Then, each partition can be processed by a different processing element on the FPGA at the same time, increasing the overall processing speed. A complete array partitioning results in all array elements being mapped to individual registers, resulting in high BRAM resource consumption.

2.3 Dynamic Voltage and Frequency Scaling (DVFS)

DVFS enables the operating frequency of a processor to be lowered (underclocking) to allow a corresponding reduction in the supply voltage (undervolting). Consequently, power consumption is reduced, leading to significant energy savings. Equation 2.1 presents the total power consumed by a CMOS integrated circuit, where C is the transistor gates' capacitance (which depends on the feature size), V is the supply voltage, f is the operating frequency, P_{static} is the static power. The necessary voltage for stable operation is related to the frequency at which the circuit is clocked, which can be reduced as the frequency is also reduced. As it can be noticed from Equation 2.1, lowering the voltage leads to a quadratic reduction in power consumption.

$$P = C * V^2 * f + P_{static} \tag{2.1}$$

Even though the advantages of DVFS are clear, its benefits have been reduced over the years (Le Sueur et al. (2010), Huang et al. (2010), Pramod Kumar et al. (2014)). When proposed by Weiser et al. (1994), the transistor technology size was 0.8 µm, while the core voltages would reach 5V. Consequently, the ratio of dynamic to static power (also known as leakage power) was high. Today's CPUs are composed of 7nm technology transistors and core voltages of 1.1-1.4V (at the highest frequency). The feature size reduction results in leakage power getting closer to the dynamic power, and, at the same time, the low voltages of the cores reduce the advantages of voltage scaling. However, even with the lowering of the benefits provided through DVFS, it still shows several advantages, as it has a simple control algorithm that can be done at the OS level or firmware (e.g., BIOS).

In modern multi-core CPUs, DVFS can be explored in two flavors (i) global DVFS and (ii) local DVFS. These techniques depend on the processor's hardware characteristics. In global DVFS, all the cores are controlled by a single voltage regulator. To avoid deadline misses, the global frequency of the chip is usually selected to match the frequency of the core with the highest execution time workload. Considering the local DVFS, multi-core CPUs incorporate voltage regulators for each core, allowing per-core DVFS, where each core operates at a particular voltage and frequency levels (KIM et al., 2008). This finer grain control provided by per-core DVFS, when smartly applied, can lead to higher energy efficiency compared to DVFS application over the whole chip, mainly for the execution of multiple parallel tasks or unsynchronized and parallel applications. This scenario can be found in Cloud environments, where CPU cores must be shared among multiple tenants. The advantages of runtime controlling and the possibility of per-core DVFS for multiple application scenarios make this method the focus of this thesis.

There are primarily two ways of employing the DVFS w.r.t multi-task applications: (i) Inter-task DVFS and (ii) Intra-task DVFS. Inter-task DVFS operates regarding the slack time generated by all processing tasks. For instance, the scaling can be used depending on the task with the highest time, where all the other tasks have their execution time stretched (by reducing the core's frequency) to avoid idle periods and provide energy reduction. However, Intra-task DVFS employs scaling over a single task, usually evaluating its worst-case execution time. In other words, depending on the input, a task can assume different execution times (e.g., due to conditional statements). Therefore, Intra-task DVFS adjusts the task execution time by stretching it to meet the maximum required execution time based on the accessed conditional statement.

Section Remarks. Energy consumption could be further reduced by exploring

voltage and frequency scaling on the FPGA side. Even though recent FPGA models and tools already support setting task designs with multiple frequency levels at the same FPGA Configuration (XILINX, 2023), unfortunately, there is no voltage scaling standard for FPGAs (SALAMI et al., 2020). Different vendors have their unique voltage management methodologies (e.g., clock/power management IPs), which makes it hard to employ this technique. Salami et al. (2020) has proposed generic methodologies for using DVFS in FPGAs, showing significant power reductions over DSPs, LUTs, and BRAM units. We look forward to the available methods to employ this technique in future works.

3 RELATED WORK

In this Section, we overview the main works developed in the correlated areas of this thesis. This chapter is organized as follows. Section 3.1 presents the works targeting multiple task execution on FPGAs and the state-of-the-art works regarding FPGA High-Level Synthesis. In Section 3.2, we study the works that use CPU power management techniques. Then, Section 3.3 discusses works that employ collaborative computing. The final section 3.4 highlights this proposal's contributions.

3.1 FPGA Environments

3.1.1 FPGAs employed for multi-tasks

Research on efficient methods for multiple-task execution in FPGA environments is not new. At the end of the 90s, several works were concerned with configuring multiple task executions over FPGA designs in resource-constrained devices. In 1997, Trimberger et al. (1997) (Xilinx, Inc.) proposed a time-multiplexed FPGA architecture, which enabled different task designs to operate over the same device by sharing the FPGA resources through reconfigurations. To effectively use the architecture, Trimberger (1998) proposed an approach for dividing large designs into multiple configurations that could be time-multiplexed into a little FPGA area through reconfigurations. After these works, several others tried to optimize/exploit the benefits of time-multiplexing tasks/subtasks in FPGAs (CADAMBI et al., 1998; MAK; YOUNG, 2003; ZHU; SUTTON, 2003; ULLMANN et al., 2004; KHAN et al., 2004).

Towards FPGA Space-Sharing. As more robust FPGA devices (with more reconfigurable resources) emerged in the early 2000s, several works were concerned with exploiting the FPGA parallelism by space-sharing resources among concurrent execution tasks. That was also the beginning of the partial reconfiguration concept. Diessel et al. (2000) proposed FPGA partial rearrangements techniques to enable fast allocation of online tasks. The fundamental assumption was that tasks that are being executed on the FPGA could be rearranged to free resources for incoming tasks. They proposed techniques with different time complexity to solve the problem, assuming rectangular resource occupation task designs. For scenarios where the reconfiguration and rearrangement delay is small compared to the processing times of the tasks, they proposed a genetic, near-

optimal approach. On the other hand (i.e., the delay is larger than the tasks processing time), they proposed lightweight algorithms named Local Repacking and Ordered Compaction. FPGAs did not provide the hardware support necessary for an arbitrary rearranging of task resources during their operation.

In the same path, Walder and Platzner (2002) proposed a study on placement techniques (First-Fit and Best Fit) to position multiple designs on the FPGA surface. They also investigated footprint transform techniques that change the resources used by a task to fit it into an existing free space. Steiger et al. (2003) proposed an operating system layer for online scheduling and placement of real-time tasks on partially reconfigurable devices. It has three modules: scheduler, placer, and loader. The scheduler receives incoming tasks and schedules their start time; the Placer distributes the tasks avoiding area fragmentation; and the loader configures tasks on the partial region. The system assumes a non-preemptive model, meaning tasks must be completed once configured on the devices.

Complementary, Handa and Vemuri (2004) proposed an algorithm to find empty spaces in partially reconfigurable FPGAs to avoid fragmentation of resources when allocating multiple tasks. Marconi et al. (2008) proposed a method for faster online placement that combines only necessary space blocks for the current task instead of merging all fragmented free space blocks." Before the spreading of FPGA devices with embedded processors, Natale and Bini (2007) was one of the first works to leverage HW/SW co-design in FPGAs by proposing a method to partition FPGA area into slots for HW tasks and soft cores for SW tasks. Before them, Pellizzoni and Caccamo (2006) proposed collaboratively using an external CPU to implement SW tasks that couldn't be placed on the FPGA, facing challenges with communication time and potential penalties on performance. In Section 3.3, we will expand the study over collaborative computing works.

Dynamic Partial Reconfiguration (DPR). With market FPGAs supporting Dynamic Partial Reconfiguration (DPR), the effective use of Partial Reconfigurable Regions for multi-task execution has been studied. Traditionally, DPR brings several advantages, like independence in time, which enables a task in a single PRR to be independently reconfigured with a new task without affecting the processing in other PRRs, enabling easy scheduling decisions such as task priorities. To effectively use these multiple PRRs, most works rely on design tools to transform an input application into a task graph.

Cordone et al. (2009) proposed an ILP-based approach to schedule multiple tasks from a single application effectively. Their approach considered configuration prefetching to hide the latency of PRR reconfiguration. Therefore, while a task is being executed over

a PRR, consumer tasks (that depend on the results of the current processing tasks) are loaded in parallel and ahead of their execution. Their approach relied on module reuse to avoid reconfigurations (of tasks that will be accessed in the future), anti-fragmentation techniques, and more than one reconfiguration controller to reconfigure more than one PRR at the same time. However, limitations due to the convergence time of their ILP approach still hold, reaching up to 39 days to produce a solution.

Wassi et al. (2014) proposed a multi-shape task management technique for partial reconfigurable FPGAs that selects between versions of the same task design, each with different resource requirements and execution times, to maximize FPGA resource utilization with a pre-defined floorplan configuration. Morales-Villanueva, Kumar and Gordon-Ross (2016) proposed a method to reduce reconfiguration time overhead by prefetching the next scheduled task bitstream and reusing the currently loaded bitstream.

Liang, Sinha and Zhang (2017) investigated multi-context FPGAs - devices that can be configured with multiple sets of configurations (also named context). The work proposed a placement algorithm and an architecture that supports pipelined reconfiguration to hide the reconfiguration overhead behind the execution of multiple tasks. Their approach divided FPGA resources into several reconfiguration units where multiple tasks are fully or partially loaded. They used a dual-port configuration memory, one for writing and reading, providing the flexibility to dynamically reconfigure reconfiguration units that are not being used at a finer granularity compared to PRRs. They also relied on a configuration memory controller to support flexible task placement and low-overhead dynamic reconfiguration. Complementarily, they proposed a static placement strategy to accommodate a set of benchmarks while maximizing resource utilization. At runtime, they used task placement strategies (First-Fit, Compaction, Minimum Conflict) to select among candidate positions and scheduling approaches to reorder task arrivals and avoid request rejection (i.e., rejecting the execution of tasks due to resource unavailability).

Dhar et al. (2021) proposed DML, a methodology for scheduling multiple application heterogeneous tasks across FPGA resources to hide the latency of Dynamic Partial Region (DPR) reconfigurations. Leveraging DPR requires manual floorplanning to designate specific regions. To overcome the challenges of DPR, they proposed an architecture design that comprises homogeneous PRRs (called slots) with fixed and uniform interface configurations. They map and generate partitioned application task graphs that fit in the slots. Also, the designer does not need to adapt the interconnection of tasks to different target configurations. Their approach uses an ILP-based scheduler that distributes

batched tasks across homogeneous-sized PRRs, focusing on overlapping and hiding the reconfiguration costs.

Enabling FPGAs in the Cloud. Other works also exploited DPR to enable FPGAs in the Cloud, as this method can provide virtualization of FPGAs in PRR granularity (resource isolation). Chen et al. (2014) proposed a hardware and software co-design framework for integrating FPGAs in the Cloud. The framework follows a mixed FPGAaaS/AaaS model, where a bitfile library is used. The Cloud provides a list of predefined accelerators, handles tenant requests, and configures accelerators into idle PRRs called slots. Each slot can only host an accelerator with compatible resource requirements and interface design. Instead of requesting programmable resources, the tenant directly requests task designs or a combination of accelerator functions. If no accelerator matches the requirements, a tenant can submit its designs, and the cloud administrator performs the synthesis and add the design to the accelerator list.

Byma et al. (2014) presented an approach to integrate virtualized FPGA-based hardware into Cloud computing systems. Similar to Chen et al., it assumes an AaaS/FPGAaaS model and uses Partial Reconfigurable Regions. However, it supports the management of multiple FPGA nodes (i.e., multi-node Level). It also adopts an input arbiter implemented as a module to control user access to their accelerator. Their prototype supports up to four tenants on one device. They implemented a round-robin to distribute the task requests among the servers for load-balancing purposes.

Fahmy, Vipin and Shreejith (2015) also proposed a Cloud FPGA prototype based on the same AaaS/FPGAaaS concepts. Their work also relied on direct memory access switch controllers to manage PRRs. Like Chen et al., they also proposed per-tenant task design chaining (also known as kernel chaining) to maximize resource utilization of the PRR. Different from the other works, all PRRs have two standard stream interfaces (AXI4-Stream) to the PCIe core and the external DRAM. They also used stream interface adapters with asynchronous FIFO to enable accelerators with different clock frequencies.

Knodel, Lehmann and Spallek (2016) proposed a framework that provides a set of software/hardware components to virtualize FPGA Partial Reconfigurable Regions. The approach enables three accelerators as service models: the exclusive use of an FPGA node, named Reconfigurable Silicon as a Service; shared FPGA resources by the use of virtual FPGAs (vFPGAs - allocated over homogeneous PRRs), named Reconfigurable Accelerators as a Service; the third one executes users' workloads as PRRs become available, named Background Acceleration as a Service. Knodel, Genssler and Spallek (2017)

extended the RC2F approach, enabling fine-grained interface control and support to task context migration among homogeneous PRRs.

Improving Resource Usage in DPR-based FPGAs. Although being an alternative for FPGA multi-tasking in the Cloud, DPR reduces flexibility in resource scaling. Minhas et al. (2021) stated that fixed-sized rectangular slots PRRs (mostly adopted by several frameworks for modern FPGAs) lead to resource underutilization, as modern data center workload tasks inherently consume heterogeneous resource requirements. According to Vipin and Fahmy (2012), within the boundaries of homogeneous PRR, the area allocated to heterogeneous tasks ranges between 38% and 51%.

To solve this problem, Zha and Li (2020) proposed a multiple homogeneous slot allocation for heterogeneous tasks, where a single task can allocate 'n' homogeneous slots to create heterogeneous slots. Nguyen and Kumar (2020) focused on maximizing the serviceability of FPGA-only multi-tenant environments. The work identified that using fixed-size and homogeneous PRRs leads to underutilization of resources, as many task requests have varying resource needs. Additionally, the resources not being used could provide more tenants to access the device in parallel. In this way, they proposed the integration of a DSE and an ILP optimization algorithm that finds the best heterogeneous-sized PRR floorplan configuration. Their solution is generated considering a pre-synthesized task library. Results showed improvements in the accelerator allocation success rate compared to homogeneous approaches. However, if new task designs are added to the library, the approach must be re-executed, taking minutes to find optimal solutions for more than nine PRRs. Moreover, the shift among floorplan configurations can also reach up to three minutes, which is prohibitive depending on the Cloud status.

In addition to the resource scaling disadvantages mentioned above, DPR also brings challenges when building designs and mapping PRRs in the FPGA fabric. As presented in Section 2.2, DPR use depends on complex design tools for partial reconfiguration, making development hard and time-consuming. Many tools try to automatically perform stages of the DPR pipeline, like floorplanning and application partitioning (among multiple tasks) (SEYOUM et al., 2021; SEYOUM; BIONDI; BUTTAZZO, 2019; RABOZZI et al., 2016). For instance, DARTS (SEYOUM et al., 2021) used an ILP approach to partition applications, generate floorplan, and schedule applications.

OpenCL as an alternative. With the emergent adoption of OpenCL by major FPGA vendors (Xilinx and Altera) and Data Centers (e.g., Amazon F1), several works have proposed exploring optimizations based on the OpenCL programming model, which

considered the use of multiple parallel task designs over the whole FPGA dynamic region (see more details in Section 2.2.1).

Minhas et al. (2021) proposed an offline DSE framework to provide an efficient clustering of tasks (kernel/task linking - OpenCL) based on their heterogeneous resource requirements and their execution time. As a single bitstream is used for each configuration, the approach selects tasks with balanced execution time to be loaded together (to avoid execution stalls due to the slowest task). The efficient use of kernel/task linking achieves higher resource utilization and performance than PRR approaches.

Bertolino et al. (2020) proposed an efficient heuristic for FPGA task scheduling (for kernel/task linking-based execution). The heuristic groups tasks by using a DAG transformation technique. Those groups are formed by considering the resource needs of tasks around a high-latency task that executes parallel to lower-latency tasks, thus hiding their latency. Results showed that their approach presents better makespan (defined as the total time to finish the execution of all tasks) solutions than Next-Fit and Earliest Finish Time algorithms in a feasible processing time (order of ms).

Dai et al. (2014) presented a benefit-based algorithm for scheduling computation resources on Cloud FPGA. The benefit-based metric considers the application throughput and uses a speedup metric that describes the FPGA computing capacity compared to a certain number of virtual CPUs. In this way, if a task achieves n times speedup on FPGA compared to its execution on the vCPU, then the FPGA area's computing capacity is equal to n CPUs. This metric models the price of allocating tasks to the FPGA based on their equivalence in terms of vCPU. They showed that the benefit-based algorithm presents higher performance than the throughput-based solution.

Jungblut and Kranzlmüller (2021) proposed an OpenCL extension called Forecast. Forecast is a C++-library on top of OpenCL that dynamically adjusts FPGA configurations to a given workload using multiple configurations of the same task (exploring SIMD or multiple PEs). For a single application, given an input workload, it selects a middle term between: an HLS version that maximizes the available FPGA resources, at the cost of configuration overhead (multiple FPGA reconfigurations); or bundles multiple processing elements into one configuration at the cost of individual task's performance. They showed that the optimal accelerator depends on the workload size and type (double or float). ECOSCALE project (MAVROIDIS et al., 2016) introduced a novel architecture that supports the benefits of partial reconfiguration and integrates a programming model that extends the OpenCL to support command queue management across PRRs. Le et

al. (2019) also proposed an OpenCL-compatible architecture. They included a custom Operating System that manages a device driver layer and a runtime manager layer. The OS can schedule services and allocate task requests to PRRs.

3.1.2 FPGA and High-Level Synthesis (HLS)

In this section, we cover the works focused on HLS optimizations, which can be orthogonally used in the future to enable: 1) easy DSE to find and analyze different combinations of HLS optimizations; 2) accurate pre-implementation estimation of performance/area metrics of each individual task; 3) expanding our benchmark and design possibilities through efficient refactoring and synthesis of tasks with variant characteristics (for example, floating-point and recursive tasks).

Khanh et al. (2015) proposed a DSE framework that exploits loop-array dependencies to find the best combination of coarse-grained HLS optimizations (loop unrolling, pipelining, and array partitioning). The framework consisted of a task analysis built upon a dependency graph with a short exploration time. Lin-Analyzer (ZHONG et al., 2016) is a tool that performs fast and accurate FPGA performance estimation and DSE according to different coarse-grained optimizations without generating any RTL implementation. It identifies bottlenecks of different FPGA implementations when applying those optimizations, assisting the designers in evaluating different architectural HLS options. MP-Seeker (ZHONG et al., 2017) is a high-level analysis framework that evaluates performance/area metrics of various accelerator options for an application at an early design space before invoking HLS tools for the final synthesis step. Contrary to Lin-Analyzer, MP-Seeker also evaluates fine-grained parameters like the tile size and number of PEs.

The COMBA (ZHAO et al., 2019) framework evaluates the effects of a multitude of directives related to functions, loops, and arrays by using analytical models, a recursive data collector, and a metric-guided DSE algorithm. Given different resource constraints, COMBA finds designs configurations with near-optimal performance. In the same scope, Cong et al. (2018b) proposed AutoAccel, a tool that automatically finds optimal HLS design configurations. AutoAccel introduced a design template that provides an analytical DSE of accelerator configurations and automatic code transformation. The template is used to evaluate the constraints of an input task, performing a reduced exploration of HLS parameters. Choi and Cong (2018) proposed a tool that produces high-performance designs for variable loop bounds. This is done through automatic code transformations

that increase the utilization of computing resources for variable loops with loop-carried dependency patterns like floating-point reduction and prefix sum.

HeteroRefactor (LAU et al., 2020) is an automated HLS refactoring tool that provides a dynamic invariant analysis (e.g., bit-width of integer, floating-point variables, and size of recursive data structures) to make traditionally HLS-incompatible programs synthesizable while optimizing the accelerator resource usage and frequency. For instance, the tool synthesizes recursive programs by refactoring pointers and recursion with access to the flattened, finite-size array.

3.1.3 Comparison over FPGA-only works

This Section shows the similarities between this thesis and the works that cover FPGA-only environments. We also present the techniques aggregated to our approach. Regarding the use of FPGA techniques, our proposal brings HLS-Versioning to generate multiple design versions (i.e., performance- and energy-oriented), which are employed at runtime to either maximize performance or reduce energy consumption.

Regarding our target execution, similarly to Minhas et al. (2021), Nguyen and Kumar (2020), we limit our scope to independent multiple-tasks from single or multiple applications, as it reflects the reality of multi-tenant Cloud. In this environment, the workload and the available resources are constantly changing. Regarding adaptability and transparency, the works that provide specific task scheduling solutions for multi-task applications are not adaptive to those changes, and some works only provide solutions in an unfeasible time. Our work dynamically adapts the resource provisioning according to the resource availability and chooses the best provisioning depending on the task request properties (e.g., task acceleration, execution time, etc.).

3.2 CPU-only Environments

The related work regarding CPU power optimization techniques is vast. The following section will focus on the relevant works that cover the techniques and their feasibility in this thesis scope.

3.2.1 Dynamic Voltage and Frequency Scaling (DVFS)

Weiser et al. (1994) were one of the first to use DVFS for energy reduction on CPU processors. They implemented an OS scheduler to gather execution traces and detect the level of slack time to select new CPU frequencies at runtime. A similar approach is used in the recent Linux on-demand governor, which minimizes idle time by changing CPU frequency in response to the current workload. After Weiser, numerous researchers have explored DVFS on single-processor systems, such as (SHIN; CHOI, 1999; SHIN; CHOI, 2000; SHIN; CHOI; SAKURAI, 2000; PILLAI; SHIN, 2001; AYDIN et al., 2004).

Inter-Task DVFS. With the spreading of multi-core architectures, a plethora of works emerged, usually targeting efficient DVFS by detecting DVFS levels over single multi-threaded applications or multiple independent tasks executed over cores. The first works were focused on managing the global DVFS ((GERARDS; HURINK; KUPER, 2014; ZHENG, 2007; LI; MARTINEZ, 2006; BHATTI; BELLEUDY; AUGUIN, 2010; PAOLILLO et al., 2014; SEO et al., 2008; MARCH et al., 2013)), where DVFS is uniformly employed over all CPU cores. For instance, in Li and Martinez (2006), the authors employed the best DVFS level depending on the number of threads a CPU executes. For that, an application is executed once for every combination (thread number and DVFS level), and energy and performance data are collected. Then, optimization mechanisms are applied to find the combination that delivers the best result. Bhatti, Belleudy and Auguin (2010) proposed a technique called Deterministic Stretch-to-Fit, which is based on inter-task real-time DVFS. Their approach dynamically extracts the completion time of all tasks inside the deadline schedule and decides the best DVFS levels so that the task execution time is stretched respecting the deadline. Similar ideas were also employed over systems that support per-core DVFS for independent (SHEIKH; PASHA, 2020; SHA et al., 2020; ZENG et al., 2009; XIAN; LU; LI, 2007) and dependent tasks (ZHANG; HU; CHEN, 2002; GUO et al., 2017; CHEN; HUANG; KNOLL, 2014).

In the past few years, DVFS has become popular in Cloud datacenters. Lin et al. (LIN et al., 2015) proposed a batch-mode task scheduling that leverages per-core DVFS on multi-cores, achieving performance and energy balance. Stavrinides et al. (STAVRINIDES; KARATZA, 2018) proposed an energy-aware heuristic for the scheduling of Cloud applications that uses per-core DVFS and approximate computing to fill schedule gaps, considering the effects of input error on the tasks' processing time. They assumed a minimum precision threshold, which must be determined offline.

Intra-Task DVFS. Several works tried to detect the best DVFS configuration at an intra-task level. In this approach, a compiler or software tool analyzes the program in advance (or in an online manner). According to the information gathered through profiling, the CPU frequency is scaled depending on the program phase with considerable accuracy. One recent work in the area was proposed by Qin et al. (2019). The work profiles information of each individual block inside a task to provide the best DVFS scheduling for a given application. They formulated an Integer Linear Programming (ILP) that considers several assumptions of transition overhead. Tatematsu et al. (2011) proposed a method based on checkpoint extraction to deal with transition overheads. By using many execution traces, it detected checkpoint places in the code for better application of DVFS to reduce transition overhead. They used a greedy approach to rank each checkpoint. Similar approaches were also proposed to use DVFS at the intra-task level efficiently (SHIN; KIM; LEE, 2001; SHIN; KIM, 2001; SEO; KIM; LEE, 2005; KONG et al., 2011; PAOLILLO et al., 2014; GUO et al., 2017).

3.2.2 Energy Efficiency through Workload Balance

Workload balance techniques guarantee that resources are allocated in a way that there is a balance between overutilization and underutilization of resources, resulting in energy optimization (KHATTAR; SIDHU; SINGH, 2019).

Several works have widely studied Energy-efficient workload balance. The works in Xiao, Song and Chen (2012) and Beloglazov, Abawajy and Buyya (2012) used VM migration for consolidating VMs into a smaller number of active servers and lower CPU utilization of overloaded servers. In Xiao, Song and Chen (2012), a set of resource allocation heuristics is proposed to save servers' energy and avoid uneven resource utilization. Beloglazov, Abawajy and Buyya (2012) proposed an energy-aware VM allocation method based on the best-fit allocation and minimization of migration policy that selects the minimum number of VMs to migrate from a host to lower the CPU utilization. The combination of techniques improved energy consumption while meeting QoS needs.

Gao et al. (2013) proposed a provisioning framework that minimizes the cloud operation cost and maximizes energy efficiency while ensuring that the user's deadlines are respected. They modeled the scheduling problem by using task graphs and balancing the distribution of workloads among servers, also considering communication bottlenecks. Resource provisioning was also considered at the node level. RALBA is a batch load bal-

ancer for non-preemptive, independent tasks focused on improving resource utilization and energy efficiency in the cloud. Yang et al. (2016) introduced an energy-performance trade-off task scheduling algorithm based on multi-objective optimization that minimizes the energy consumption of data centers and the response time of tasks simultaneously. Similarly, Juarez, Ejarque and Badia (2018) proposed a real-time dynamic scheduling approach for task-based applications in cloud multiprocessors focused on multi-objective optimization that combines energy and performance.

3.2.3 Comparison over CPU-only works

Here, we present the differences and similarities between this thesis and the works that cover CPU-only environments, highlighting the power optimization techniques aggregated to our approach. Regarding the DVFS technique, our proposal employs a local DVFS (see more in Section 2.3) to optimize the CPU execution. We selected a per-core scaling of frequency and voltage as in Cloud scenarios resource provisioning can be done at the core level. In a global DVFS approach, all CPU cores would have their frequency scaled, harming the performance of all running tasks. On the other hand, with local DVFS, we can perform fine-grained control over all independent tasks being executed. This technique enables energy efficiency in our architecture and different scaling over the workload addressed to specific cores.

The local DVFS is used in an Inter-Task manner (see more in Section 2.3). We select this granularity to easily adjust the DVFS levels regarding multiple parallel task requests, which is the reality of multi-tenant Cloud environments. In addition, finding efficient DVFS levels in an intra-task and inter-task level would increase the complexity of our power optimization. In other words, it would require a learning phase that checks for all the possible DVFS level combinations in a single task and their influence over the DVFS employed in other tasks. In this way, we focused on a lightweight Inter-Task DVFS solution that can be dynamically adopted in our collaborative environment. Efficient methods to explore intra-task and inter-task DVFS can be the object of study in the future to further leverage the benefits of this power optimization technique.

3.3 Collaborative Computing

We divide this section into three parts. First, we present works that apply CPU-FPGA collaborative computing techniques over different application niches. Then, we highlight works that mix collaborative computing and power optimization techniques for different target architectures and niches. Finally, we present state-of-the-art works that employ collaborative computing in cloud environments.

3.3.1 Collaborative Computing in CPU-FPGA Architectures.

The effectiveness of task and data partitioning has been studied for CPU-FPGA environments. Huang et al. (2019) explored collaborative execution between CPU and FPGAs by comparing task and data partitioning over several applications. They showed that the right partitioning strategy could leverage performance in CPU-FPGA environments. Likewise, Chang et al. (2017) compared the impact of both techniques considering a CPU-FPGA and a CPU-GPU architecture. They showed that a partitioning technique could bring more benefits for the same application depending on the target architecture.

Several works have explored DPR and task-based scheduling over SoC-FPGA environments. Most relied on finding an optimal distribution of multi-task applications over an embedded processor and multiple PRRs. These works also considered techniques to hide partial reconfiguration latency, like module reuse (reuse the currently loaded bitstream when possible) and configuration prefetch. They also used task pipelining to increase throughput gains.

Given that, Mei, Schaumont and Vernalde (2000) was one of the prior works to work with HW/SW partitioning. It proposed an algorithm that combines a genetic heuristic with list scheduling. Their approach targets dynamically reconfigurable devices, taking into account partial reconfiguration overhead. Later on, Banerjee, Bozorgzadeh and Dutt (2006) proposed an ILP algorithm to map and schedule multiple tasks over multiple 1-D PRRs (configuration of the whole column) and an ARM (embedded processor), considering communications and configuration prefetching. After this work, several others, like Cordone et al. (2009) (using ILP) and Ferrandi et al. (2013) (using Ant Colony Optimization), also focused on this optimization problem. However, the mainstream FPGAs are 2D reconfigurable (i.e., they can have PRR in the form of rectangles with predefined height and width partially reconfigured).

Purgato et al. (2016) proposed a heuristic to schedule tasks in a 2D-PRR ARM-FPGA SoC to reduce the overhead incurred by partial dynamic reconfiguration and leverage the number of concurrent tasks hosted on the FPGA. PaRA-Sched (CATTANEO et al., 2014) is a reconfiguration-aware scheduler that provides HW/SW scheduling considering 2-D PRRs, which considers communication latency, configuration prefetching, and module reuse. Similarly, Dorflinger et al. (2018) proposed a A^* search scheduling approach that exploits most optimization techniques previously described (no HLS use).

One of the most recent works regarding HW/SW co-design in SoC DPR FPGAs was proposed by Tang, Guo and Wang (2020). The work introduced an exact mixed-integer linear programming and a multi-step hybrid method that combines graph partitioning and mixed-integer linear programming to reduce the time complexity of the problem. The mixed-integer linear programming approach could not produce a solution in a feasible time (i.e., they considered a two hours time limit) for most of the studied applications, while the hybrid method took minutes to produce most solutions.

CPU-FPGA collaborative computing has been noticeable in specific niches, like graph processing and neural networks. Zhou and Prasanna (2017) proposed a graph partitioning scheme to enable efficient parallel computation of graph analytics applications in CPU-FPGA. The scheme explores the trade-off of vertex-centric and edge-centric paradigms to leverage the throughput of the Breadth-First Search and Single-Source Shortest Path algorithms. GraphACT (ZENG; PRASANNA, 2020) incorporated multiple algorithm-architecture co-optimizations to design a novel accelerator for training graph convolutional networks on CPU-FPGA heterogeneous systems.

Meng, Kuppannagari and Prasanna (2020) optimized the Proximal Policy Optimization of Reinforcement Learning algorithms. They proposed an accelerator for CPU-FPGA heterogeneous platform that targets both Proximal Policy Optimization training and inferences phases. The work uses task partitioning, distributing the tasks so that the FPGA executes the computationally intensive tasks, which can benefit from fine-grained parallelism. At the same time, other computations (e.g., computation of actors, rewards, and objective functions) are offloaded to the CPU, as the amount of computation is small, leading to the underutilization of FPGA resources and unnecessary communication.

NEURAghe is an efficient hardware/software solution for accelerating Convolutional Neural Networks on Zynq SoCs (MELONI et al., 2018). The solution uses a co-operative heterogeneous computing approach that distributes convolutional layers to the FPGA while the ARM cores execute hard-to-accelerate tasks (e.g., the fully-connected

layers and data marshaling), taking advantage of the NEON vector engines to achieve speedup. Compared to state-of-the-art accelerators, it achieves performance and energy improvements.

The authors in (WANG et al., 2022) proposed a hardware/software approach to speed up model training in Federated Learning by reducing the computational complexity of cryptographic algorithms in CPU-FPGA environments. The hardware part, the Hardware-aware Montgomery Algorithm (HWMA), accelerates encryption, decryption, and ciphertext-space computation through data parallelism and pipeline on an FPGA circuit. The software part, the Operator Scheduling Engine (OSE), handles non-computation tasks and divides the target algorithm into multiple calls to the HWMA.

EcoSys (ZHANG et al., 2021) is a framework for DNN-based (DNN - Deep Neural Network) video analysis that explores co-design and optimization opportunities on CPU-FPGA heterogeneous systems. The DNN layer tasks are distributed among the CPU and FPGA. To find the best task parallelism configuration for the target architecture, it uses an offline DSE to find the best number of architecture units for the FPGA (given BRAM, DSP, and memory access bandwidth constraints) and the best multi-threading configuration for the CPU (given the number of available threads). The framework includes a coherent memory space shared by the host and accelerator to enable efficient task partitioning and online DNN model optimization with reduced data transfer latency.

Collaborative computing in CPU-FPGA is also explored in Face Detection applications (e.g., computer vision and security). Mohanty et al. (2016) proposed a suite of acceleration techniques to perform real-time face detection with energy efficiency and accuracy. They first mapped the face detection algorithm to an integrated OpenCL environment. Then, they matched the algorithm's structure and found a specific face detection model (e.g., adjusting sliding window size and the number of parallel classifiers through a scaling factor) to speed up memory access and computing iterations. The performance-critical classifier stage was implemented on FPGA, and the non-critical stages were evaluated on the host CPU.

Big Data applications are also optimized through collaborative execution. Cong et al. (2017) proposed an adaptive dataflow model to orchestrate the computation among multiple CPU cores and the FPGA to improve overall system resource utilization considering big data applications. The dataflow uses the CPU cycles saved from FPGA acceleration to use I/O better. It divides the application into several stages (multiple tasks that exploit data-level parallelism among CPU and FPGA) so that all stages share in-memory

data queue connections and work in a pipelined fashion. Their approach monitors the CPU utilization for each pipeline stage at runtime to determine the CPU thread allocation that maximizes resource utilization.

Rodríguez et al. (2020) proposed a scheduler that adaptively decides the chunk of iterations (i.e., parts of parallel loop iterations) assigned to CPU and FPGA. Their approach estimates the FPGA chunk size at runtime by increasing the chunk size and sampling the application throughput until its throughput is stabilized. They used a logarithmic approach to guarantee fast convergence. The CPU chunk size is determined based on the FPGA chunk size to balance the load for both devices. In Rodríguez et al. (2022), the authors propose a solution to improve the chunk size computation in CPU-FPGA systems. The goal of their proposal is to reduce the overhead associated with the computation of the optimal chunk sizes, making the solution suitable for applications that require frequent adjustments in chunk partitioning.

3.3.2 Collaborative Computing and Power Optimization Works

Wei et al. (2017) optimized the throughput of streaming applications (individually) in CPU-FPGA heterogeneous systems using task partitioning algorithms. They used a max-flow min-cut heuristic to generate a pipelined accelerator for applications with multiple tasks. In this way, they could overlap the frame processing in both architectures. Depending on the FPGA latency at each stage, CPU DVFS levels are regulated to obtain power savings. They used POSIX threads to enable task concurrency between pipeline stages in the CPU. Thread synchronization must be manually performed for each application to support their approach. ETCF (KNORST et al., 2021b) optimizes the EDP of CPU-FPGA architectures by automatically detecting the data-partitioning level for CPU and FPGA and the near-optimal CPU number of threads for multithreaded applications (individually). The data-partitioning level (workload balance) and CPU number of threads are provided using a Hill-Climbing algorithm. The approach requires the application re-execution to detect the ideal workload balance.

The use of power optimization in collaborative computing is also explored in CPU-GPU and multi-FPGA environments. OPTiC (WANG; ANANTHANARAYANAN; MITRA, 2018) is an analytical framework that optimizes collaborative computing on mobile devices with thermal constraints. Given a thermal constraint, OPTiC can deliver optimal CPU-GPU co-execution by applying frequency throttling and adjusting co-execution

partitioning points. ETCG (KNORST et al., 2021a) is a framework that automatically selects the CPU's near-optimal number of threads to minimize the energy-delay product (EDP) of single applications in CPU-GPU. The authors statically used different levels of data partitioning to balance the load among both architectures. Jing, Zhu and Li (2013)propose eAEE, an energy-efficient scheduling algorithm for multi-FPGA for independent and dependent tasks. eAEE is based on an ant colony optimization algorithm and considers the FPGA reconfiguration overhead of all devices to reduce the global makespan and meet the deadline requirements of computing tasks.

Deiana et al. (2015) proposed a solution to provide task partitioning over an application in an ARM-FPGA SoC (2D-PRRs). Their algorithm can optimize performance or power by using HLS loop unrolling to generate multiple task versions. However, their solution can only schedule up to five tasks at a time and takes unfeasible time to complete (up to 100 seconds). Even though hiding reconfiguration latency, it does not consider communication latency among tasks.

3.3.3 Collaborative Resource Provisioning in Cloud Environments

TRIPP (VICENZI et al., 2021) schedules OpenCL applications in a CPU-GPU multi-tenant scenario to reduce makespan and energy consumption. It does not require interaction from the programmer, and the scheduling is performed at runtime. It gathers the task's execution time through OpenCL events and estimates the task acceleration by comparing the CPU and GPU execution time. If the task is executed for the first time, the acceleration is estimated based on the number of task work-items (OpenCL compute units that can execute workloads in parallel).

The work in (KNORST et al., 2022) explores CPU Thread Throttling and FPGA HLS-Versioning in a CPU-FPGA multi-tenant cloud environment. It evaluates the effects of these techniques on performance, energy consumption, and Energy-Delay Product (EDP) when both are employed together. The results point out that optimal results can only be achieved by selecting specific Thread Throttling and HLS-Versioning configuration combinations, which vary depending on the incoming application kernel.

Majumder et al. (2021) proposed a provisioning strategy called "Efficient Resource Allocation of Service Request" (ERASER). Given a set of service requests (tasks) and heterogeneous processing elements, ERASER schedules tasks among CPU and FPGA to reduce energy consumption. To increase throughput, it also migrates VMs between

servers. It uses an ILP-based technique with timing constraints to map the requests to the appropriate device.

Zhou et al. (2021) proposed MOCHA, a framework focused on cost savings for arbitrary applications with FPGA accelerators in public clouds. MOCHA profiles applications and identifies performance bottlenecks (CPU-bottleneck or FPGA-bottleneck) to partially offload tasks to CPU and FPGA nodes. Considering FPGA-bottleneck applications, MOCHA manages CPU cores to execute some tasks instead of offloading all to the FPGA. MOCHA shares one FPGA among multiple CPU nodes through the network for CPU-bottleneck ones. Liu et al. (2018) proposed an energy-efficient task scheduling algorithm to accelerate multi-task single applications in a heterogeneous multi-server CPU/GPU/FPGA infrastructure. It determines which server the application tasks must be assigned to and the DVFS profile to avoid missing the time constraint (scheduling length). They evaluate their approach over a Fast Fourier Transform and a Gaussian Elimination over synthetic time constraints.

Section Remarks. As the focus of this thesis is collaborative environments, the comparison between our proposal and collaborative works will be given in the next Section, highlighting our main contributions.

3.4 Our Contributions

In this section, we highlight the contributions of this thesis. As previously discussed, this work targets performance and energy efficiency in CPU-FPGA Cloud, where multiple task requests must be provisioned over the CPU and FPGA devices. In rented services (e.g., AaaS and FPGAaaS models), the workload is highly variant, requiring **adaptive** methods for efficient provisioning. Particularly when the architecture is shared among multiple users, the resource provisioning must be adapted according to the available resources, which vary at runtime. Moreover, in most Cloud services, **transparency** is also needed since the end-user must have their workloads executed without the knowledge about the underlying resources.

Especially in CPU-FPGA environments, there are several opportunities for reducing energy consumption, such as the use of **power optimization** techniques on the CPU, **HLS-Versioning** on the FPGA, and **collaborative computing** considering both architectures. Therefore, a solution to minimize energy in the presented scenario must consider adaptive and transparent ways of using all the aforementioned optimization axes. This ap-

proach should also minimally impact the performance of the tenant requests to respect a minimum quality of service. All these optimization fronts must be employed in a feasible convergence time (i.e., **convergence time aware**), as the Cloud workload is constantly shifting, and most of the requested tasks have short execution times (over 90% of task durations range from dozens of seconds to a few minutes (HAN et al., 2022)).

Here, we compare our thesis contributions to the state-of-the-art, considering the capability of supporting the techniques mentioned above with adaptability and transparency. Table 3.1 presents the differences between our proposal and works focusing on collaborative environments. In this table, the symbol ("✓") means that the feature studied in the respective column is employed by the work; the symbol ("✗") means that the feature is not employed; and the ("-") symbol means that the feature is not applicable to the work (e.g., Knorst et al. (2021a) consider a CPU-GPU architecture, so it is impossible to employ an FPGA optimization technique in the scope).

Overall Contribution: While some works provide the standalone use of CPU-FPGA collaborative computing and others combine collaborative computing with CPU-only power optimization techniques (e.g., DVFS and Thread Throttling) ***OR*** FPGA HLS-Versioning; this thesis is the first to bridge the gap between collaborative computing, DVFS on CPU ***AND*** HLS-Versioning to provide an efficient execution in CPU-FPGA architectures (Table 3.1, Columns 2, 3 and 4).

Our approach is designed for Cloud environments, where multiple task requests must be provisioned over the CPU and FPGA architectures. Different from all works, such collaborative provisioning is used alongside CPU and FPGA optimization techniques in a conversant manner - without reducing each other's potential. On the FPGA side, HLS-Versioning is used to enable the selection of optimized designs for FPGA execution at runtime (e.g., performance-oriented or energy-oriented). As previously studied in this thesis, using a determined HLS-Version means altering workload properties. In this scenario, our solution is the only one to provide the adaptability required to extract the maximum benefits of the optimized versions, as our provisioning adapts depending on the workload characteristics. On the other hand, our thesis synergistically employs DVFS on the CPU to further reduce energy consumption without harming the task provisioning makespan (defined as the total time to finish the execution of all tasks).

As previously studied in this Section, most works rely on detecting the best task/-data partitioning, while others detect the best power optimization for specific multi-task applications. However, those methods are unsuitable for an environment where available

Table 3.1: Comparison between collaborative computing works and this thesis.

	Collaborative Computing	Optimization Techniques		Adaptive Provisioning	Convergence Time Aware	Transparent
		CPU	FPGA			
Thesis	✓	✓	✓	✓	✓	✓
Mei et al. (2000)	✓	✗	✗	✗	✗	✓
Banerjee et al. (2006)	✓	✗	✗	✗	✗	✓
Natale and Bini (2007)	✓	-	✗	✗	✗	✓
Cordone et al. (2009)	✓	✗	✗	✗	✗	✓
Ferrandi et al. (2013)	✓	✗	✗	✗	✗	✓
Jing et al. (2013)	✓	-	✗	✓	✗	✓
Deiana et al. (2015)	✓	✗	✓	✗	✗	✓
Mohanty et al. (2016)	✓	✗	✗	✗	✗	-
Purgato et al. (2016)	✓	✗	✗	✗	✗	✓
Chang et al. (2017)	✓	✗	✗	✗	✗	✗
Zhou and Prasanna (2017)	✓	✗	✗	✗	✗	-
Cong et al. (2017)	✓	✗	✗	✗	✗	-
Wei et al. (2017)	✓	✓	✗	✗	✗	✗
Meloni et al. (2018)	✓	✗	✗	✗	✗	-
Dorflinger et al. (2018)	✓	✗	✗	✗	✗	✓
Wang et al. (2018)	✓	✓	-	✓	✗	✓
Liu et al. (2018)	✓	✓	✗	✓	✓	✓
Huang et al. (2019)	✓	✗	✗	✗	✗	✗
Zeng and Prasanna (2020)	✓	✗	✗	✗	✗	-
Meng et al. (2020)	✓	✗	✗	✗	✗	-
Rodríguez et al. (2020)	✓	✗	✗	✗	✗	✓
Tang, Guo, Wang (2020)	✓	✗	✗	✗	✓	✓
Zhang et al. (2021)	✓	✗	✗	✗	✗	-
Knorst et al. (2021a)	✓	✓	-	✓	✗	✓
Knorst et al. (2021b)	✓	✓	✗	✗	✗	✓
Vicenzi et al. (2021)	✓	✗	-	✓	✗	✓
Majumder et al. (2021)	✓	✗	✗	✓	✓	✓
Zhou et al. (2021)	✓	✗	✗	✓	✓	✓
Wang et al. (2022)	✓	✗	✗	✗	✗	-
Rodríguez et al. (2022)	✓	✗	✗	✗	✓	✓
Knorst et al. (2022)	✓	✓	✓	✗	✗	✗

resources and workload constantly change, as they are not **adaptive** (Table 3.1, Column 5). Unlike all these works, this thesis enables collaborative provisioning that dynamically adapts its solution based on the workload at hand, available resources, and the service objective, which are highly variant in Cloud scenarios.

Besides, our provisioning is **convergence time aware** (Table 3.1, Column 6), as it is always performed in a feasible time, taking into consideration the workload duration (i.e., short- or long-running workloads). To cover all workload behaviors, our work relies on - short convergence time strategies (i.e., ms to converge) that present good provisioning solutions for specific workload properties; and more robust strategies that cover a wider range of workload behaviors but with a long convergence time (i.e., seconds to converge). The most suitable provisioning techniques are selected at runtime through a fast classifi-

cation technique considering workload/architecture properties. At the same time, all the optimizations are lightweight and also **end-user transparent** (Table 3.1, Column 7), as they do not require any interference from the end-user.

4 RAHD FRAMEWORK

In this Chapter, we provide a comprehensive overview of the proposed framework. Our discussion starts with a high-level description of RAHD, followed by an in-depth analysis of its stages in Sections 4.1 and 4.2.

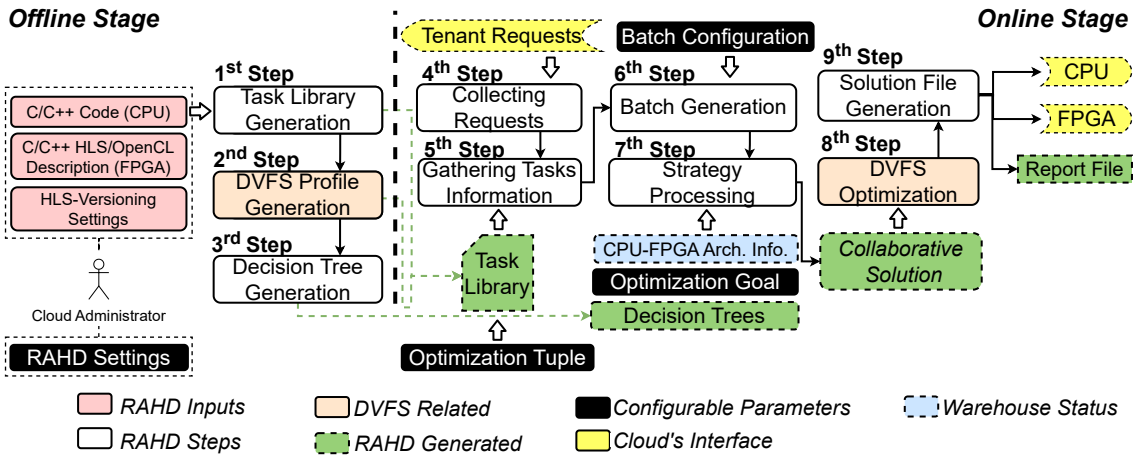
RAHD follows the AaaS model described in Section 2, where clients request the execution of their particular data inputs over pre-designed tasks offered by the Cloud. These tasks, in the form of bitstreams (i.e., for FPGA execution) and binaries (i.e., for CPU execution), along with their particular information, populate a Task Library. Upon receiving client task requests, RAHD retrieves the appropriate bitstreams and binaries from the Task Library and provisions the tasks over CPU and FPGA resources at runtime.

This provisioning can either be focused on energy efficiency or performance and employed for distinct demands (i.e., number of tenant requests). For that, RAHD enables the cloud administrator to configure three parameters - the optimization goal, which can be either set to generate high-performance or energy-efficient task allocations; the batch size, which is the number of incoming task requests dynamically batched for execution (to meet up services with variant demands); and the optimization tuple, which determines the task HLS-version to be used (e.g., performance-oriented or energy-oriented versions).

Figure 4.1 gives an overview of RAHD. It comprises offline and online stages. The offline stage is composed of three steps. In the first step, RAHD generates a Task Library composed of both *FPGA and CPU executable files* and their information (e.g., latency and power consumption), which is used in the Online Stage for efficient provisioning (for example, detecting workload properties, providing DVFS optimization, etc.).

To generate *FPGA executable files* (i.e., bitstreams), the cloud administrator in-

Figure 4.1: RAHD overview.



puts tasks' hardware descriptions (either OpenCL-based or HLS C/C++ descriptions), also providing HLS optimizations and their parameters for HLS-Versioning DSE (further detailed in Section 4.1.1.3). RAHD uses the hardware descriptions and HLS-Versioning settings to automatically generate multiple design versions for each task. RAHD also collects synthesis data of these designs (i.e., from FPGA synthesis) to gather the task's power, latency, and resource consumption information when executed over the FPGA.

To produce *CPU executable files*, the cloud administrator must provide software implementations for the offered tasks (i.e., C/C++ codes). Then, RAHD automatically compiles the software codes for the target architecture and executes their binaries (i.e., over the CPU) to gather the task's power/latency information. The second step brings additional data for the CPU tasks, adding their DVFS behavior information in the Task Library. For that, RAHD automatically executes and profiles each task's power/latency information over available DVFS levels.

As discussed in Sections 1.3 and 3.4, RAHD comprises several provisioning strategies to cover variant workload characteristics in a feasible time. In the third step, RAHD uses the Task Library information to generate synthetic workloads and produce two decision trees, which are responsible for detecting the best provisioning strategy for the incoming workloads during the Online Stage. One of the decision trees is optimized to identify the provisioning strategy that yields the best performance improvements, while the other decision tree is designed to select the strategy that prioritizes energy efficiency.

After finishing the three steps, the cloud administrator specifies the RAHD settings (i.e., optimization tuple, optimization goal, and batch configuration) that were previously described at the beginning of this Section (these settings can also be updated at runtime).

At the Online Stage, the server is configured to receive tenant task requests. In the 4th Step, RAHD collects task IDs and associated data inputs from tenant requests. During the 5th Step, it retrieves the properties of the requested tasks from the Task Library, including the characteristics of the task when executed on both CPU and FPGA, such as latency, power consumption, and resource usage. To do so, RAHD accesses the Task Library and retrieves the line that contains the task information using the task's ID. After fetching the line, RAHD analyzes the data and collects the CPU-related and FPGA-related task properties.

For each task, RAHD has multiple HLS versions for FPGA execution and only retrieves information of the version set by the cloud administrator at the Offline Stage, which is defined by the optimization tuple (further explained in Section 4.2.1). For in-

stance, if the optimization tuple is set for performance, RAHD will retrieve data only for the performance-oriented HLS version (i.e., the version considered for FPGA execution).

In the 6th Step, each task ID and respective properties are sent to a FIFO to form a batch of tasks (size configured by the cloud administrator). The 7th Step is responsible for selecting a strategy to perform the distribution of the batched tasks among CPU and FPGA. For that, RAHD reads the optimization goal (configured by the cloud administrator) to decide whether to use the performance or energy decision tree for strategy selection (generated in RAHD's 3rd Step). Then, RAHD uses the chosen decision tree to output the most suitable strategy to distribute the tasks from the current batch, considering the collected task properties and target architecture.

After selection, the strategy produces a provisioning solution with tasks distributed over both devices (named Collaborative Solution). Then, based on the generated Collaborative Solution, the 8th Step uses a fast heuristic to detect the DVFS levels that will balance FPGA and CPU execution to achieve energy savings without harming the makespan (the total time to finish the execution of all tasks).

Finally, the 9th Step generates a Collaborative Solution file with a queue of tasks addressed to each CPU core (with annotated DVFS levels and tasks' ID) and a queue of tasks to the FPGA (with annotated tasks' ID), which can be translated into execution queues supported by the OpenCL model. This step also produces a report file, which comprises the Collaborative Solution's makespan (i.e., metric used for performance in this work) and energy consumption. The next Sections bring more details for each stage.

4.1 Offline Stage

This Section brings detailed information on all RAHD Offline Stage steps. We present the steps in the same order presented in Figure 4.1.

4.1.1 Task Library Generation (1st Step)

This Section provides a detailed description of RAHD's 1st Step. We start by describing the Task Library structure; then, we explain how the Task Library components are generated.

4.1.1.1 Task Library Structure

Figure 4.2 depicts the Task Library structure. It is divided into execution files and profiled information. Regarding the execution files, it comprises at least one binary and one bitstream for each available task to enable its execution over the CPU and/or FPGA. Due to HLS-Versioning, a task might present multiple HLS-Versions, producing several bitstreams with variant delay/power/area characteristics (for example, BitstreamA has versions V1 and V2).

As RAHD adopts the OpenCL execution model (Multi-Task Full Reconfiguration - MTFR), described in Section 2.2.1, the Task Library also includes containers that comprise multiple tasks within the same bitstream. This allows for concurrent execution of tasks on the FPGA through execution queues, as further discussed in Section 4.2.4.1.

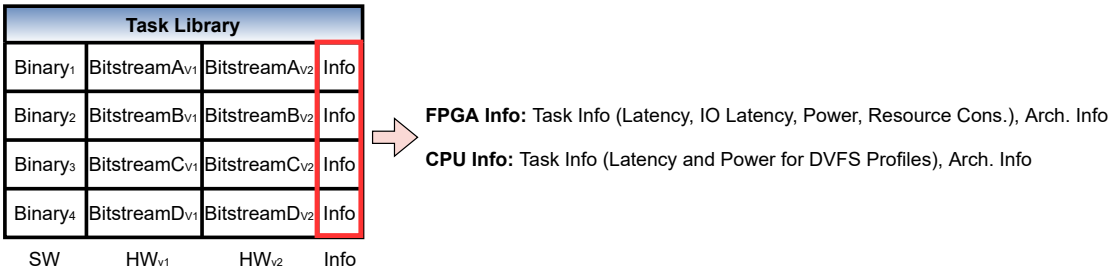
The Task Library also contains the profiled information of each task regarding its execution on FPGA and CPU (e.g., latency, resource requirements, power consumption, etc.). Moreover, all CPU tasks in the Task Library have their delay and power consumption added for different DVFS profiles. RAHD also profiles architectural information, such as the FPGA reconfiguration time and the CPU power consumption in the idle state.

The next Sections describe how the Task Library execution files and profiled information are extracted. Section 4.1.1.2 shows how binaries and bitstreams are generated. Section 4.1.1.3 explains the HLS-Versioning procedure to generate multiple task design versions. Finally, Section 4.1.1.4 shows how the tasks and architecture properties are extracted.

4.1.1.2 Binaries and Bitstreams Generation

Software binaries are easily built using software compilers, allowing straightforward assignment of tasks to CPU resources (e.g., address the binary to an individual or

Figure 4.2: Task library structure.



Source: The Author.

multiple CPU cores). The cloud administrator must provide each task implementation code in C/C++ so the compiler can translate them into binaries. RAHD uses GCC 12.1 as the compiler for generating the binaries for the target architecture.

For FPGA executable file generation, the cloud administrator inputs task hardware descriptions, which must be either implemented in OpenCL or HLS C/C++. RAHD uses the hardware descriptions to automatically generate multiple design versions for each task (i.e., for HLS-Versioning exploitation) by using the Xilinx Vivado HLS tool.

Each task design produces object files, which can be merged into containers through the kernel/task linking technique described in Section 2.2.1, enabling the parallel execution of tasks in the same FPGA Configuration. These containers can be produced by using different policies. For instance, we can prioritize generating containers that combine the tasks usually assigned to the FPGA by our provisioning strategies. This can be easily achieved at the Offline Stage by executing synthetic workloads over our strategies. This information can also be tracked at the Online Stage, and containers can be scheduled for generation in idle periods or dedicated machines.

It is important to notice that RAHD implements a logic to operate when specific containers are unavailable, as will be described in Section 4.2.4.1. Our experiments evaluate RAHD under a container limitation scenario (described in Section - 5.1). The next Section details how HLS-Versioning is provided in our framework.

4.1.1.3 Generating Multiple HLS Versions

To generate multiple design versions, RAHD receives as inputs: (a) the HLS C/C++ accelerator description, provided by the cloud administrator; (b) a list of HLS pragma optimization options to be explored - loop pipelining, unrolling, and array partitioning -; (c) and, their particular factors (e.g., initiation interval for pipelining, unroll factor for unrolling, and partitioning factor).

To provide HLS-Versioning, the accelerator description (a) must comprise the annotation - "//insertpragma" - in the code regions to be optimized by loop unrolling and partitioning techniques. For employing array partitioning, the accelerator description must comprise the annotation "//insertpartpragma.NameOfArray", where the "NameOfArray" is the name of the array to be partitioned.

RAHD uses the accelerator description as a template, replacing the provided annotations with each optimization pragma (b) and their factor combinations (c), generating multiple accelerator descriptions with variant HLS pragma combinations. Finally, our

Figure 4.3: Distribution of versions for the MD5 and Syr2k designs.

MD5 Versions

α:0.0	β:0.0	β:0.2	β:0.4	β:0.6	β:0.8	β:1.0
γ:0.0	-	17	17	17	17	17
γ:0.2	7	7	17	17	17	17
γ:0.4	7	7	7	17	17	17
γ:0.6	7	7	7	7	17	17
γ:0.8	7	7	7	7	7	17
γ:1.0	7	7	7	7	7	7

α:0.2	β:0.0	β:0.2	β:0.4	β:0.6	β:0.8	β:1.0
γ:0.0	1	23	23	23	23	23
γ:0.2	23	23	23	23	23	6
γ:0.4	6	6	6	6	6	6
γ:0.6	6	6	6	6	6	6
γ:0.8	6	6	6	6	6	6
γ:1.0	6	6	6	6	6	6

α:1.0	β:0.0	β:0.2	β:0.4	β:0.6	β:0.8	β:1.0
γ:0.0	1	1	1	23	23	23
γ:0.2	1	23	23	23	23	23
γ:0.4	23	23	23	23	23	23
γ:0.6	23	23	23	23	23	23
γ:0.8	23	23	23	23	23	23
γ:1.0	23	23	23	23	23	23

Syr2k Versions

α:0.0	β:0.0	β:0.2	β:0.4	β:0.6	β:0.8	β:1.0
γ:0.0	-	4	4	4	4	4
γ:0.2	2	2	2	4	4	4
γ:0.4	2	2	2	2	2	2
γ:0.6	2	2	2	2	2	2
γ:0.8	2	2	2	2	2	2
γ:1.0	2	2	2	2	2	2

α:0.2	β:0.0	β:0.2	β:0.4	β:0.6	β:0.8	β:1.0
γ:0.0	27	2	2	2	2	2
γ:0.2	3	2	2	2	2	2
γ:0.4	2	2	2	2	2	2
γ:0.6	2	2	2	2	2	2
γ:0.8	2	2	2	2	2	2
γ:1.0	2	2	2	2	2	2

α:1.0	β:0.0	β:0.2	β:0.4	β:0.6	β:0.8	β:1.0
γ:0.0	27	3	3	3	2	2
γ:0.2	3	3	3	3	2	2
γ:0.4	3	3	3	3	2	2
γ:0.6	3	3	3	2	2	2
γ:0.8	3	3	3	2	2	2
γ:1.0	3	3	2	2	2	2

HLS-Versioning approach synthesizes the produced HLS designs through the Xilinx Vivado HLS tool, using the various descriptions to generate design versions with variant performance/energy/area trade-offs.

The generated versions are organized over the Task Library by each design version property. This structure is used at the Online Stage to find a design by a tuple of weights (α, β, γ), where α represents the area, β the performance, and γ the energy weights. Figure 4.3 presents the distribution of different design versions in our Task Library for two of our evaluated benchmarks: Syr2k and MD5. Each table is set for one value of Area weight (α, 0.0, 0.2, 1.0). In their rows, the Energy Weight (γ) varies with a granularity of 0.2, and in their columns, the same granularity is used to vary the Performance Weight (β). The automatic generation from HLS leads to a significant design variety. On average, five different versions were generated for the benchmarks from our experiments, demonstrating the potential of our HLS-Versioning feature.

HLS-Versioning Limitations. We limit our multiple-design version generation in frequency and processing loop sizes. We have adopted both limitations due to time constraints in this research. Multiple frequency task designs in the same bitstream are a common practice in more recent FPGAs. Xilinx Vitis tool enables to set of different frequencies for each individual task addressed to a bitstream (XILINX, 2023). However, in our HLS-Versioning process, we have limited our investigation by synthesizing all tasks with the same target frequency, respecting slack time boundaries.

Regarding loop sizes, our approach does not vary the loop range when performing HLS-Versioning. We understand that managing loop boundaries can also lead to more

variety of tasks as HLS optimization capabilities like loop unrolling and pipelining are affected when the size of the loop changes.

4.1.1.4 Task Library Data

RAHD needs several data, which are comprised in the Task Library and used during the Strategy Processing Step (to evaluate workload properties and provide efficient provisioning - explained in Section 4.2.2.1) and to predict the energy/makespan spent by our produced solutions (explained in Section 4.2.4.2). This Section describes each piece of information and how they are collected in the following bullets:

- *FPGA task's latency*. The time a task takes to execute over the FPGA. Our framework considers the FPGA task's latency as the sum of its computing time on the FPGA and its communication latency. This data is extracted through previous hardware emulation reports (OpenCL approach - Xilinx SDAccel) or HLS synthesis (Xilinx Vivado HLS), where the task runtime and its IO latency can be measured;
- *FPGA task's power*. The power consumption of a given task design. This data is extracted through the Xilinx Vivado tool (after task implementation) or Xilinx Power Estimator, which enables a fast power estimation based on the task's used resources;
- *FPGA task's resource consumption.* The resources used by each task over the FPGA - in terms of LUT/FF/BRAM/DSP/IO. This data is extracted through the Xilinx Vivado tool. We track both the individual task's area consumption and also the area of the whole bitstream (in case multiple task designs are merged);
- *FPGA reconfiguration time and reconfiguration power.* The time the FPGA takes to load a new configuration. The reconfiguration time depends on several parameters, like the bitstream size, disk IO, PCIe latency, configuration overhead, and many other operations. Our experiments showed an average reconfiguration time of 400ms, which is close to values present in the literature and Xilinx tutorials for similar FPGA architectures (MOODY, 2021; XILINX, 2020; XILINX, 2019) (used in our methodology). Considering reconfiguration power, some previous studies (NAFKHA; LOUET, 2016; RIHANI et al., 2016) have reported values of up to $\sim$ 720mW, which is the value considered in this work;
- *CPU task's latency and power consumption*. The time a task takes to execute and the power it consumes over the CPU. The data extraction is performed offline using

the AMD uProf tool. For Intel CPUs, alternative tools such as PAPI, Intel RAPL, or Linux turbostat can be utilized. The collected power information includes core power, idle core power, and uncore power.

Section Remarks. It is important to notice that we track the latency of a single task iteration. This enables RAHD to calculate, at runtime, the estimated execution time for a specific task with variant input sizes. We understand that depending on the input type, some tasks may present different behaviors - for instance, they may access different code conditions that lead to variant latencies. Several works aim at tracking workload behavior at runtime. In future works, we will consider these methodologies to adjust our provisioning at runtime so that we can further improve the efficiency of our framework.

4.1.2 DVFS Profile Generation (2nd Step)

In this step, RAHD first collects the voltage and frequency configurations supported by the target CPU architecture using the CPUFreq tool native from Linux (KUMAR; CHAWLA; MUKHOPADHYAY, 2020). Using the tool, we can gather the frequency range and the available scaling governors. In this work, we use the userspace governor, which enables us to set per-core CPU frequency by using the following commands [1]:

```
$ sudo cpufreq-set -c $core -g userspace
$ sudo cpufreq-set -c $core -f $frequency
```

RAHD uses these commands over a bash script that automatically executes all CPU binaries present in the Task Library for different DVFS profiles, gathers their latency and power consumption, and updates the library with these data. Our approach uses the AMD uprof tool to extract latency and power.

4.1.3 Decision Tree Generation (3rd Step)

This step focuses on the generation of decision trees, which are responsible for selecting the most suitable provisioning strategy during the Online Stage (Strategy Processing step). This step uses the Scikit Learn tool to generate two decision trees, one for

[1] where -c is the argument to set a specific governor/frequency for a core, -g defines the scaling governor, while -f defines the particular frequency.

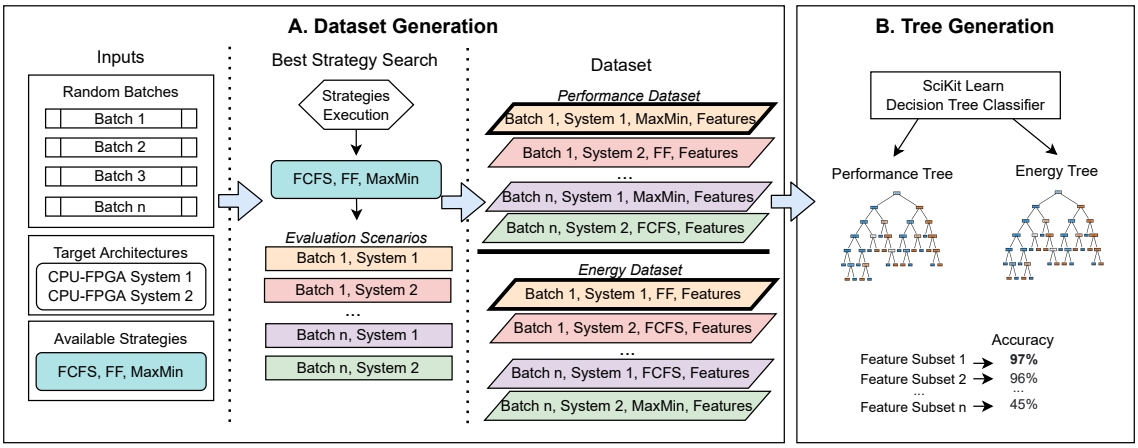
performance optimization and the other for energy optimization. For that, it receives as input a dataset composed of synthetic batches used for training, the target architecture, and provisioning strategies.

This process is illustrated by Figure 4.4. It is composed of two main phases, Dataset Generation and Tree Generation. During the **Dataset Generation phase**, it takes as input synthetic generated batches (randomly produced by using tasks with variant characteristics), available target architectures, and provisioning strategies (*Inputs*). It uses the inputs to generate evaluation scenarios that are given by tuples. The first element of the tuple is the workload (in the example, Batch 1, Batch 2, and so on), and the second is the target architecture (in the example, System 1 or System 2). Each tuple passes through an exhaustive search that uses all available provisioning strategies (in the example, FCFS, FF, and MaxMin) to find the best provisioning strategy for a given evaluation scenario in terms of performance and energy, which will be used for training the decision trees.

The search will produce the correct classification (i.e., best strategy) for performance and energy. In the example of Figure 4.4, the best strategy for performance considering the Evaluation Scenario Batch 1-System 1 tuple was the MaxMin strategy. In contrast, the best one for energy was the FF. The Oracle execution generates a Performance and an Energy Dataset.

The Datasets comprise the Evaluation Scenario (used as a dataset ID), the best provisioning strategy for the scenario (dataset label), and the Evaluation Scenario features. The Evaluation Scenario features are descriptive attributes that characterize the batch, while the label is the information we want to predict. During the **Tree Generation**, the generated Dataset is used to create the decision trees automatically. For that, we inserted the Datasets in the Scikit Learn tool - using the decision tree classifier class.

Figure 4.4: Decision tree generation.



After generated, the decision trees will be composed of several internal nodes representing a test on a feature that will conduce to a leaf represented by the label. Each leaf node represents a label (decision taken after computing all features), which, in our scenario, is the best provisioning strategy for a batch represented by a set of features. Therefore, features are used to train and as input to find the best strategy after the trees are already produced. Moreover, different sets of features lead to diverse accuracy levels. RAHD automatically tests multiple subset combinations to find the optimal set for accuracy until a stopping criteria (e.g., accuracy goal) [2] In the example, subset one was selected for its highest accuracy. For the decision trees used in this work, the selected features were divided into: architectural information - FPGA model and CPU model; workload information - batch size; and workload information when executed over the target architecture - average task acceleration, average task CPU execution time, average task resource consumption, and FPGA task parallelism (given by the arithmetic mean of the number of tasks that fit in the same FPGA reconfiguration).

The final product of the Decision Tree Generation is performance and energy decision trees capable of choosing the best strategy for a given workload that must be executed over a target architecture. We point out that, by using this process, we enable the cloud administrator an easy alternative to inserting, on-demand, any provisioning strategy to cover even more workload behaviors.

Dataset Generation Inputs. We used batches with different characteristics (i.e., workload type and batch size) to generate our dataset. We produced the dataset using the seven workload types described in Section 5.1 and three batch sizes - 40, 100, and 200. We used this methodology to produce high input variability when training both trees. The dataset comprised the execution of 210.000 random batches (i.e., random combinations of task requests from our AaaS library, no repetition) over the ten architecture combinations.

We used 80% of the aforementioned dataset to train and 20% to test our Decision Trees, following the same methodology employed in (HEMDAN; SHOUMAN; KARAR, 2020; KUMARI; MEHTA, 2020), which uses the Pareto principle distribution. For each workload type (7 workload types), we experimented with 2.000 batches of 40, 100, and 200 task requests (6.000 batches for each workload type, totaling 42.000 batches) that were executed over ten architecture combinations. It is important to notice that the batches used for testing in our experiments were not included in the decision trees' training process. By using this methodology, we could achieve 86% accuracy for the Performance

[2] In future works, we aim to test other approaches with faster processing time for feature selection (VISALAKSHI; RADHA, 2014).

Decision Tree and 84% accuracy for the Energy Decision Tree.

4.1.4 Wrap-up

As shown in Figure 4.1, the Offline Stage results in a Task Library populated with binaries, bitstreams, and their additional information (e.g., latency, resource, and power consumption). It also comprises information on the impact of each DVFS profile over each task and multiple HLS design versions for each task (i.e., organized by their properties), which are necessary data for the Online Stage to perform resource provisioning, DVFS, and HLS-Versioning. This stage also generates decision trees, which are responsible for selecting provisioning strategies at the Online Stage.

4.2 Online Stage

This Section brings detailed information on RAHD Online Stage steps. We present the steps in the same order presented in Figure 4.1.

4.2.1 Collecting Requests and Gathering Tasks Information (4th and 5th Steps)

At the Online Stage, the server is set to receive tenant task requests. The 4th step collects these task requests through their IDs and respective data inputs. During the 5th step, RAHD accesses the Task Library to collect each task's properties (characteristics of the task when executed over CPU and FPGA, such as latency, power consumption, and resource usage). For that, RAHD fetches the line that comprises the task's information using the task's ID. The line is divided into CPU-related information and FPGA-related information. The FPGA-related information contains data related to each design version previously generated through HLS-Versioning. RAHD uses the optimization tuple to collect only the data related to the design version that optimizes for the desired metric. For example, if the tuple is set for performance optimization, it will only extract data from the performance-oriented version.

In more detail, the logic used to collect the FPGA-related information for the desired HLS Version in the Task Library consists of the following steps: first, **(a)** RAHD receives as input the task request's ID, and a set of weights (*Optimization Tuple* (α,β,γ));

then, **(b)** for each version of the task ID in the Task Library, calculates the Design Value, which is given by Equation 4.1[3]. The equation uses the optimization tuple to get the task's most suitable version. Then, it selects the design that presented the lowest Design Value (the lower the value, the larger the benefit); finally, **(c)** RAHD delivers the selected design version information to the next Step. For example, in Figure 4.3, in case the Cloud administrator wants to maximize energy benefits, it would consider a tuple with weights $\alpha = 0$, $\beta = 0$, and $\gamma = 1$, leading to Syr2k's design version number 2 is selected.

$$\text{Design Value} \rightarrow \alpha\ Area + \beta\ Exec.Time + \gamma\ Energy \tag{4.1}$$

4.2.2 Batch Generation and Strategy Processing (6th and 7th Steps)

In the Batch Generation step, each task ID and respective information is sent to a FIFO to form a batch. The number of incoming task requests dynamically batched for execution respects the batch size configuration defined by the cloud administrator (initially set at the Offline Phase but can be updated at runtime).

Once the batch is built, it is sent to the Strategy Processing step. This step *automatically selects and executes the most appropriate provisioning strategy for each incoming batch of tasks*. For that, it uses the trees generated during the Decision Tree Generation step to select the best provisioning strategy for a given workload and target architecture. Figure 4.5 illustrates how this step works. It receives as input: an optimization goal to select between the Performance or Energy Tree; an input tuple composed of the batch features and target architecture for the Decision Tree Execution; and the batch, composed of task IDs and their characteristics used in the provisioning strategy processing. In the example, the Performance Decision Tree was selected as the optimization goal was con-

[3] Area, Execution Time and Energy were normalized considering the Task Library's maximum and minimum values of each. The area is given by the task's worst-case percentage of FPGA resource utilization (BRAM, LUT, DSP, FF, and IO)

Figure 4.5: Decision tree execution.

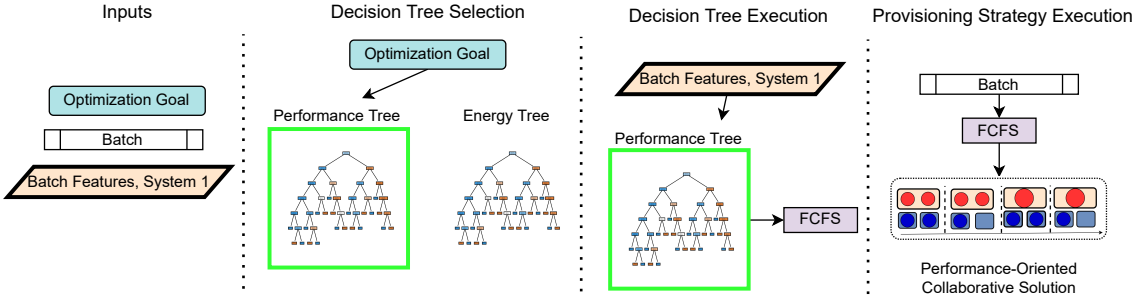
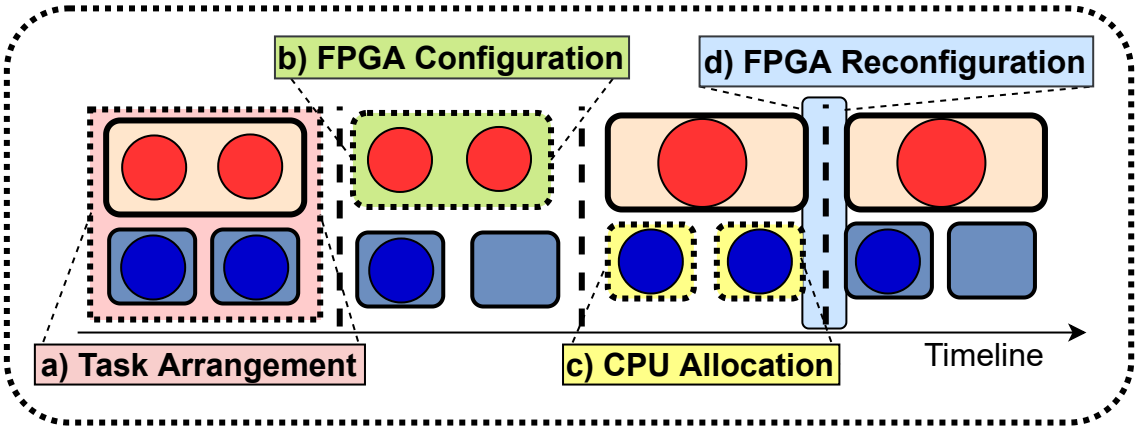


Figure 4.6: Collaborative Solution overview.



figured for performance (*Decision Tree Selection*). Based on the workload features and target architecture, the Performance Decision Tree selected the FCFS strategy to perform the allocation of the batch (*Decision Tree Execution*), producing a performance-oriented Collaborative Solution (*Provisioning Strategy Execution*).

The selected provisioning strategy generates a Collaborative Solution, depicted by Figure 4.6. It can be composed of one or multiple Task Arrangements (Figure 4.6.(a)) distributed over time (x-axis, in the example four arrangements). A Task Arrangement is comprised of tasks dispatched to the FPGA (FPGA Configuration - Figure 4.6.(b)), and tasks offloaded to the CPU (CPU Allocation - Figure 4.6.(c)). An FPGA reconfiguration (Figure 4.6.(d)) occurs between Task Arrangements.

4.2.2.1 Provisioning Strategies

This Section lists all the available strategies that compose the Decision Trees and briefly describes how they produce Task Arrangements to build a Collaborative Solution. A detailed description of the resource provisioning strategies is present in the appendix (Section Appendix.A).

First-Come First-Served: Assigns tasks in incoming order to the FPGA (until they fit). The next tasks are distributed among CPU cores (one task each). This strategy does not consider any workload characteristics when distributing tasks, distributing tasks among CPU and FPGA according to their resource availability.

First-Fit: This strategy is projected to consider the task's FPGA Acceleration when addressing tasks. It assigns the batch's accelerable tasks (i.e., tasks that present higher acceleration than a threshold value) to the FPGA until they fit. Other tasks are assigned to the CPU so that the Task Arrangement completion time, or makespan, given

by the FPGA Configuration, is not increased. First-Fit suffers because tasks may demand high resource consumption, producing FPGA reconfigurations and impacting the FPGA task parallelism.

Genetic Multidimensional Knapsack (GMK): It was proposed to optimize multi-task allocation in CPU-FPGA environments by the author in Jordan et al. (2021c). It was designed for performance (GMK-P) or energy (GMK-E). It first generates an FPGA Configuration by using a genetic procedure that selects task designs that: I) are more representative in terms of energy (GMK-E) or performance (GMK-P); II) has more acceleration benefits when executed on the FPGA; and, III) require less resource provisioning so that more tasks can be allocated in parallel. Then, GMK-P/E dispatch the remaining tasks to the CPU so that the Task Arrangement makespan is not increased. However, the GMK-E only assigns tasks to the CPU if they consume less energy than when executed in the FPGA. GMK genetic parameters are set as follows: population to 200, generations to 200, crossover to 0.8, and mutation to 0.08.

Both GMK-P and GMK-E consider several task characteristics in their allocation. Given that, they can generate efficient solutions considering variant types of workload inputs. However, as shown in Section 5.3, they suffer from poor convergence time, which is the time the algorithm takes to produce a solution. In this way, this strategy is not adequate for scenarios where the solution must be delivered fast.

MaxMin and MinMin: Both heuristics distribute the tasks over the FPGA and CPU by sorting their FPGA processing time. They sort the task list by FPGA Time in ascending order (MinMin) and descending order (MaxMin), assigning tasks to the FPGA while they fit. The remaining tasks are assigned to the CPU so that the Task Arrangement makespan, given by the built FPGA Configuration, is not increased. The idea behind the MaxMin algorithm is that we can execute many short tasks concurrently while executing the larger one. The total makespan is determined by the execution of the longer task in this case. On the other hand, MinMin focuses on finishing as many tasks as possible within a schedule. However, MaxMin and MinMin do not consider resource consumption, which leads to a high number of FPGA reconfigurations.

RASA: The algorithm was built through a comprehensive study and analysis of MinMin and MaxMin task scheduling algorithms in Parsa and Entezari-Maleki (2009). When assigning batch tasks to a resource, RASA applies the MaxMin and MinMin algorithms alternatively. Basically, a large task is selected immediately after a small one and vice versa. We employ the technique for the FPGA while tasks fit (i.e., considering their

execution time when executed on the FPGA). After, we distribute tasks to the CPU in the same way without affecting the overall makespan. The idea is that it uses the MinMin strategy to execute small tasks before the large ones and applies the MaxMin strategy to avoid delays in the execution of large tasks and to support concurrency in the execution of large and small tasks. RASA does not consider the task resource consumption characteristic, suffering from the same problem stated in the First-Fit and MaxMin/MinMin approaches.

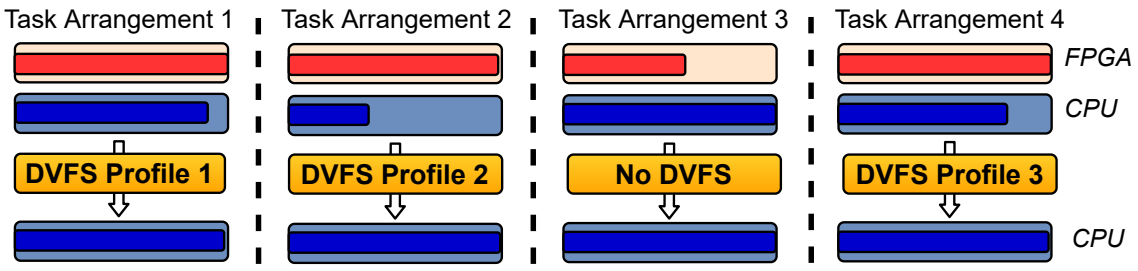
Round-Robin (RR) and Weighted Round-Robin (WRR): RR builds the allocation by distributing one task to each architecture cyclically. WRR can be configured to distribute more tasks to the FPGA. We applied a 3:1 distribution to WRR, where three tasks are assigned to FPGA and a single task to CPU. Similar to FCFS, these strategies do not consider task characteristics in their allocation.

4.2.3 DVFS Optimization (8th Step)

The 8th Step employs the DVFS energy optimization over the CPU, considering the *Collaborative Solution* from the previous step by using the DVFS Profiles information collected during the 2nd Step (DVFS Profile Generation), which was stored in the Task Library. Figure 4.7 illustrates the technique employed for a Collaborative Solution composed of four Task Arrangements. The framework evaluates the most suitable DVFS levels so that FPGA and CPU execution times are balanced (i.e., for each Task Arrangement). As it can be seen, this step avoids employing DVFS (i.e., Task Arrangement 3) when the FPGA execution time is longer than the CPU's for a given Task Arrangement.

To explore the best DVFS profiles for each Task Arrangement, this step follows a Hill Climbing heuristic. Let us consider having ten different profiles (defined by an array of profiles $\alpha[n]$, where n is the array index), composed of a no-DVFS profile ($\alpha[0]$) plus nine profiles ranging from the DVFS profile of highest frequency and voltage ($\alpha[1]$) up to

Figure 4.7: DVFS Optimization Step overview.



the lowest one ($\alpha[9]$). We define the Hill Climbing step size as β.

For each Task Arrangement where the FPGA execution time is higher than the CPU execution time, the algorithm takes the steps below on each CPU core:

1. test DVFS profile $\alpha[9]$. If the makespan is not increased, finish the algorithm. Else, set $\beta = floor(n/2)$;
2. decrease n using β ($n = floor(n - \beta)$). Go to step 4;
3. increase n using β ($n = n + \beta$);
4. if $n = 0$, finish the algorithm using no DVFS profile. Else, set $\beta = \beta/2$. If $\beta <= 1$, set β to 1;
5. test $\alpha[n]$. If the makespan is not increased, go to step 3. If $\beta = 1$ and the last tested profile did not increase the makespan, select the last tested profile and finish the algorithm. Else, go to step 2.

4.2.4 Solution File Generation (9th Step)

RAHD's final step generates a collaborative solution file, which outputs the tasks assigned to each device and information on used DVFS profiles and HLS versions. It also generates a report file (i.e., by using the collaborative solution file), which comprises information regarding the collaborative solution's energy consumption and makespan. Section 4.2.4.1 overviews the collaborative solution file and how it can be used for execution. Section 4.2.4.2 gives more details about the report file and explains how makespan and energy metrics are calculated.

4.2.4.1 Collaborative Solution File

As it can be seen in Figure 4.1, in its final step, RAHD is responsible for generating a Collaborative Solution file. This file contains information on where each task request was assigned (i.e., FPGA or CPU execution), the sequence of their execution, and the CPU DVFS configuration for each core. The file can be divided into two parts: FPGA and CPU execution queues. The FPGA queue will be composed of a sequence of FPGA Configurations, each one comprising the ID of the assigned tasks. If the FPGA Configuration comprises more than one task, the order of their execution will be given by the task with the longest execution time to the shortest. The CPU queue will be composed of

a sequence of CPU configurations. Each CPU configuration will have sequences of tasks assigned to each CPU core, the ID of each task, and DVFS level data for each core.

Having this file format provides the information needed to provision task requests over FPGA and CPU resources (through the FPGA and CPU queues), retrieve the necessary binaries and bitstreams/containers from the Task Library (through each task's ID), and configure DVFS levels for each core at runtime. Next, we explain how these solutions can be executed and which tools can be used. Before that, we first need to understand how the available programming models enable the configuration of multiple tasks. The current version of the framework was projected for MTFR support. In this way, we explain how the OpenCL model works and how it can be used to execute Collaborative Solutions.

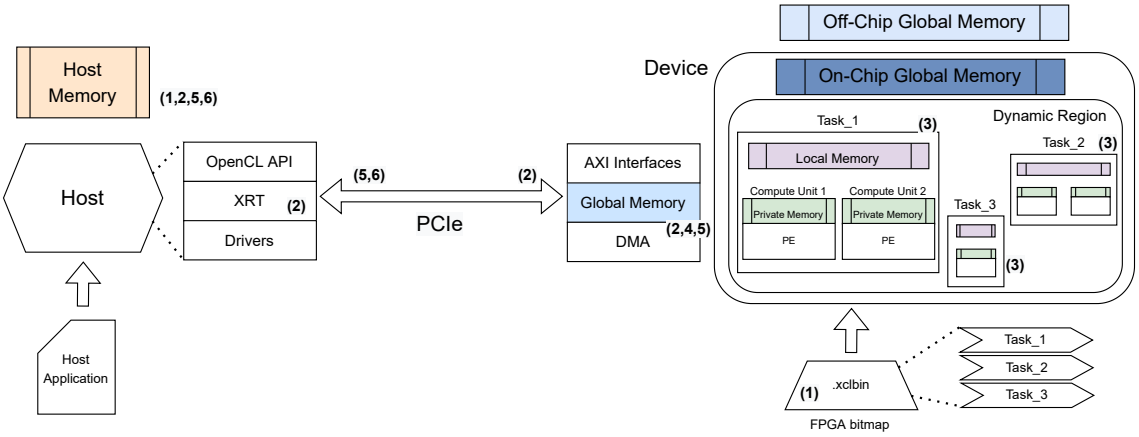
Understanding Xilinx SDAccel/Vitis OpenCL Model. Figure 4.8 shows the general structure of this acceleration platform. In this environment, an application is split between the host application and device-accelerated tasks, which communicate through a shared channel. The FPGA hardware platform contains the hardware-accelerated task designs configured in the Dynamic Region.

The Application Programming Interface calls are managed by the Xilinx Runtime Library (XRT), which is used to communicate with the tasks (hardware accelerators). The communication between host and device (e.g., control and data transfers) occurs through a PCIe bus. The control information is transferred between specific memory locations (Local/Private Memory), while the global memory is used for data transfer between the host application and the tasks. Both host and device have access to the global memory. All compute units can access the local memory inside a task design, while the private memory is only accessible to the specific compute units.

The execution model can be broken down into the following steps:

1. The host application loads a .xclbin binary in the FPGA, which comprises one or several task designs (i.e., an FPGA Configuration, in our scope) that will be used during the application's processing;
2. The host application writes the data needed by a task into the global memory of the FPGA device through the PCIe Interface;
3. The host application triggers the execution of the task on the FPGA;
4. The task performs the required computation while reading data from global memory, as necessary;
5. When the task finishes its processing, it writes the data back to the global memory

Figure 4.8: Xilinx SDAccel/Vitis OpenCL model.



Source: The Author.

and notifies the host that it has finished its execution;

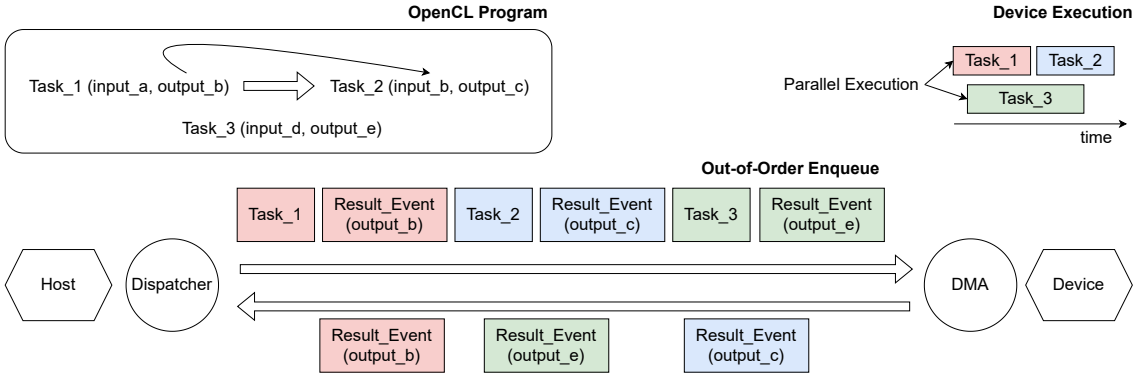
6. The host application reads data back from global memory into the host memory; continues processing as needed, or finishes the overall execution.

The FPGA can accommodate multiple task design instances at once; this can occur between different types of tasks (to provide task parallelism) or multiple instances of the same task (to provide data parallelism). The XRT transparently orchestrates the communication between the host application and the tasks in the accelerator. Compilation options determine the number of instances of a task. Next, we detail how CPU and FPGA Configurations can be executed using the OpenCL model.

Enqueueing Tasks from an FPGA Configuration. In an OpenCL application, the concurrent execution of tasks in an FPGA Configuration can be provided through an out-of-order command queue. In this approach, a dispatcher uses the Xilinx Runtime Library to transfer task arguments and workload, trigger commands, and start the computation on the accelerator running on the device. During queue programming, the host must define dependencies and synchronizations among tasks if they exist.

Figure 4.9 illustrates the behavior of an OpenCL Out-of-Order Queue. In this example, let us consider the execution of three tasks: Task_1, Task_2, and Task_3. Task_1 and Task_2 must be executed sequentially (i.e., due to data dependency), while Task_3 can be executed in parallel. The out-of-order enqueuing can execute the events in any other. In this way, it triggers Task_3 and Task_1 for parallel execution. As soon as the Task_1 result is ready, it triggers the computation of Task_2. The device computation outputs are delivered as soon as the computation is finished. *It is important to notice* that the tasks used in the enqueueing process must be present in the loaded container

Figure 4.9: OpenCL out-of-order enqueueing process example.



Source: The Author.

(following step 1 of the execution model). If an FPGA Configuration comprises tasks not present in the container, we have to deal with FPGA Configuration leftovers, which will be explained in detail next.

FPGA Configuration Leftovers. During the Strategy Processing, the strategies may converge to FPGA Configurations that do not have ready-to-deploy containers in the Task Library (i.e., the container was not yet produced). In this case, RAHD tries to find a similar container to be used in the Collaborative Solution file. The selected container is the one that has more similar tasks in comparison to the current FPGA Configuration.

The leftover tasks (i.e., not present in the selected container) are assigned for CPU execution among the CPU cores with less workload (i.e., workload balancing). Before that, we clear the previously converged DVFS profiles in the Collaborative Solution file. Therefore, CPU cores will have larger idle times, which can be filled by the leftover tasks, potentially reducing makespan overheads. Then, after allocating all leftover tasks, our lightweight DVFS optimization is re-executed. Finally, the Collaborative Solution file is updated, and RAHD extracts the impact on makespan and energy (i.e., further discussed in Section 4.2.4.2).

We avoid building new task designs and container implementations at runtime since the process of producing a bitstream may demand hours. Because of that, FPGA Configurations that have not yet been built can be scheduled for container implementation and added to the Task Library in idle periods or dedicated machines/resources.

Executing the CPU Configurations. Tasks executed on the CPU device can be multiplexed at different granularities depending on the availability of the resources. For instance, tasks can be distributed over CPU cores by adjusting the tasks' process affinity to an individual or a set of CPU cores. These tasks can also be en-queued in a sequential manner to specific resources. This could also be achieved by using the OpenCL device

fission that allows sub-dividing a device into two or more sub-devices. For example, one can divide a 16-core device into 16 subdevices of 1 core each. Then it is also possible to create a command queue for each subdevice and enqueue tasks to individual cores.

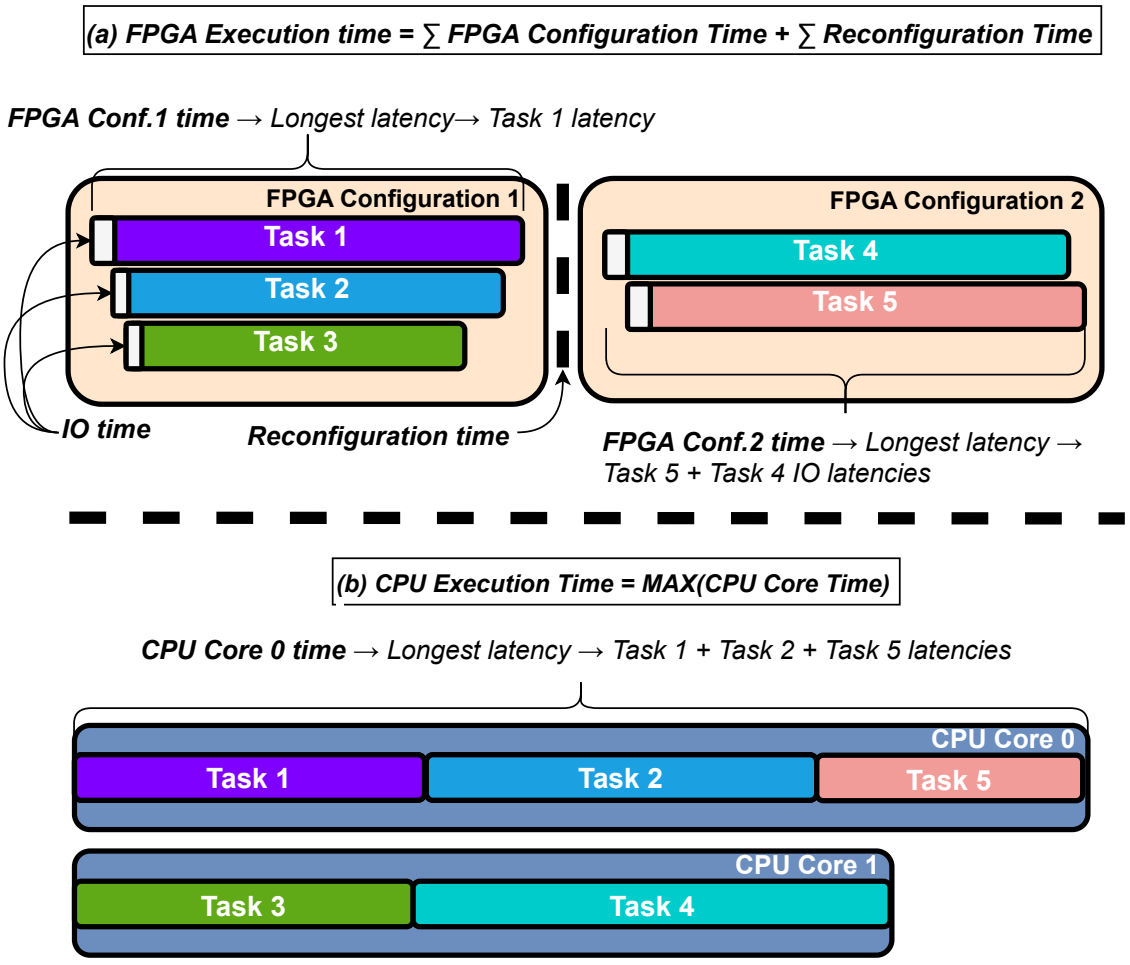
The next Section shows how makespan and energy consumption metrics are calculated by using the collaborative solution file.

4.2.4.2 Report File Generation

The report file comprises the energy and makespan information of the current Collaborative Solution, which are calculated by using the data previously stored in the Task Library (described in Section 4.1.1.4). Next, we show how makespan and energy consumption are extracted.

Makespan extraction. The makespan is given by the longest execution time between FPGA execution and CPU execution. Figure 4.10 gives an illustrative example of how both metrics are calculated based on a Collaborative Solution.

Figure 4.10: CPU and FPGA execution time extraction.



We start by evaluating the FPGA execution time (Figure 4.10.(a)). To calculate FPGA execution time, RAHD evaluates all FPGA configurations from the Collaborative Solution. The FPGA execution time is given by the sum of each FPGA configuration time plus the time taken on each FPGA reconfiguration (equation in Figure 4.10.(a)).

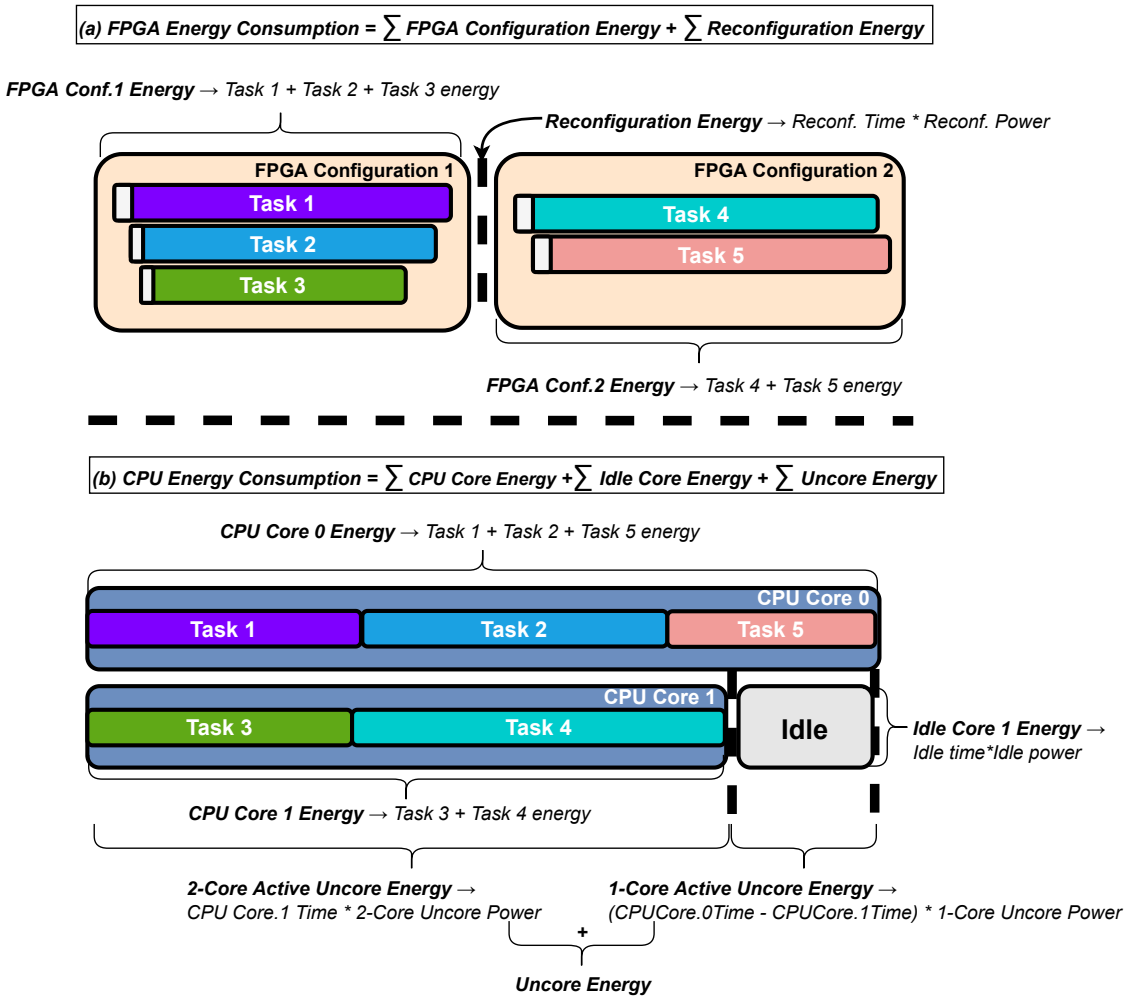
The FPGA configuration time is measured by gathering each task's latency in the Task Library. If the FPGA configuration is composed of a single task, the FPGA configuration time is given by the latency of the task. In case it is composed of multiple tasks, RAHD orders the tasks by their latency in descending order - RAHD dispatches tasks (i.e., for each FPGA configuration) from the longest to the shortest time, so the task with the longest time will not be affected by the IO time of other tasks, and the shorter tasks can be executed in parallel with the longest one (similarly as employed in (BERTOLINO et al., 2020)). After ordering, RAHD calculates each task's latency, where the latency of the first task is given by the sum of its IO time and latency; the second task's latency is given by the sum of the first task's IO plus its IO time and latency; the third task's latency is given by the sum of the first and second task's IO plus its IO time and latency; and so on. The FPGA configuration time is given by the time the last task finishes its execution minus the start time of the first task.

In the example from Figure 4.10.(a), the Collaborative Solution comprises two FPGA Configurations. In the first FPGA Configuration, the latency is given by Task 1, which has the longest time. As can be noticed, even though Tasks 2 and 3 started their execution later, they finished their execution before Task 1 completion, resulting in no additional time. This highlights the importance of ordering tasks' dispatch by their latency. In the second FPGA Configuration, even though RAHD ordered both tasks in descending order, both tasks have similar times. Consequently, the execution time was given by Task 5 latency plus Task 4 IO latency.

To calculate CPU execution time, RAHD evaluates the execution time of each CPU core. The core with the longest execution determines the CPU execution time, as presented in Figure 4.10.(b). For that, it first checks the tasks assigned for each core, also collecting the DVFS profile employed for each task. Then, with the task ID and DVFS profile at hand, it extracts the respective delays from the Task Library. Each CPU core time is given by the sum of the task delays assigned to the respective core. In the example from Figure 4.10.(b), CPU Core 0 presented the longest execution time, given by the sequential execution of tasks 1, 2, and 5 (i.e., task 1 + task 2 + task 5 latencies).

Energy extraction. The total energy is given by the sum of the energy consumed

Figure 4.11: CPU and FPGA energy consumption extraction.



by the FPGA and CPU devices. Figure 4.11 uses the same Collaborative Solution example from the former figure to show how both metrics are calculated.

We start by evaluating the FPGA energy consumption (Figure 4.11.(a)). The FPGA energy consumption is given by the sum of the energy spent on each FPGA configuration and the energy spent each time the FPGA loads a new configuration (i.e., Reconfiguration Energy). The FPGA configuration energy is measured by the sum of the energy consumed by each executed task. Their energy is calculated by multiplying their delay and power consumption, which are present in the Task Library. The reconfiguration energy will be given by the reconfiguration latency multiplied by the reconfiguration power. In the example from Figure 4.11.(a), the FPGA Configuration 1 will consume the sum of the energy spent by Tasks 1, 2, and 3, while the FPGA Configuration 2 will consume the sum of the energy spent by Tasks 4, and 5.

The CPU energy consumption is given by the sum of - (1) the energy spent by each core (considering core power only); (2) the energy consumed by each core in the

idle state; (3) the energy consumed by the uncore during the CPU execution (i.e., power of CPU parts that are not in the core, such as the L3 cache).

The energy consumed by each core (1) is given by the sum of the energy spent by each task assigned to the respective core. Each task's energy is given by the multiplication of their delay and power consumption (i.e., only the power consumed by the core used by the task is considered - depends on the DVFS Profiles), which are both present in the Task Library. The idle energy (2) is given by the time the core spent idle multiplied by the core power in the idle state.

As shown in (GUPTA et al., 2012; CHENG et al., 2015), the uncore power (3) increases with the number of active cores, mainly due to the increase in the last-level cache (LLC) access rate. As our work explores the execution of multiple concurrent tasks, to extract the consumed power with maximum precision, we would have to test all task combinations when executed over the cores, which would be unfeasible. Therefore, we limit our scope to tracking the uncore power depending on the number of active cores. The uncore energy is given by the sum of the energy spent when all cores (n) were active; n-1 cores were active; n-2 cores were active; and so on.

In the example from Figure 4.11.(b), Core 0 energy is given by the sum of the energy spent when executing Tasks 1, 2, and 5, while Core 1 energy is given by the sum of the energy consumed by Tasks 3 and 4. We can also notice that CPU Core 1 was kept idle for some time, so the energy spent by this core in the idle state is also counted. Finally, the uncore energy is measured. In the example, both cores were active until the Core 1 workload was finished. Therefore, the uncore energy consumed until this time considers the uncore power when two cores are active. Then, Core 0 continued its execution for a time, which is given by the Core 0 workload deadline minus the Core 1 workload deadline. The uncore energy consumed during this period considers uncore power when one core is active. The uncore energy is given by the sum of the energy when both cores were active plus when a single core was active.

5 EXPERIMENTAL RESULTS

This chapter evaluates RAHD by studying its optimization fronts individually and their symbiotic use. We start by presenting the methodology (Section 5.1) used in our experiments. Section 5.2 shows the potential of collaborative provisioning and multi-tenancy on CPU-FPGA Cloud. Section 5.3 evaluates RAHD's adaptive selection of multiple resource provisioning strategies, showing the need for adaptability when workload and target architectures vary. Section 5.4 explores the RAHD's DVFS technique, reinforcing the need for synergistic DVFS to achieve energy benefits. Then, Section 5.5 evaluates the effectiveness of RAHD's HLS-versioning to either maximize performance or reduce energy consumption. Section 5.6 shows RAHD's full benefits by exploiting all the optimization axes - adaptive resource provisioning, HLS-Versioning, and DVFS. Finally, we summarize the conclusions taken from each experiment in Section 5.7.

5.1 Methodology

Evaluation Setups: Our evaluation environment is presented in Table 5.1. It comprises an 8-core AMD Ryzen 7 3800x, a 64-core AMD Ryzen Threadripper 3990X CPU, and five FPGAs from Xilinx: Alveo U200, Alveo U50, Virtex-7 7VX1140T, and 7VH870T, and Kintex-7 7K410T. The tools used to build and extract the task library information were the following: **AMD uProf** for CPU performance and power; **Vivado HLS 2019.1** for synthesis and FPGA performance; and **Xilinx Vivado 2019.1** for FPGA implementation and power extraction. Regarding the DVFS profiles, the RAHD's second step could collect profiles ranging from 3.9GHz to 1.1GHz considering the AMD 3800x CPU and profiles ranging from 2.9GHz to 2.2GHz for the AMD 3990X CPU.

Benchmark Sets: The tasks were taken from Xilinx Vision (XILINX..., 2019),

Table 5.1: Evaluation environment.

CPU	**Cores**	**Frequency**	**Cache**	**TDP**
AMD Ryzen 7 3800X	8	3.9GHz - 4.5GHz	32Mb L2	105W
AMD Threadripper 3990X	64	2.9GHz - 4.3GHz	32Mb L2	280W
FPGAs	**LUTs**	**BRAMs**	**FFs**	**DSPs**
Xilinx Alveo U200	1,182,240	4,320	2,364,480	6,840
Xilinx Alveo U50	872,000	2,688	1,743,000	5,952
Xilinx Virtex-7 7VX1140T	712,000	3,760	1,424,000	3,360
Xilinx Virtex-7 7VH870T	547,600	2,820	1,095,200	2,520
Xilinx Kintex 7 7K410T	254,200	1,590	508,400	1,540

Table 5.2: Benchmark set 1 - characterization and workload types.

	Characterization Tuple	Workload Types						
		Long Execution	Short Execution	High Acceleration	Low Acceleration	High Resources	Low Resources	Heterogeneous
3D Rendering	(M,M,L)						■	■
ADI	(L,L,L)		■		■		■	■
Convolve	(H,M,L)	■					■	■
Digit Recognition	(H,H,H)	■		■		■		■
Face Detection	(L,L,M)		■		■			■
FIR	(L,M,L)		■				■	■
Floyd-Warshall	(L,H,L)		■	■			■	■
Histogram	(H,H,H)	■		■		■		■
IDCT	(H,M,H)	■				■		■
Kmeans	(M,M,H)					■		■
MxM	(M,M,H)					■		■
MD5	(M,M,L)						■	■
Median Filter	(L,H,L)		■	■			■	■
Optic Flow	(M,M,H)					■		■
Pivot	(L,L,L)		■		■		■	■
RowCol	(L,L,L)		■		■		■	■
Seidel	(L,H,L)		■	■			■	■
Spam Filter	(L,L,H)		■		■	■		■
TriSolv	(L,L,L)		■		■		■	■
Watermark	(H,H,L)	■		■			■	■

Rosetta (ZHOU et al., 2018), (LIU; BAYLISS; CONSTANTINIDES, 2015), and (CONG et al., 2018a). All the tasks were synthesized with a target frequency of 300MHz for Alveo U200 and U50 and 200MHz for the other FPGAs, respecting slack time boundaries. We characterized the benchmarks using a tuple, where the first element is the CPU Execution Time, the second is FPGA Acceleration, and the third is FPGA Required Resources. They were classified into High (H), Medium (M), or Low (L). The CPU Execution Time level is determined by comparing the task CPU time with the task with the longest CPU time. Resource requirements are given by the task highest % of consumed resources over the FPGA (among BRAM/LUT/DSP/FF/IO). For both metrics, 0-15% is low, 16%-40% is medium, and above is high. The FPGA acceleration is the speedup over CPU, where 0.5x-2x is low (under 1x means slowdown); 2.1x-4x is medium, and above is high. The resource requirements classification considered the average of the metric extracted from each available FPGA. The execution time and acceleration level classification were performed using the metrics extracted from the Alveo U200 and AMD Ryzen 7 3800x.

We divided our benchmarks into two sets. The first benchmark set is used in the experiments performed in Sections 5.2, 5.3, 5.4, while the second benchmark set is used in Sections 5.5 and 5.6, which involve HLS-Versioning experiments. Due to time limitations, we could only extract HLS versions for the second benchmark set. Both benchmark sets were classified using the same methodology. Tables 5.2 and 5.3 present the benchmarks (leftmost column) that comprise sets 1 and 2, respectively. Our workloads comprise Machine Learning, Image/Video Processing, Spam Filtering, Digital Signal Processing, Graph/Visual-Crowd Analysis, Cryptography, Mathematical, and Stencils. The bench-

Table 5.3: Benchmark set 2 - characterization and workload types.

	Characterization Tuple	Workload Types						
		Long Execution	Short Execution	High Acceleration	Low Acceleration	High Resources	Low Resources	Heterogeneous
ADI	(L,L,L)		■		■		■	■
Atax	(L,L,M)		■		■			■
Backprop.	(M,L,H)				■	■		■
Bicg	(L,M,L)		■				■	■
CFD	(L,H,H)		■	■		■		■
DCT	(H,L,L)	■			■		■	■
Ges.	(L,L,H)		■		■	■		■
Heat$_3D$	(H,L,M)	■			■			■
Jac$_1D$	(H,M,L)	■					■	■
Jac$_2D$	(H,L,H)	■			■	■		■
KNN	(L,L,M)		■		■			■
MD5	(M,M,L)						■	■
NW	(M,H,M)			■				■
Pathf.	(L,L,H)		■		■	■		■
Pivot	(L,L,L)		■				■	■
RowCol	(L,L,L)		■		■		■	■
Seidel	(L,H,L)		■	■			■	■
Srad	(H,H,H)	■		■		■		■
Syr2k	(H,L,L)	■			■		■	■
Syrk	(H,L,L)	■			■		■	■

marks' references, classifications, and descriptions can be found in Appendix B.

Evaluation Scenarios: To evaluate RAHD over variant workloads, we used the aforementioned characterization to generate seven batch types by grouping different task requests. We have configured RAHD's Batch Generation step (see more in Section 4.2.2) to consider variant batch size configurations (i.e., 40, 100, and 200) to see the impact of larger batches over resource provisioning. The produced batch types were: heterogeneous, considering requests of all tasks; High Acceleration ("high" FPGA acceleration requests); Low Acceleration ("low" FPGA acceleration requests); Short Execution Time ("low" execution time requests); Long Execution Time; Low Resources ("low" resource consumption); and, High Resources. Tables 5.2 and 5.3 highlight the tasks that comprise each batch type.

For the benchmark set 2, used in our HLS-Versioning experiments, we configured RAHD's optimization tuple (see more in Section 4.2.1) to use performance-oriented (i.e., optimization tuple configuration - 0,1,0) and energy-oriented designs (i.e., optimization tuple configuration - 1,0,0), as our work focuses on performance and energy improvements. We have also considered non-optimized designs as our baseline in Sections 5.5 and 5.6. For each scenario from benchmark set 1, 3.000 batches (i.e., totaling 21.000 different batches) were executed over the ten architecture combinations. For each scenario from benchmark set 2, 3.000 batches were executed (i.e., totaling 21.000 different batches) for each design type (i.e., baseline, performance-oriented, and energy-oriented) over the ten architecture combinations. Therefore, 42.000 different batches were consid-

ered for testing, following the methodology presented in Section 4.1.3.

We also study how well our framework performs under a limited number of available containers (see more in Sections 2.2.1 and 4.2.4.1) as the Cloud environment may have strict time and resources to produce containers. We aimed to determine a feasible number of containers that could be generated within a one-week period. Our experiments revealed that the implementation time for most containers on an AMD 3800x with 64Gb RAM ranges from several minutes to a few hours. Based on these results, we established an implementation time estimate of 1 hour per container, yielding a total of 168 containers for the week, which corresponds to the number of hours in a week.

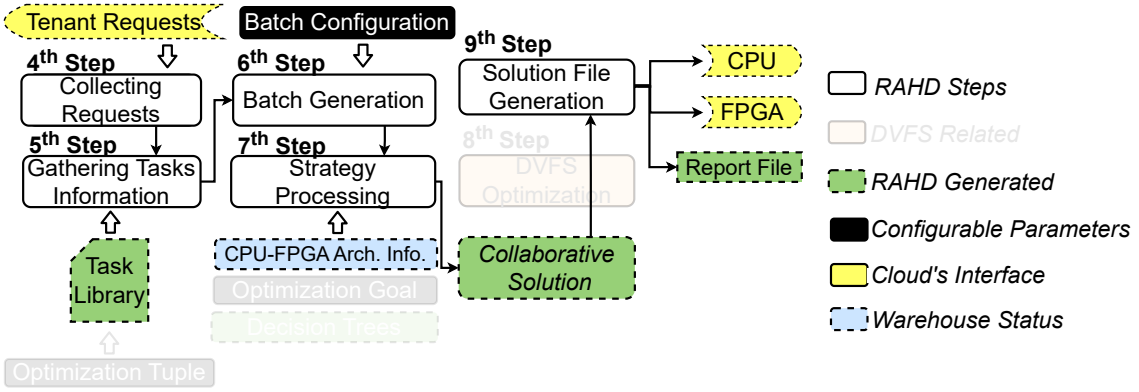
We targeted selecting the containers that could cover the FPGA Configurations generated by our provisioning strategies over variant workload properties (i.e., our batch types). For that, we collected all FPGA Configurations produced by our strategies for each batch type (i.e., 7 batch types) and stored the top 12, resulting in 84 containers. We performed this for the performance-oriented versions and energy-oriented ones - resulting in 168 configurations. The top 12 configurations included those produced by GMK-P, GMK-E, FF, MaxMin, MinMin, and RASA, with two configurations being selected from each strategy considering the current batch type (i.e., the most converged ones). We excluded configurations produced by load balancing strategies (FCFS, RR, and WRR) as they produce FPGA Configurations based on the order that tasks arrive, which may result in unprofitable configurations. We consider the limit of containers during Section 5.6, which evaluates RAHD as a whole. For the other Sections, we aim to show the full benefits of each optimization axis, so we did not restrict the number of containers.

5.2 Multi-Tenant Collaborative Execution Benefits

This Section uses RAHD to show the potential of combining multi-tenancy and collaborative resource provisioning over multi-tenancy only, collaborative execution only, or neither, following the experiments proposed in (JORDAN et al., 2021c). For that, we disable the DVFS, HLS-Versioning, and adaptive provisioning selection to gather the isolated gains provided by multi-tenancy and collaborative execution.

Figure 5.1 illustrates the RAHD Online Stage features evaluated in this Section. As can be seen, all the configurable parameters related to the selection of HLS versions (i.e., Optimization Tuple) and the selection of decision trees (i.e., Optimization Goal) are disabled. The DVFS Optimization Step was also deactivated.

Figure 5.1: RAHD Online Stage (Section 5.2 evaluated features).



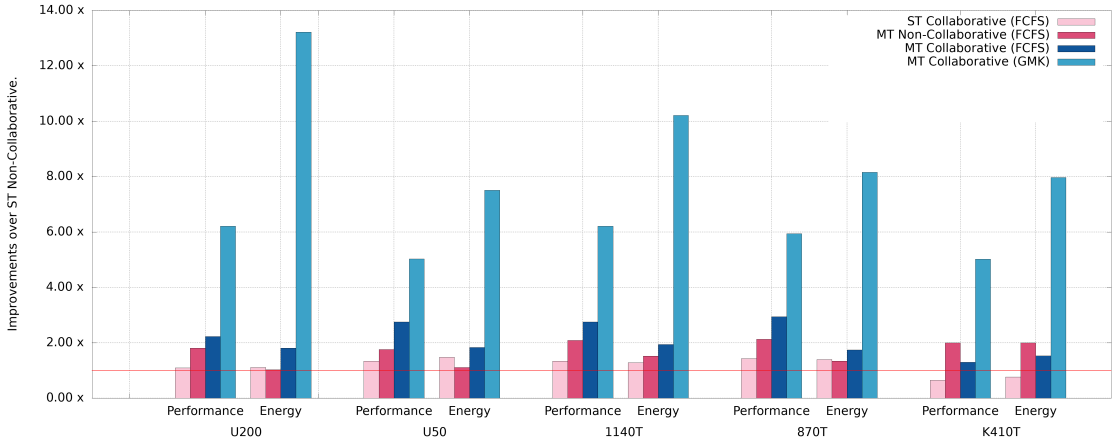
Source: The Author.

5.2.1 Results Evaluation

For this experiment, we use the AMD Ryzen 7 3800x CPU and vary the FPGA to see how different collaborative architectures benefit from multi-tenant collaborative execution. We consider ten tenant request sets, each one comprising 20 tasks (totaling the execution of 200 tasks - "Heterogeneous" scenario from benchmark set 1).

In Figure 5.2, we compare the Multi-Tenant Collaborative execution, Multi-Tenant Non-Collaborative, and Single-Tenant Collaborative approaches over the Single-Tenant Non-Collaborative (baseline). In the y-axis, we have the performance (given by the makespan metric) and energy improvements for the Single-Tenant Non-Collaborative environment as the baseline. In the x-axis, we separate performance and energy results for each target FPGA architecture. We executed all environments using FCFS resource provisioning. We also considered using a more efficient resource provisioning (GMK) to show

Figure 5.2: Cloud environments performance and energy comparison.



Source: The Author.

the potential of resource provisioning in the Multi-Tenant Collaborative environment.

As it can be seen, on average, the Single-Tenant Collaborative (FCFS), Multi-Tenant Non-Collaborative (FCFS), Multi-Tenant Collaborative (FCFS), and the Multi-Tenant Collaborative (GMK) presented 1.17x, 1.96x, 2.39x, 5.08x performance gains over the ST Non-Collaborative, and 1.20x, 1.40x, 1.77x, and 9.41x energy gains, respectively. The Single-Tenant Collaborative brought up to 1.43x and 1.48x performance and energy gains. The use of Collaborative execution enables the offload of tasks to the CPU, resulting in higher RLP and fewer FPGA reconfigurations for each tenant's workload. When only multitenancy is considered (Multi-Tenant Non-Collaborative scenario), more tasks can be addressed in the same Task Arrangement, enabling more efficient use of the resources. Combining both approaches (Multi-Tenant Collaborative) brought gains in most scenarios, even when using the naive FCFS provisioning strategy. When considering a more sophisticated provisioning strategy, the Multi-Tenant Collaborative environment can reach up to 13.22x gains.

In a collaborative environment, the target architecture and the workload characteristics have a significant influence on the system's final performance/energy. The FCFS strategy is not adequate for these scenarios, as it only focuses on balancing the workload among both architectures. For instance, considering the K410T architecture, the use of FCFS resulted in poor performance and energy improvements for both Collaborative environments. As this architecture provides fewer FPGA resources (compared to the other architectures), more tasks with high FPGA acceleration and execution time were naively addressed to the CPU, resulting in a performance/energy downgrade. On the other hand, GMK considers task characteristics for resource provisioning, which explains the better exploitation of the Multi-Tenant Collaborative approach.

Experiment Final Considerations: This Section presented the advantages of bringing multitenancy over space-sharing collaborative architectures. As more tasks are available, higher resource utilization can be achieved. We also showed that efficient resource provisioning strategies could benefit from multitenancy to reduce the number of FPGA reconfigurations (i.e., when the GMK is optimized for performance) and bring higher request-level parallelism and resource utilization. Finally, each target architecture results in variant levels of energy and performance improvements as they offer different RLP and power consumption trade-offs (between CPU and FPGA).

5.3 RAHD's Resource Provisioning Adaptability

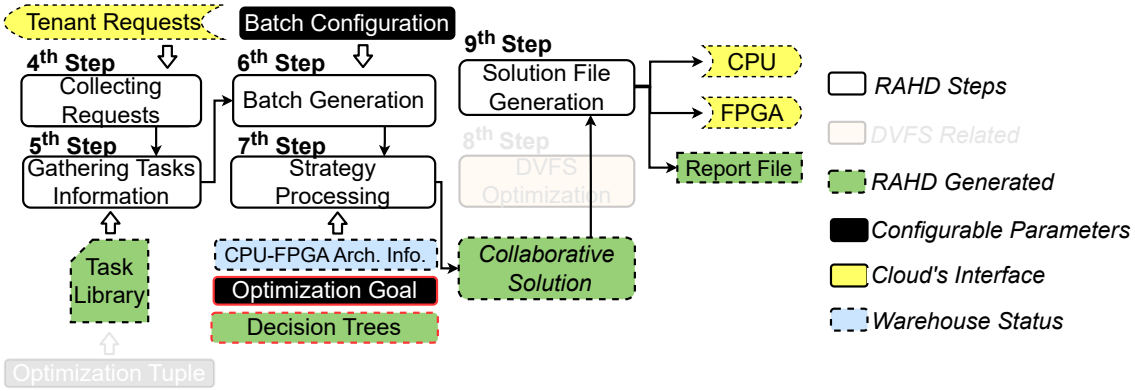
As previously discussed in Chapter 1 and shown in the work Jordan et al. (2021a), different resource provisioning strategies may be adequate depending on the architecture and workload characteristics. This Section shows us the importance of resource provisioning adaptability and evaluates RAHD's adaptive provisioning selection, performed according to the CPU-FPGA architecture and current workload.

5.3.1 Results Evaluation

This Section is divided into two parts. Section 5.3.1.1 shows that different provisioning strategies lead to variable resource provisioning solutions (i.e., in terms of performance/energy), which are more effective depending on the target architecture and workload characteristics. We also evaluate the strategies' convergence time to show the differences in their scalability over different batch sizes. By doing both experiments, we reinforce the need for an automatic solution that leverages multiple strategies in multi-tenant CPU-FPGA environments. Then, Section 5.3.1.2 compares RAHD adaptive provisioning against the use of single strategies and an Oracle in performance (given by the makespan metric) and energy.

For both Sections, we disabled the DVFS and HLS-Versioning. For Section 5.3.1.1, we considered the same features used in the previous experiments from Section 5.2 (shown in Figure 5.1) - adaptive provisioning is disabled. For Section 5.3.1.2, we enable the use of adaptive provisioning, as shown in Figure 5.3. As can be seen, the configurable parameter related to the selection of decision trees (i.e., Optimization Goal) was enabled, while

Figure 5.3: RAHD Online Stage (Section 5.3.1.2 evaluated features).



Source: The Author.

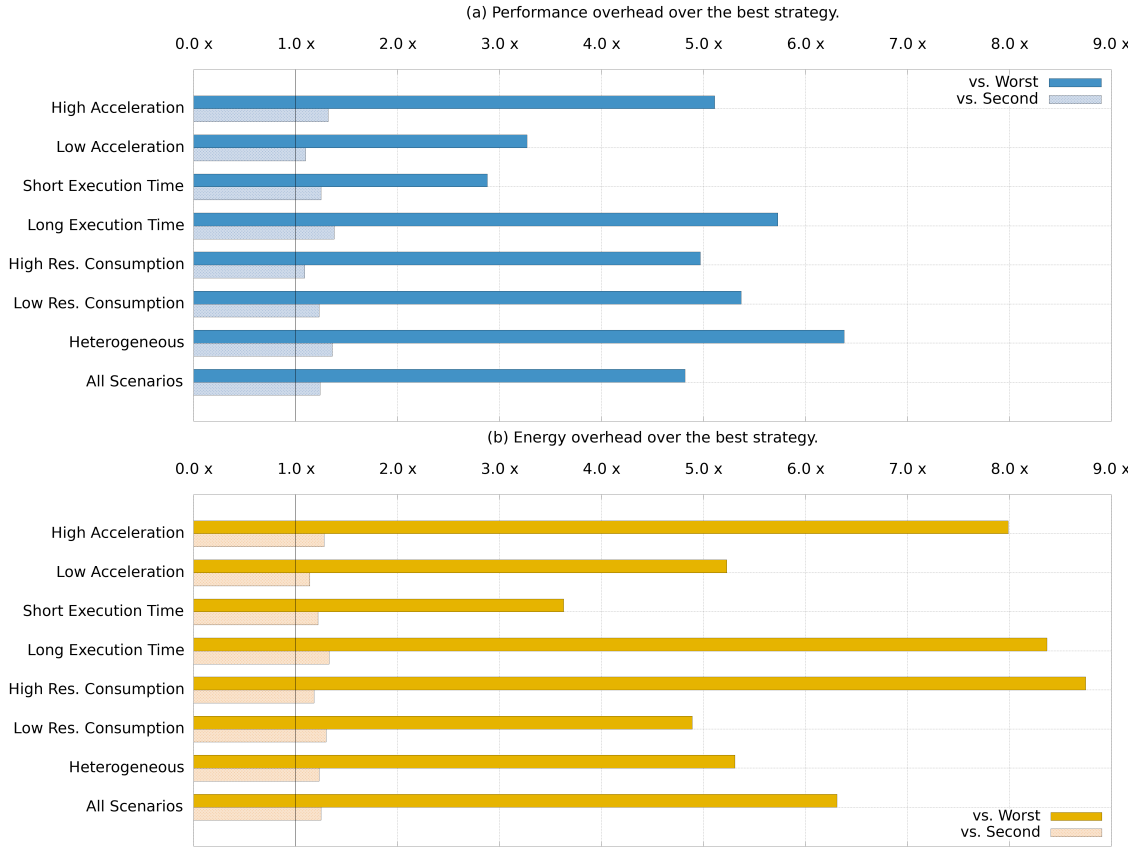
the configurable parameter related to the selection of HLS versions (i.e., Optimization Tuple) and DVFS-related steps were disabled.

5.3.1.1 Need for Resource Provisioning Adaptability

This experiment shows the need for multiple resource provisioning strategies depending on the workload and architecture. For that, we considered the execution of the 21.000 batches (i.e., from benchmark set 1) for all CPU-FPGA architecture combinations shown in Section 5.1, using the fixed execution of all available provisioning strategies (i.e., once a given strategy is set, it will be used throughout the whole execution). As aforementioned, this experiment does not consider the use of decision trees.

Figure 5.4 shows the impact of selecting a non-ideal strategy on performance (Figure 5.4 (a)) and energy (Figure 5.4 (b)) by comparing the best strategy (i.e., that provides the best solution in terms of energy and performance for a given batch) to the worst and the second-best allocation (the lower the value, the better). The results are filtered considering the evaluation scenarios presented in Section 5.1. As we can see, using the non-ideal

Figure 5.4: Performance and energy overhead of the worst strategy and the second best strategy over the best strategy.

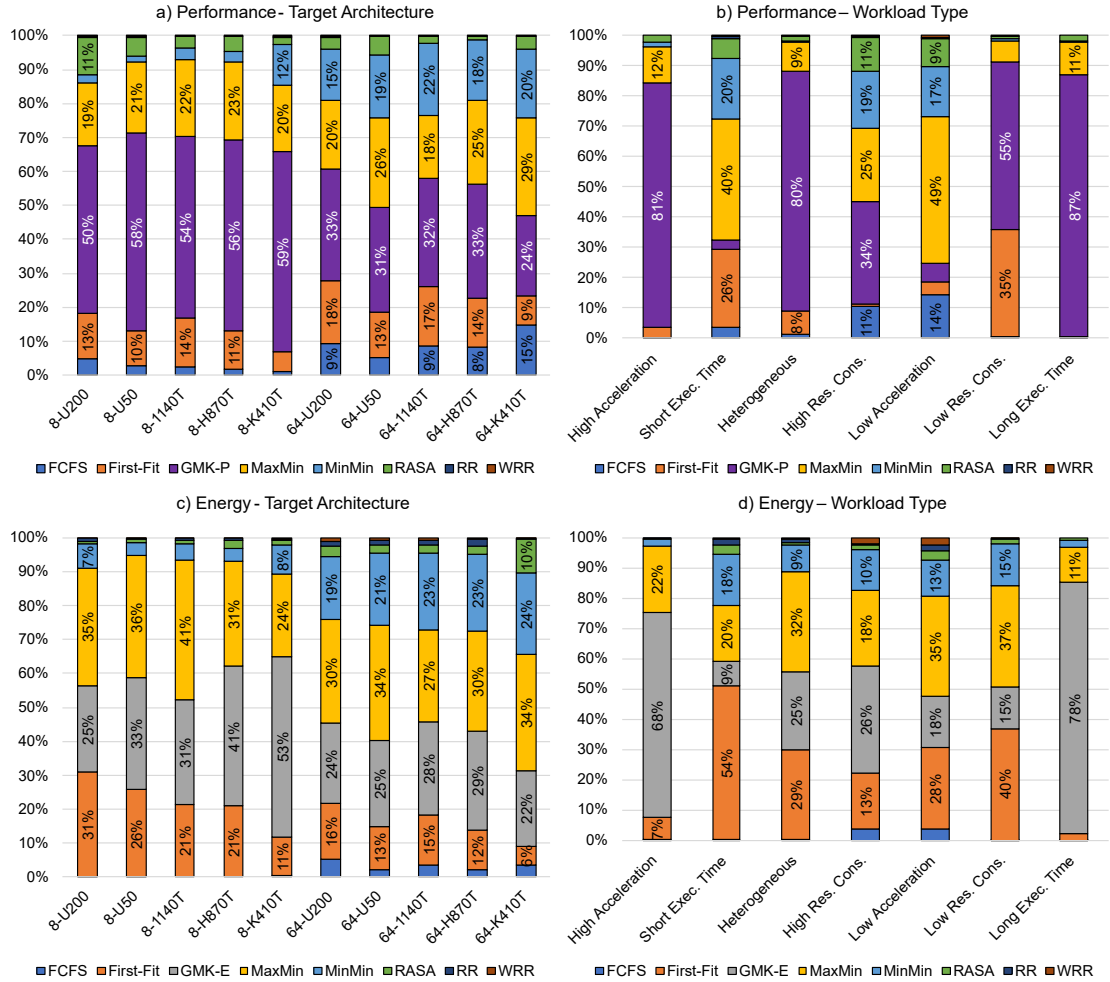


provisioning strategy can lead to performance losses ranging from 1.09x to 6.38x and energy increases ranging from 1.14x to 8.75x, showing the importance of selecting the most suitable strategy depending on the workload at hand.

Figure 5.5 shows the percentage of times a provisioning strategy produced the best solution for performance (top) and energy optimization (bottom) for different workload and architecture scenarios. The scenarios were filtered by workload type (right side) and target architecture (left side). For instance, in chart "a" column 8-U200 (architecture scenario - 8-core CPU, U200 FPGA), the GMK-P produced the best performance solution compared to other strategies for 50% (68% - 18%) of the batches from the scenario. Considering the same chart but column 64-K410T, the GMK-P produced the best performance solution for 24% of the batches.

Considering all scenarios (each column from the four charts), no single strategy

Figure 5.5: Percentages of times a provisioning strategy produced the best solution for performance (top) and energy (bottom) for different scenarios - target architectures (left) and workload types (right).



could provide the best solution for 100% of the batches in any scenario (different workloads/architectures and target optimizations - performance/energy). Moreover, different scenarios lead to a variant distribution of percentages. For instance, in chart "a" column 8-U200, the top-three strategies that produced more best solutions were GMK-P, MaxMin, and First-Fit (50%, 19%, and 13%, respectively). In contrast, for the same chart but column 64-K410T, the top-three strategies were MaxMin, GMK-P, and MinMin (29%, 24%, and 20%, respectively), showing the complexity of the problem. This distribution of percentages also varies when optimizing for performance or energy (comparing "a" vs. "c" and "b" vs. "d").

Regarding the target architectures ("a" and "c"), considering our less robust system (8-core and K410T FPGA), we can notice that GMK is the most suitable strategy, as it has a more complex profit model that considers resource consumption in its allocations, extracting Task Arrangements with good request level parallelism even in more strict architecture scenarios. For systems that offer more parallelism on the CPU side, strategies like MaxMin and MinMin dominate the best solutions, as they can fully exploit the parallelism offered by the CPU device but in a shorter convergence time (i.e., the time needed for a given strategy to reach a solution) compared to GMK.

Considering workload type (charts "b" and "d"), for the "Heterogeneous" workload, the strategy that presented the best energy solution for most batches was MaxMin (chart d - bottom right). The same could not be stated regarding the best performance solutions (chart b - upper right), as MaxMin could only achieve the best solutions in 9% of the batches for this scenario. This evaluation shows us that a solution with the best performance may not lead to the most energy-efficient solution, as tasks may perform better in a particular architecture that can hugely vary in power dissipation.

We also noticed that both genetic approaches (GMK-P and GMK-E) presented the best solutions for only a few batches for performance and energy scenarios for the "Short Execution Time" workload. In this workload type, fast heuristics are recommended, as they can produce solutions in a short convergence time.

For instance, in high-demand cloud environments, the strategy convergence time becomes relevant, as the time for allocation processing is a severe constraint. Moreover, depending on the batch, some strategies are unfeasible as their time to produce a solution may be way longer than the effective execution of the batch (for example, in the "Short Execution Time" workloads). Therefore, we discuss the convergence time of the strategies with different batch sizes and how each one scales as the batch size increases. For that,

Table 5.4: Strategies convergence time.

Batch Size	20	40	80	100	200
GMK-P/E	1.47s	2.35s	6.19s	9.57s	20.91s
RASA	0.01s	0.01s	0.02s	0.02s	0.04s
FCFS/RR/WRR	∼0.01s	∼0.01s	∼0.01s	∼0.01s	∼0.01s
First-Fit/MaxMin/MinMin	∼0.01s	∼0.01s	∼0.01s	∼0.02s	∼0.02s

we experimented with batches containing 20, 40, 80, 100, and 200 task requests. Convergence time is given by the arithmetic mean of the algorithm convergence time when executed in all evaluation systems.

As shown in Table 5.4, GMK-P/E was the slowest strategy, producing solutions up to a hundred times slower than the other strategies. However, those strategies can produce near-optimal solutions for several scenarios (as presented in the next Section). For larger computing problems, strategies with good scalability can deliver collaborative solutions in a short time, even when the batch size is increased. They are essential in scenarios of high task request frequency and of batches with short execution times.

Therefore, resource provisioning must not only prioritize high performance and energy efficiency but also ensure that these objectives are achieved in a feasible time.

5.3.1.2 RAHD's Adaptability Evaluation

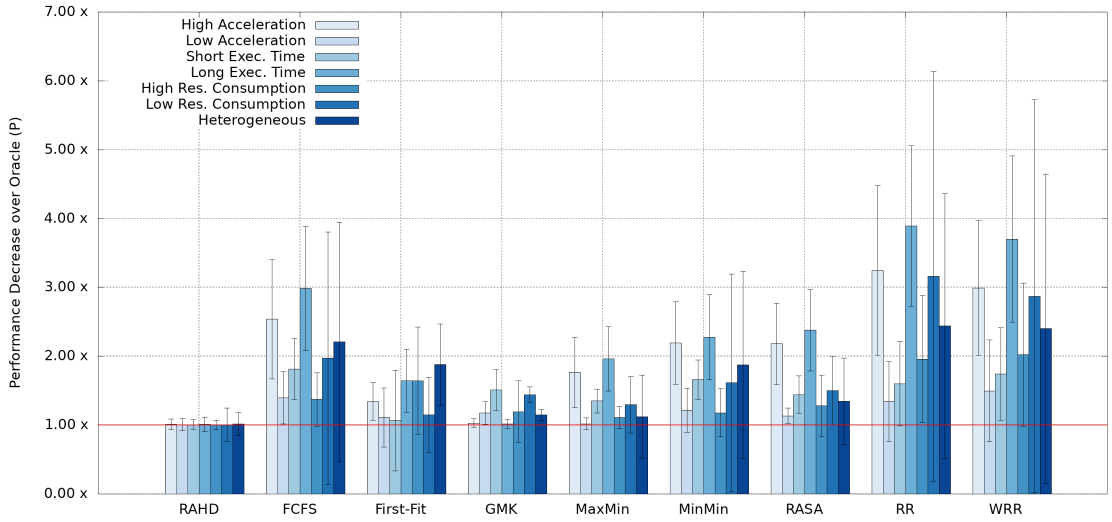
In this Section, we evaluate RAHD's capability to automatically explore the multiple provisioning strategies presented in the previous Section. In this way, these experiments evaluate RAHD adaptive provisioning selection (i.e., use of the decision trees), as shown in Figure 5.3.

For that, we compare RAHD (in performance and energy) with the fixed execution of all the other available provisioning strategies (i.e., once a given strategy is set, it will be used throughout the whole execution), considering the 21.000 batches executed over all the CPU-FPGA system combinations. The batches used in our experiments were not employed in the decision trees' training process.

Performance Evaluation. Figure 5.6 compares the performance of RAHD and fixed provisioning strategies over an Oracle that always chooses the strategy that delivers the best performance solution for each batch (Oracle (P) - best dynamic arbiter). To produce the Oracle, we exhaustively tested each provisioning strategy for each evaluated batch and took the strategy with the best results.

By comparing RAHD to an Oracle, we aim to show the effectiveness of our Decision Trees. The comparison with the fixed execution of all provisioning strategies shows

Figure 5.6: Performance decrease of RAHD and fixed strategies over the Oracle (P).



us the advantages of RAHD adaptability. It is important to notice that the time RAHD takes to select the strategy (i.e., decision tree convergence time) and the strategies' convergence time are both taken into account. For the Oracle, we only consider the strategy convergence time. As we evaluate performance, we configured RAHD (i.e., Optimization Goal configurable parameter) to use the Performance Decision Tree. The values indicate how close each strategy solution is to Oracle (the lower the value, the better).

Load-balancing algorithms, like FCFS, RR, and WRR, presented the worst results. These algorithms do not consider task characteristics when assigning tasks to the architecture, which may lead to low acceleration and high resource consumption tasks being assigned to the FPGA, producing FPGA Configurations with poor task parallelism and poor acceleration. These strategies can still be considered for particular batches as they present a fast convergence time. First-Fit also produces solutions quickly but prioritizes FPGA execution for tasks with high acceleration, producing good solutions for "Short Execution Time" workloads.

The GMK-P had the best overall results among fixed strategies but led to performance decreases of more than 1.2x in three workload scenarios. For instance, it achieved a 1.51x performance decrease in the "Short Execution Time" scenario, as it takes a huge convergence time to produce a solution compared to other strategies. The GMK-P also presented a 1.18x performance reduction in the "Low Acceleration" workload. As most tasks present poor/no acceleration when executed on the FPGA, this behavior increased the total execution time. In this scenario, MaxMin could achieve a 1.02x performance reduction w.r.t Oracle, as it allocates the most significant tasks to the FPGA and maximizes the task parallelism of the CPU by allocating less significant tasks to the CPU without

affecting the Task Arrangement makespan. However, the fixed use of MaxMin led to performance reductions of up to 1.96x.

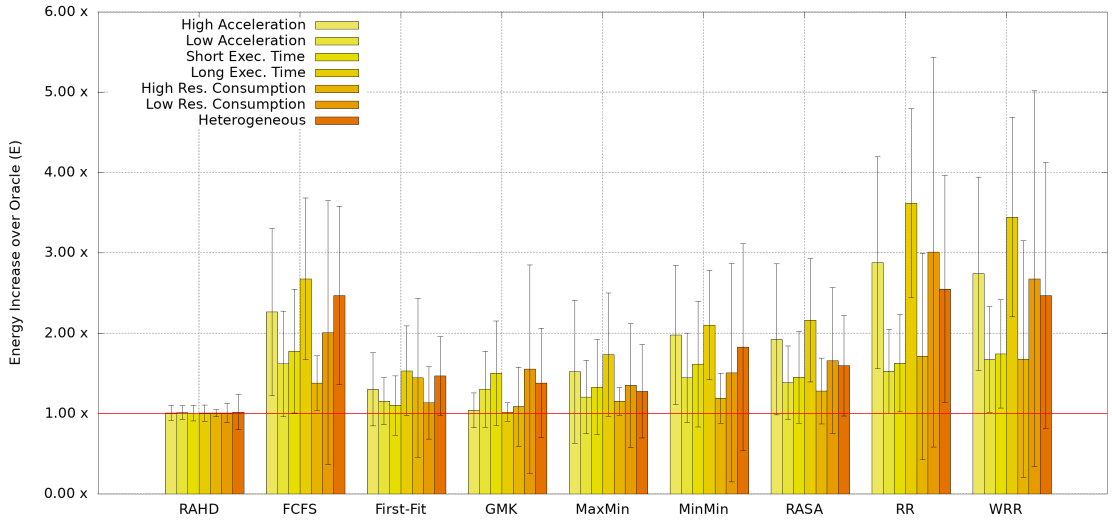
Considering all workload scenarios, RAHD, tuned for performance in this experiment, presented, on average, just a 1.01x performance reduction over the Oracle (P), achieving the best performance results for all workloads. For instance, if the best-fixed strategy were considered (GMK-P), performance reductions of up to 1.51x would be presented, mainly since this strategy takes huge convergence time to produce a solution in scenarios where the Cloud demand is high (for example, batches with a large number of task requests to be processed). In short, RAHD leverages efficient Decision Trees to select the most suitable provisioning strategies, keeping low-performance degradation compared to Oracle. This shows that multiple strategies and a dynamic selection framework are mandatory.

Energy Evaluation. In this Section, we repeat the previous experiments, but now we consider energy consumption. Figure 5.7 shows the provisioning strategies' energy over an Oracle that always chooses the strategy that delivers the best energy solution for each batch (the lower the value, the better), using the same methodology of the last experiment. The energy consumed for RAHD to converge to a strategy and due to strategies' convergence time is considered, while the energy taken for the Oracle to converge to a strategy is not considered.

As expected, FCFS, RR, and WRR presented the worst results in terms of energy. First-Fit also presented poor energy results for most scenarios. However, its fast convergence guarantees acceptable solutions for the short convergence time scenario. It could also be considered for the "Low Resource Consumption" one, as it can address more accelerable tasks in the same FPGA Configuration, leading to more tasks being addressed in parallel to the FPGA and fewer reconfigurations. MaxMin's best energy results were given by scenarios where higher performance improvements were achieved ("Low Acceleration" and "High Resource Consumption" scenarios). Overall, strategies focused on performance reduction (like RASA, MaxMin, and MinMin) presented high energy overhead in several scenarios, as they maximize FPGA and CPU utilization.

RAHD tuned for energy savings achieved the best energy results for all scenarios. Although GMK-E as a fixed strategy resulted in an energy increase of just 1.04x and 1.02x compared to Oracle (E) for the "High Acceleration" and "Long Execution Time" workloads, it resulted in energy increases of more than 1.3x in four workload scenarios. GMK-E always prefers providing allocations to the FPGA, only addressing tasks to the

Figure 5.7: Energy increase of RAHD and fixed strategies over the Oracle (E).



CPU when they do not affect the energy of a Task Arrangement. However, depending on the scenario, it can result in many FPGA Configurations with poor task parallelism, affecting the overall performance and energy consumption. Moreover, not exploiting CPU parallelism when highly available can lead to substantial performance penalties and, consequently, energy consumption overhead. Furthermore, the GMK-E suffers from high convergence time, which also harms the overall energy consumption. On average, RAHD presented 27% energy gains over GMK-E, which was the best single strategy for energy.

Experiment Final Considerations. This Section concludes that in CPU-FPGA Cloud environments with variant workloads and architectures, the use of existent single provisioning strategies will not provide the best solutions in terms of performance and energy for all scenarios for the following reasons.

- First, different CPU-FPGA architectures can hugely vary regarding CPU and FPGA task parallelism opportunities. Some strategies can provide better task parallelism exploitation in a specific architecture for a given workload;
- Second, task designs demand physical space on the FPGA, different from the CPU. Hence, the indistinct assignment of tasks to the FPGA results in few tasks being executed in parallel over the FPGA and a high number of FPGA reconfigurations, which incurs in time overhead;
- Third, the strategy convergence time plays an essential role in this environment. As shown in the experiments, even though bringing advantages over long execution time batches, strategies like GMK result in huge penalties for short execution time ones (e.g., in some cases, their convergence time is longer than the naive execution

of the batch);

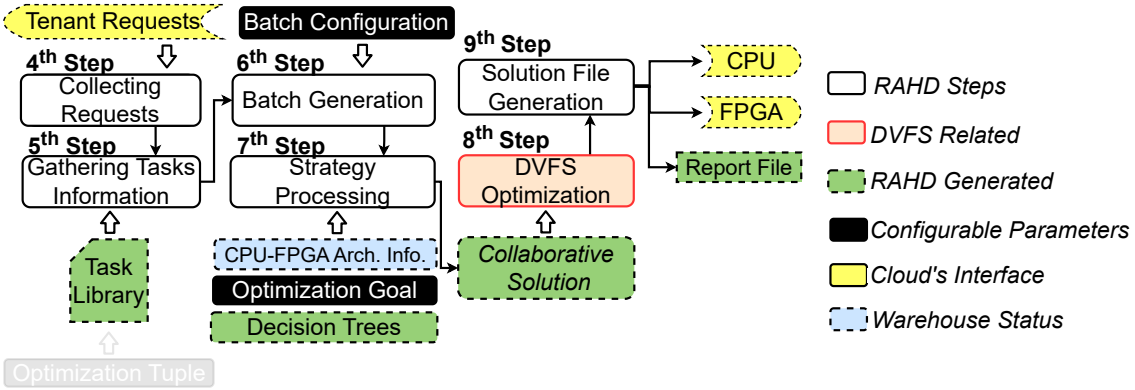
- Fourth, using energy-oriented and performance-oriented Oracles showed us that the best strategy for performance does not necessarily lead to the best energy solution, as both architectures vary in power dissipation. In this way, avoiding allocating tasks to the power-hungry architecture may be beneficial for energy improvements;
- Finally, our decision trees enable the dynamic selection of the best provisioning strategy for each batch, depending on the workload and the CPU-FPGA architecture characteristics. By using this process, we enable the cloud administrator an easy alternative to inserting, on-demand, any provisioning strategy to cover even more workload behaviors.

5.4 RAHD's DVFS Evaluation

This Section studies the effects of CPU DVFS in our scope. It also shows how RAHD can further increase energy benefits by synergistically employing the DVFS technique over Collaborative Solutions. The use of DVFS in collaborative CPU-FPGA environments was exploited by the author in Jordan et al. (2022).

For these experiments, we disabled HLS-Versioning. Figure 5.8 illustrates the RAHD Online Stage features evaluated in this Section. As can be seen, the RAHD 8th Step (i.e., DVFS Optimization) was enabled, while the configurable parameter related to the selection of HLS versions (i.e., Optimization Tuple) was disabled.

Figure 5.8: RAHD Online Stage (Section 5.4 evaluated features).



Source: The Author.

5.4.1 Results Evaluation

For these experiments, we consider the same methodology used in Section 5.3. Initially, we evaluate the impact of using the lowest frequency static DVFS level on performance and energy consumption. The lowest frequency static DVFS level sets a fixed, lowest frequency configuration for all CPU cores. Then, we present the energy improvements brought by the RAHD's fully-adaptive DVFS Optimization Step.

Need for DVFS Adaptability. DVFS has already been proven an effective way of reducing power consumption in CPU-based Cloud. However, if naively employed, it can result in performance degradation. Figure 5.9 shows the performance degradation on each strategy's and RAHD's solutions when using the lowest frequency static DVFS levels over the Oracle (P) (always selects the best resource provisioning for a given workload). We use the same methodology from the last sections.

As we can observe, all scenarios presented performance degradation, as employing DVFS unaware of the tasks executed on each core can hugely increase the latency of the ones responsible for a heavy burden, leading to the CPU execution time surpassing the FPGA execution time. Consequently, we can observe a huge unbalance between FPGA and CPU execution time. Using the lowest frequency DVFS levels over RAHD brought more than 1.5x performance decrease in five scenarios, as it hugely increased the execution time of the tasks assigned to the CPU (consequently increasing the makespan time).

Figure 5.9: Performance degradation when using the lowest frequency (red columns) static DVFS levels over the Oracle (P). Blue columns indicate the results without performance penalties (RAHD's DVFS).

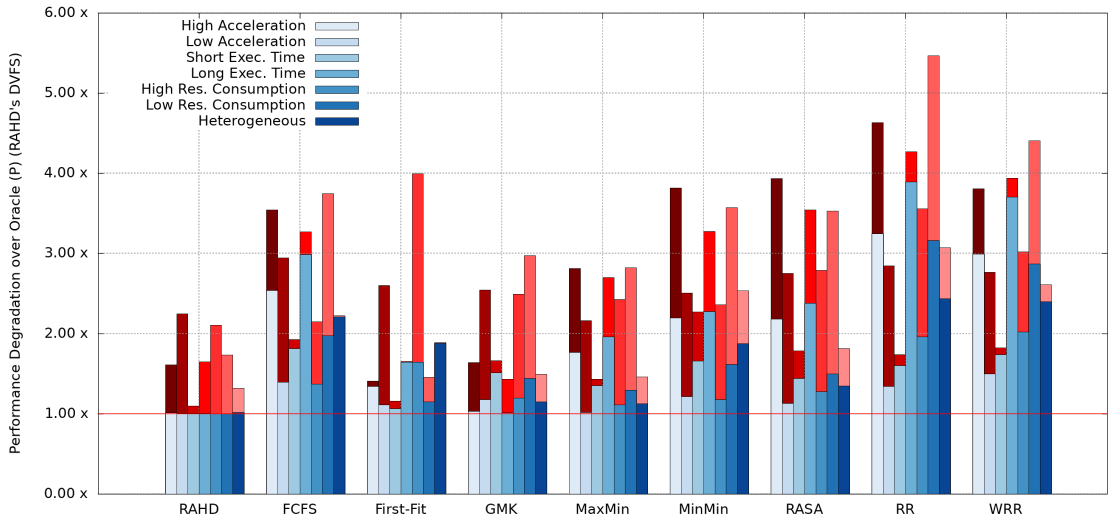
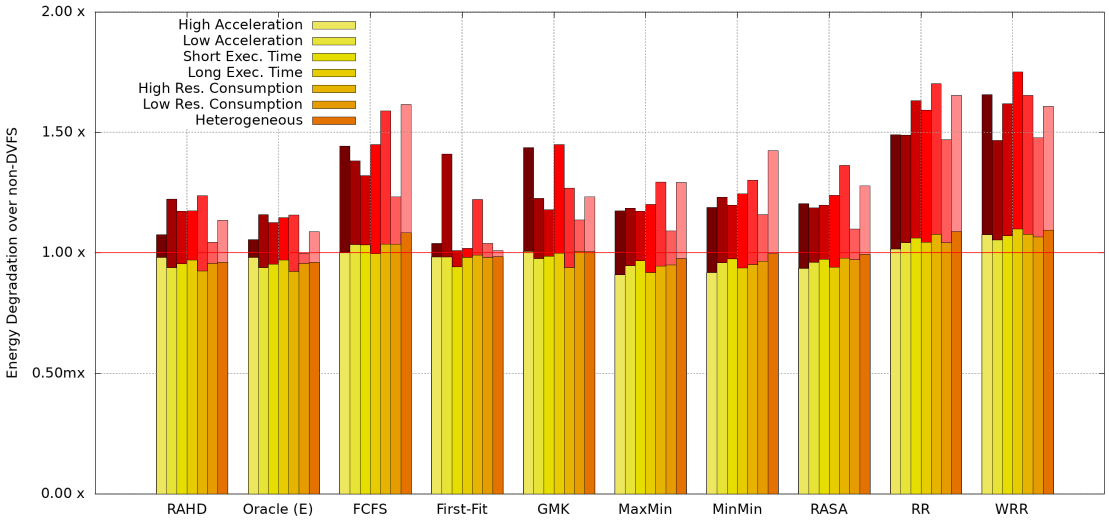


Figure 5.10 evaluates the energy degradation when using the lowest frequency and highest frequency static DVFS levels over RAHD, Oracle (E), and the fixed strategies without using DVFS (indicated by the red line fixed in 1.00x). The energy degradation considering the highest frequency DVFS profile is given by the yellow columns, while the energy degradation considering the lowest frequency DVFS level (i.e., lowest possible frequency) is given by the red columns (the lower the value, the better). For instance, considering the 3800x CPU, the lowest frequency DVFS level is given by 1.1GHz, while the highest frequency DVFS level is given by 3.5GHz.

Only a few scenarios presented energy benefits by using a static DVFS (values under 1.00x). By employing the static use of the highest frequency DVFS profile, little/no energy improvements are achieved. In some scenarios, this level of DVFS brings energy penalties. This energy degradation is noticeably higher considering the lowest frequency DVFS levels, bringing energy degradations of up to 1.85x.

Employing DVFS unaware of the workload on each core may lead to only a few cores being active most of the time, as most provisioning strategies can not fully use the CPU cores in several scenarios (e.g., due to a few remaining tasks for CPU execution). When only a few cores are active, the uncore power (i.e., power of CPU parts that are not in the core, like the L3 cache) dominates the CPU power consumption, which makes some DVFS profiles unprofitable. For instance, when a single core is active, the core power represents only 37% and 19% of the power consumption for the 3800x and 3990x, respectively. As RAHD's DVFS Optimization balances CPU and FPGA execution based on the FPGA execution time, it also ends up balancing the workload among CPU cores

Figure 5.10: Energy degradation when using the lowest (yellow columns) and the highest (red columns) static DVFS levels.



for most scenarios, bringing energy benefits. In future works, we also aim to use DVFS over the uncore. Moreover, different benchmarks are affected differently when DVFS is employed, which makes some of them not benefit from the lowest frequency DVFS levels even when the core power represents the largest power portion. As RAHD tracks all DVFS profiles, it avoids using DVFS levels that increase energy consumption.

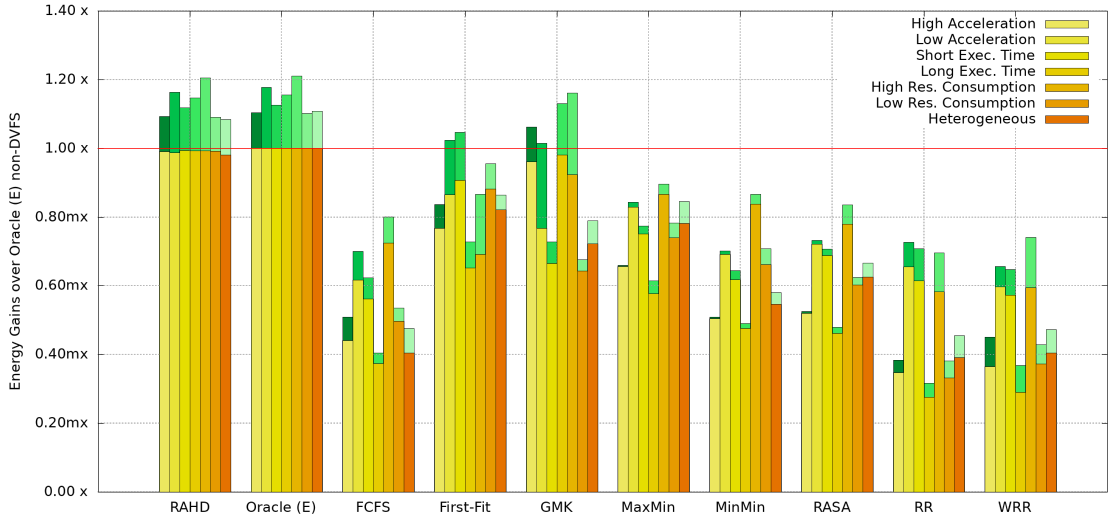
Finally, these results also point out that different batch types and strategies lead to different levels of degradation. Therefore, DVFS levels must be adapted considering the opportunities present in each Collaborative Solution (i.e., using the best DVFS levels for each Task Arrangement, as shown in Section 4.2.3).

RAHD DVFS Optimization Improvements. This Section evaluates RAHD's DVFS benefits by enabling RAHD's 8th Step. We focus only on energy evaluation, as our DVFS optimization does not affect the Collaborative Solution makespan (as shown in Section 4). Therefore, the performance follows the effectiveness of the provisioning without the use of CPU DVFS, which was already discussed in Section 5.3.1.2.

Figure 5.11 shows the energy improvements when using RAHD's DVFS optimization. We consider the Oracle (E) (always selects the best resource provisioning for a given workload) without DVFS optimization as the baseline. The yellow bars indicate the energy gains without DVFS, while the green bars indicate the DVFS gains considering the DVFS optimization. Results consider the average of the gains obtained for each of the 21.000 batches (presented in section 5.1).

As can be noticed, MaxMin, MinMin, and RASA do not present many improvements, as they focus on balancing the CPU and FPGA execution time, leaving little/no

Figure 5.11: Energy improvements with (green bars) and without DVFS (yellow bars) over Oracle (E) with no DVFS (the higher the value, the better).



space for CPU DVFS. Conversely, GMK-E achieved up to 1.32x improvements due to DVFS, as it distributes tasks to FPGA considering profit models, always respecting the makespan (FPGA-dominated in their case). Hence, the tasks assigned to the CPU have a shorter execution time than the FPGA tasks, creating more DVFS opportunities. At the same time, in the "High Resource Consumption" scenario, FF achieved the largest improvements due to DVFS (1.25x). Even though they present considerable improvements in some scenarios, RAHD achieves better energy results in all scenarios.

When considering the batch types, we note that DVFS shows the smallest improvements for "Short Execution Time" batches. Particularly, most tasks in this scenario present low execution time and low acceleration, which makes strategies like GMK-E avoid assigning unprofitable tasks to FPGA, generating Task Arrangements that exploit RLP over CPU. Consequently, there will be fewer DVFS opportunities, as most of our strategies tend to balance the workload among CPU cores, producing short execution time gaps between them.

The opposite happens for "Long Execution Time" batches, where most tasks are highly profitable for FPGA execution. For the "Low Resource Consumption" scenario, more tasks are assigned to FPGA. As fewer tasks are sent to the CPU, more DVFS opportunities emerge for each core. Conversely, gains are achieved in batches with high resource usage since more Task Arrangements are produced, resulting in more DVFS optimization opportunities. For instance, GMK-E produces good solutions for this scenario (composed of some high acceleration tasks), as it prioritizes execution on FPGA for most architectures, causing low CPU load and more DVFS opportunities.

Regarding the potential of our DVFS Optimization step, we state that DVFS improvements could be further improved for all provisioning strategies if more DVFS profiles were considered, as our step only considers employing DVFS over the CPU cores if the makespan is not affected. With more profiles, our approach has more options to balance the load between FPGA and CPU, which would result in more energy gains.

By combining RAHD's adaptive strategy selection with a synergistic DVFS, it produced up to 1.61x energy gains over GMK-E, the best-fixed energy strategy for energy. Compared to the Oracle (E) with no DVFS optimization, it presented up to 1.22x energy improvements when employing the DVFS technique. RAHD also presented results close to the Oracle (E) with CPU DVFS, resulting in only 2% of energy penalties. Therefore, besides optimizing allocation, RAHD also employs a DVFS optimization that further extracts energy reductions independently of the strategy without increasing the makespan.

Experiment Final Considerations. This Section explored the use of adaptive resource provisioning and DVFS optimization. The experiments showed us a space for applying DVFS when the Task Arrangement has an FPGA Configuration with a higher execution time than the CPU configuration. To exploit this CPU power technique, we dynamically adapt per-core DVFS levels to provide energy benefits without harming the solution's makespan (i.e., given by the FPGA Configuration). We also highlight that DVFS can be better exploited depending on the workload characteristics and the employed provisioning strategy.

5.5 RAHD's HLS-Versioning Evaluation

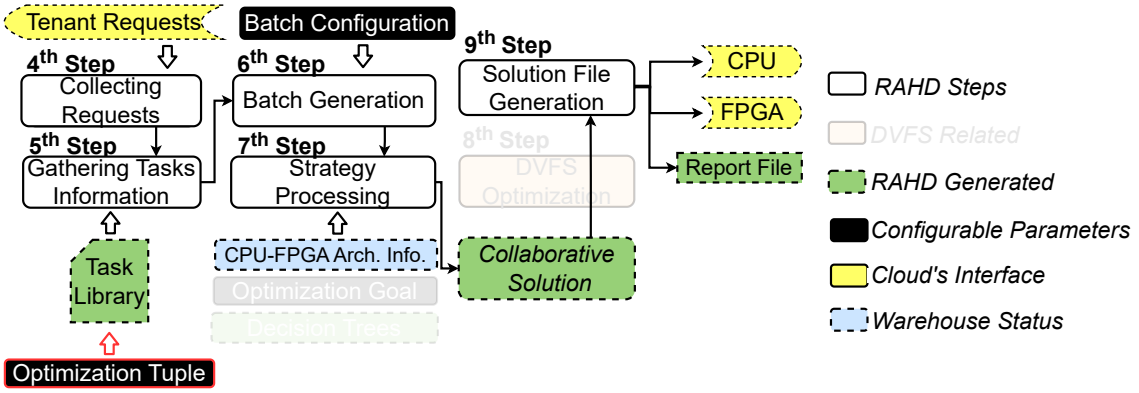
In this Section, we use RAHD to study HLS-versioning, which enables exploiting tasks with different performances, resources, and power consumption. The exploration of HLS-Versioning in collaborative CPU-FPGA environments was proposed in Lignati et al. (2021a). For these experiments, we disable DVFS and adaptive resource provisioning features. Figure 5.12 illustrates the RAHD Online Stage features evaluated in this Section. As can be seen, the configurable parameter related to the selection of HLS versions (i.e., Optimization Tuple) was enabled, while the configurable parameter related to the selection of decision trees (i.e., Optimization Goal) and DVFS-related steps were disabled.

5.5.1 Results Evaluation

For these experiments, we used benchmark set 2, presented in Section 5.1. For each task from this set, we generated multiple versions by using the following HLS pragma options - loop pipelining, unrolling, and array partitioning. Their optimization factors - initiation interval, unroll factor, and partitioning factor - were configured with the following values - 1, 2, 4, 8, 16, 32, 48, and 64. As our results focus on performance and energy optimization, we configured RAHD to select performance-oriented designs by using a full-performance tuple - (1,0,0) and energy-oriented designs by using a full-energy tuple - (0,0,1).

HLS-Versioning over individual tasks. First, we show the potential of RAHD's automatic HLS-Versioning generation at the Offline Stage (RAHD's 1st Step). Table 5.5

Figure 5.12: RAHD Online Stage (Section 5.5 evaluated features).



Source: The Author.

presents the performance and energy improvements, and resource consumption[1] overhead of performance-oriented and energy-oriented designs over the baseline (i.e., no HLS-Versioning). For all task designs, RAHD produced versions that presented gains compared to the baseline. When using the performance-oriented designs, RAHD achieved up to 249.74x performance and 139.79x energy gains over the baseline. Using the energy-oriented designs, RAHD presented up to 173.27x performance and 159.39x energy gains.

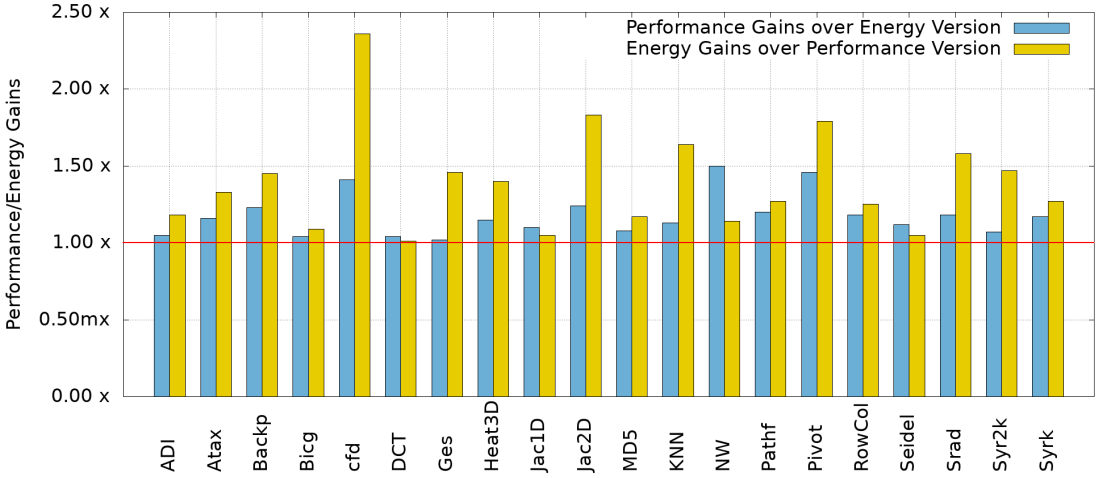
We can also observe that all performance-oriented versions consumed more resources than the energy-oriented ones, consequently leading to higher power consumption. This explains why the energy-oriented designs, even with higher latency, can still achieve higher energy efficiency than the performance-oriented designs. On average, the

[1]Resource consumption is given by the task design highest % of consumed resources over the FPGA (among BRAM/LUT/DSP/FF)

Table 5.5: Performance and energy improvements of performance-oriented and energy-oriented task design versions over no-directives version.

	ADI			Atax			Backpropagation			Bicg		
	P	E	R	P	E	R	P	E	R	P	E	R
Full-Performance	7.34x	4.47x	(-)9.92x	13.43x	6.05x	(-)43.12x	19.11x	8.42x	(-)10.20x	24.22x	18.46x	(-)7.38x
Full-Energy	7.01x	5.28x	(-)4.21x	11.46x	7.97x	(-)14.88x	15.57x	12.21x	(-)2.59x	23.28x	20.12x	(-)3.26x
	CFD			DCT			Gesummv			Heat3D		
	P	E	R	P	E	R	P	E	R	P	E	R
Full-Performance	12.81x	2.71x	(-)33.16x	16.31x	14.99x	(-)1.52x	10.12x	6.07x	(-)14.39x	7.74x	4.75x	(-)11.98x
Full-Energy	9.11x	6.40x	(-)3.02x	15.72x	15.14x	(-)1.31x	9.91x	8.86x	(-)3.27x	6.77x	6.65x	(-)1.36x
	Jacobi 1D			Jacobi 2D			KNN			MD5		
	P	E	R	P	E	R	P	E	R	P	E	R
Full-Performance	21.03x	16.69x	(-)6.57x	13.31x	4.58x	(-)55.26x	12.29x	10.51x	(-)31.74x	5.67x	3.98x	(-)5.71x
Full-Energy	19.06x	17.52x	(-)1.66x	10.76x	8.40x	(-)6.98x	10.86x	17.24x	(-)1.72x	5.27x	4.70x	(-)1.52x
	NW			Pathfinder			Pivot			RowCol		
	P	E	R	P	E	R	P	E	R	P	E	R
Full-Performance	249.74x	139.79x	(-)5.19x	2.39x	1.69x	(-)8.65x	3.53x	1.14x	(-)32.35x	10.01x	5.71x	(-)25.16x
Full-Energy	173.27x	159.36x	(-)3.13x	1.99x	2.15x	(-)6.48x	2.42x	2.04x	(-)1.54x	8.54x	7.19x	(-)4.10x
	Seidel			Srad			Syr2k			Syrk		
	P	E	R	P	E	R	P	E	R	P	E	R
Full-Performance	2.21x	1.87x	(-)3.59x	81.78x	21.88x	(-)25.14x	18.92x	7.86x	(-)42.24x	26.38x	15.35x	(-)22.60x
Full-Energy	1.97x	1.96x	(-)1.14x	69.49x	34.57x	(-)9.06x	17.76x	11.59x	(-)17.04x	22.56x	19.51x	(-)4.20x

Figure 5.13: Performance/energy gains of each version over their counterparts.



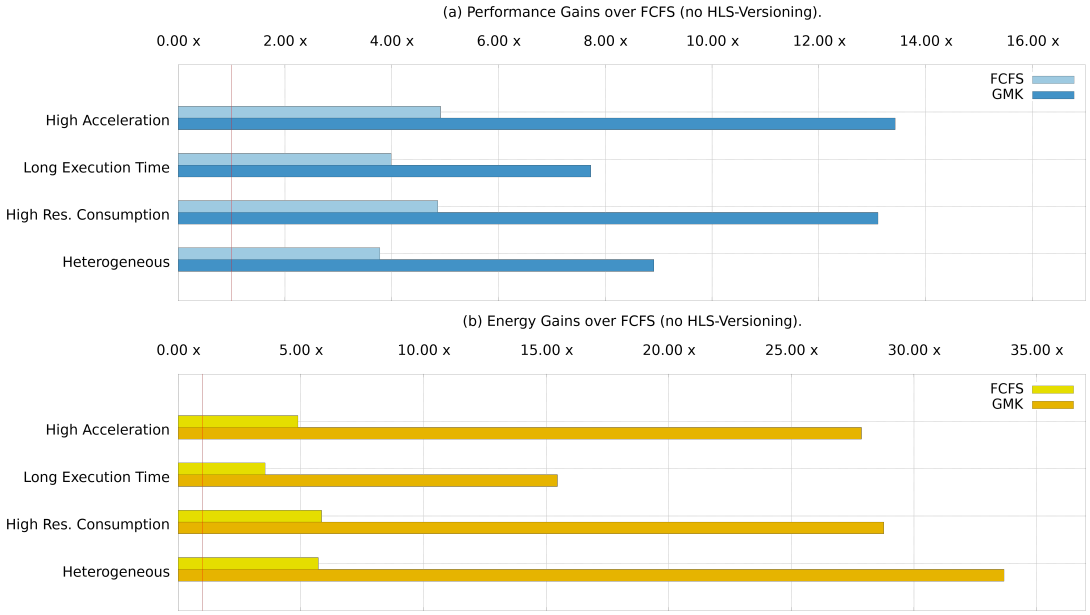
performance-oriented versions consume 19.79x more resources than the baseline, while the energy-oriented versions consume 4.62x more resources.

It is also important to highlight that all benchmarks presented versions with distinct energy/performance trade-offs, showing the effectiveness of our HLS-Versioning generation in producing specialized designs. Figure 5.13 depicts the performance gains of performance-optimized designs over the energy-optimized versions (blue bars) and the energy gains of each energy-optimized design over the performance-optimized version (yellow bars). Our results show that the performance-oriented designs present up to 1.50x performance improvements over the energy-oriented designs, while the energy-oriented designs present up to 2.36x energy savings over the performance-oriented designs.

Even though we observe huge benefits when using standalone specialized designs, these improvements come at different trade-offs in consumed resources, performance, and energy, which also affects the effectiveness of the provisioning strategies when considering multiple task requests. In the next Section, we evaluate the RAHD HLS-Versioning when executed over multiple task workload scenarios.

Potential of HLS-Versioning and Resource Provisioning over multiple tasks. Figure 5.14 shows the performance (a) and energy (b) improvements of HLS-Versioning combined with FCFS and GMK over the baseline (FCFS with no HLS-Versioning). The blue bars depict performance gains with performance-optimized designs over the baseline, while the yellow bars represent energy gains with energy-optimized designs over the baseline. For this experiment, we consider four workload scenarios - "High Acceleration", "Long Execution Time", "High Resource Consumption", and "Heterogeneous". We selected these scenarios as they pose a significant challenge for load balancing pro-

Figure 5.14: (a) Performance and (b) energy gains of FCFS and GMK over the baseline (FCFS with no HLS-Versioning).



visioning strategies such as FCFS, as previously shown in results from Section 5.3.1.2, and are also greatly influenced by HLS-Versioning. For instance, the tasks that belong to the "High Resource Consumption" scenario have their resource consumption further increased by HLS-Versioning. This results in a strict limit on the number of parallel tasks over the FPGA. At the same time, HLS-Versioning drastically reduces the execution time of "Long Execution Time" workloads and further increases the acceleration of tasks from the "High Acceleration" batch, offering huge optimization opportunities. On the other hand, the "Heterogeneous" scenario presents a challenge for effective resource allocation, as it represents a mix of workloads with varying requirements.

The results show that FCFS is not able to fully exploit the advantages of specialized designs or manage the resource overhead imposed by the HLS-Versioning technique. By combining FCFS with HLS-Versioning, RAHD achieved, on average, 4.39x performance improvements and 4.98x energy savings. In contrast, by combining GMK and HLS-Versioning, RAHD reached energy improvements of up to 33.67x, substantially outperforming FCFS in all scenarios.

Even though we could observe HLS-Versioning's potential, we notice that some strategies may not always extract the full benefits of optimized task versions, as a subtle change in the task's properties affects the provisioning strategy solutions. For instance, selecting energy-oriented designs can lead to a considerable deceleration of significant tasks in the workload, which can make some strategies assign more tasks to the CPU,

leading to higher energy consumption. On the other hand, high-performance tasks with high resource consumption can result in many Task Arrangements, reducing the task request parallelism, which can harm the produced solutions' performance. Strategies that do not consider resource consumption metrics can be hugely affected in this situation.

Experiment Final Considerations. This Section investigated HLS-Versioning advantages in our proposed framework. Our experiments showed us that, by exploring variant HLS pragma combinations, RAHD could generate profitable tasks to optimize different fronts (i.e., energy and performance). Moreover, combining HLS-Versioning and resource provisioning brings huge benefits. However, a subtle change in the task's properties alters the workload characteristics, affecting the provisioning strategy effectiveness. Therefore, we reinforce the importance of RAHD's automatic provisioning that selects the best strategy for a given workload, fully exploiting HLS-Versioning advantages.

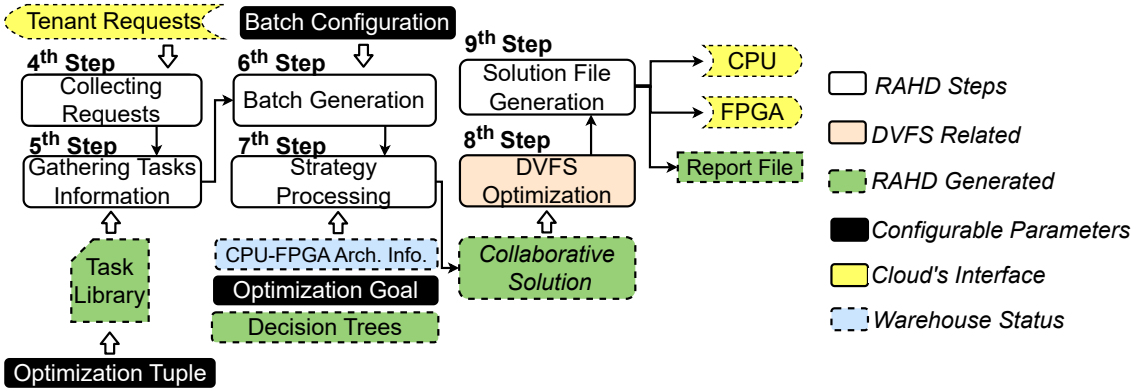
5.6 RAHD's Overall Evaluation

This Section evaluates the full benefits of RAHD, which comprises all optimization axes studied so far. Therefore, all framework configurable parameters and steps are enabled, as shown in Figure 5.15.

5.6.1 Results Evaluation

This experiment shows RAHD's capabilities when considering all optimization fronts. We divided this evaluation into three parts. First, we evaluate RAHD perfor-

Figure 5.15: RAHD Online Stage (Section 5.6 evaluated features).



Source: The Author.

mance and energy benefits. Then, we provide a scalability evaluation, showing RAHD advantages when considering variant batch sizes (i.e., configured through the Batch Configuration parameter). For both experiments, we consider no restriction in the number of containers. Finally, we evaluate RAHD in terms of performance and energy under a limited number of available containers.

For these experiments, we used the same benchmarks and HLS versions used in the last Section (i.e., from benchmark set 2). We also consider all target architectures and DVFS profiles introduced in Section 5.1. We used performance-oriented designs for performance evaluation and energy-oriented designs for energy evaluation.

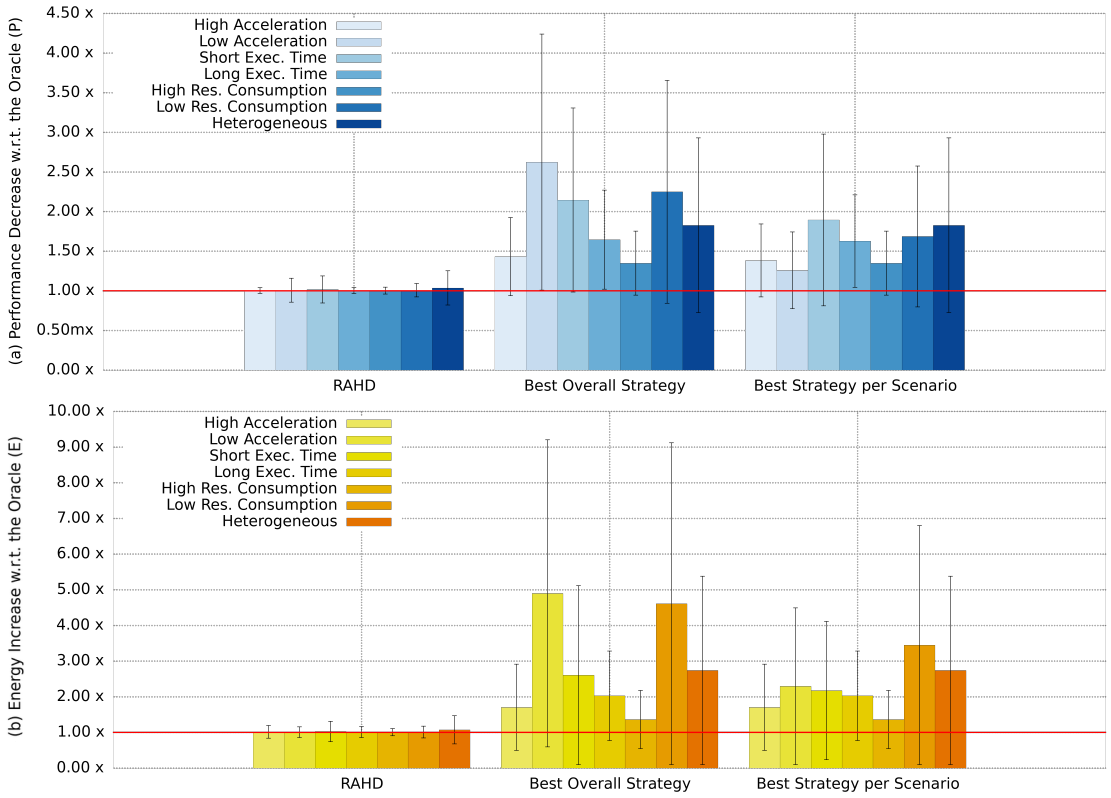
RAHD Overall Evaluation: These experiments evaluate RAHD in terms of performance and energy, considering all its optimization axes. For that, Figure 5.16 presents a performance (a) and energy (b) study of RAHD, the "Best Overall Strategy", and the "Best Strategy per Scenario" approaches (the lower the results, the better). The "Best Overall Strategy" approach outputs the results achieved by the standalone strategy that achieved the best average results considering all workload scenarios (i.e., for these scenarios GMK-P in (a) and GMK-E in (b)). The "Best Strategy per Scenario" approach outputs the results achieved by standalone strategies that presented the best results for each workload scenario. For instance, the best strategy for the "Heterogeneous" scenario in terms of performance was GMK-P, and the best strategy for the "Low Acceleration" scenario in performance was MaxMin. Both "Best Overall Strategy" and "Best Strategy per Scenario" approaches do not consider DVFS.

The performance results (blue bars) consider Oracle (P) as the baseline, while energy results (yellow bars) consider Oracle (E) as the baseline. The Oracle considers the strategy that produces the best results considering all optimization axes (i.e., DVFS included). To produce the Oracle, we exhaustively tested each provisioning strategy for each evaluated batch and took the strategy with the best results.

As can be noticed, only RAHD can effectively use HLS-Versioning and DVFS, as it always selects the provisioning strategies that fully extract the potential of the optimized designs. By evaluating the "Best Overall Strategy" and "Best Strategy per Scenario" cases, we notice that only HLS-Versioning itself will not leverage the full potential of the CPU-FPGA environment. Moreover, using a single strategy can not cover the high workload heterogeneity imposed by the Cloud and will not always take advantage of the optimized design versions, as can be observed in the makespan results.

Considering the performance results, RAHD achieved, at most, a 1.04x perfor-

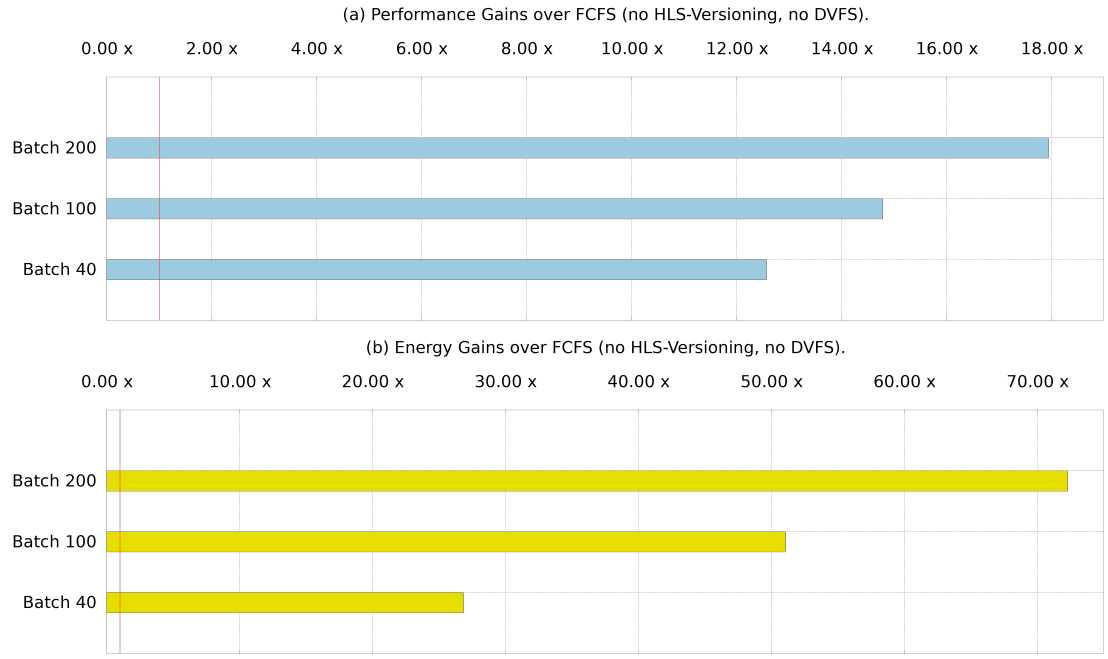
Figure 5.16: (a) Performance decrease over the Oracle (P). (b) Energy increase over the Oracle (E).



mance decrease compared to the Oracle (P), while the "Best Overall Strategy" and "Best Strategy per Scenario" cases decreased the performance by more than 1.50x for at least three workload scenarios. For the energy results, RAHD reached a 1.07x energy decrease compared to the Oracle (E) in the "Heterogeneous" scenario. However, the other scenarios presented average energy penalties of 2.85x and 2.25x, considering the "Best Overall Strategy" and "Best Strategy per Scenario" cases, respectively.

RAHD Scalability Evaluation: Now, we demonstrate RAHD's potential over standard provisioning and show how it scales when the batch size is increased, which is a relevant problem in our scope as Cloud demand is ever-increasing, resulting in more task requests being received in a short period, forcing large batches to be executed. Figure 5.17 shows RAHD's performance (a) and energy (b) results over the FCFS with no HLS-Versioning and no DVFS (No Optimization) considering variant batch sizes. The results indicate that the larger the batch size, the higher the performance and energy improvements. For large batch sizes with multiple task requests, RAHD has more opportunities to explore the highly profitable designs provided by our HLS-Versioning step. Moreover, large batch sizes usually result in collaborative solutions with several Task Arrangements, generating multiple DVFS opportunities. For energy optimization (b), RAHD achieved

Figure 5.17: (a) Performance and (b) energy gains of RAHD over the FCFS with no HLS-Versioning and DVFS.



more than 70x energy improvements over the baseline when considering the largest batch size. Even for small batch sizes, RAHD guarantees more than 20x energy gains over the baseline. RAHD also achieves significant performance improvements (a), reaching more than 10x higher performance than the baseline in all scenarios.

RAHD Evaluation over Limited Containers: At last, we evaluate how well RAHD performs when limiting the number of available containers, as the Cloud environment may have strict time and resources to produce them. Table 5.6 evaluates the performance and energy of RAHD with limited containers. The second column presents the scenarios used for performance comparison, while the fourth column presents the scenarios used for energy comparison. RAHD is compared against the following scenarios - Oracle (P) and Oracle (E) (Limited Containers), which always selects the strategy that delivers the best solution for the target metric *with limited containers*; Oracle (P) and (E), which always selects the strategy that delivers the best solution *without limit of containers* (same used in previous experiments); "Best Overall Strategy", "Best Strategy Per Scenario", and FCFS (No Optimization) all *without container limitation*.

For limited container scenarios, we have restricted the number of containers to 168, considering the methodology presented in Section 5.1 (i.e., 1 week to produce containers). The RAHD with limited containers was retrained for this experiment using the same methodology presented in Section 4.1.3, but with a training dataset that considers

Table 5.6: RAHD's evaluation with container limitation.

	Performance Comparison	RAHD (Limited Containers)	Energy Comparison	RAHD (Limited Containers)
1	Best Overall Strategy	1.49x	Best Overall Strategy	2.51x
2	Best Strategy Per Scenario	1.25x	Best Strategy Per Scenario	1.97x
3	FCFS (No Optimization)	11.98x	FCFS (No Optimization)	43.90x
4	Oracle (P) (Limited Containers)	(-) 1.02x	Oracle (E) (Limited Containers)	(-) 1.03x
5	Oracle (P)	(-) 1.26x	Oracle (E)	(-) 1.14x

the restriction in the number of containers.

As can be noticed, RAHD achieves results close to the Oracles with limited containers (Table 5.6 line four), being only 2% and 3% far from their energy and performance results, respectively. We point out that the accuracy of our produced decision trees dropped considering a limited number of containers - from $86\% \rightarrow 82\%$ for the performance tree and from $84\% \rightarrow 81\%$ for the energy tree). Although not always selecting the most suitable strategy, it still selects strategies that reach solutions close to the best ones.

On the other hand, the "Best Overall Strategy" and "Best Strategy per Scenario" approaches (Table 5.6 lines one and two), even with no container limitation, lag far behind RAHD with limited containers. In fact, RAHD achieved 1.25x and 1.97x performance and energy gains over the "Best Strategy per Scenario". Furthermore, RAHD, despite being limited in the number of containers, still outperforms the "FCFS (No Optimization)" approach by a significant margin, as seen in Table 5.6 (line three), with 43.90x improvements in energy efficiency.

Finally, when compared to the Oracle (P) without limit of containers (Table 5.6 line five), RAHD (Limited Containers) still achieves acceptable results, showing, on average, 1.26x performance degradation. This can only be achieved by RAHD FPGA Configuration leftover logic (explained in Section 4.2.4), which efficiently distributes remaining tasks among CPU cores, minimally harming the Collaborative Solution makespan. Furthermore, compared to Oracle (E), it achieves only 1.14x energy degradation. For the energy metric, the effects of container limitation are reduced as RAHD explores DVFS optimization over the leftover tasks dispatched to the CPU (explained in Section 4.2.4). As discussed in Section 4.2.4, RAHD benefits can be leveraged with time as more containers are made available (i.e., can be either scheduled for implementation in idle periods or dedicated machines/resources).

Experiment Final Considerations. Our experiments demonstrated that only RAHD has enough adaptability to extract the full benefits of collaborative provisioning, HLS-Versioning, and DVFS, also presenting good scalability as the size of the problem increases (i.e., the batch size). Furthermore, even in constrained scenarios (i.e., a limited

number of containers), RAHD still delivers substantial performance and energy improvements compared to fixed provisioning strategies (i,e., without container limitation). Next, we briefly discuss what we have learned from the experiments shown in this thesis.

5.7 Wrap-up

This Chapter explored RAHD to evaluate the benefits of adaptive resource provisioning, DVFS, and HLS-Versioning individually and symbiotically.

Section 5.2 showed us the advantages of using multitenancy over different CPU-FPGA nodes by increasing resource utilization. It also highlighted the importance of collaborative resource provisioning (to optimize performance and energy further), also revealing that different target architectures may present different levels of improvement.

Section 5.3 explored the benefits of an adaptive selection of the most suitable provisioning strategy for a given task request batch, showing that different strategies are needed depending on the target architecture and workload characteristics. The experiments provided an in-depth analysis of RAHD provisioning strategies and detected the main workload and architectural characteristics that affected the solutions, like their overall acceleration, execution time, and resource-level parallelism. We also highlight that some strategies, although bringing efficient results for several scenarios, may present long convergence times. The use of those strategies for short execution workload scenarios is unfeasible. These experiments taught us the importance of adapting the provisioning strategy for incoming workloads. For that, we proposed a decision tree classification method that detects the best provisioning strategy for each batch to either maximize performance or minimize energy, presenting similar solutions compared to an Oracle.

Then, Section 5.4 explored DVFS optimization over several resource provisioning strategies and RAHD (i.e., without HLS-Versioning). The experiments showed us that DVFS must be adapted synergistically to provide energy benefits without harming the solution's performance and energy. After, Section 5.5 explored only HLS-Versioning in the proposed environment. The experiments showed us that RAHD could generate design versions that considerably boost the energy and performance of individual designs. Moreover, we reinforced the importance of RAHD dynamic provisioning when using HLS-Versioning over multiple task requests, as a subtle change in the selected designs (e.g., performance-oriented or energy-oriented) can affect the workload properties, also impacting the provisioning strategy effectiveness.

Finally, Section 5.6 evaluated RAHD as a whole, showing that it efficiently bridges the gap between HLS-Versioning, DVFS, and adaptive resource provisioning. Our experiments showed that RAHD could extract the benefits provided by each optimization while never reducing each other effectiveness.

6 CONCLUSIONS

This work has presented RAHD, a Resource Provisioning Framework for CPU-FPGA Environments with Adaptive HLS-Versioning and DVFS, which symbiotically explores adaptive resource provisioning, HLS-Versioning, and DVFS.

RAHD was born as a framework to explore collaborative computing and multi-tenancy in CPU-FPGA AaaS Cloud, where the main challenge was to investigate the potential of variant provisioning strategies in this scope. This study led us to observe a lack of specialized resource provisioning methods for the niche, driving us to adapt standard Cloud provisioning strategies to our scope (i.e., FCFS, First-Fit, MaxMin, MinMin, RASA, RR, WRR) and develop the Genetic-Multidimensional Knapsack (GMK), which was modeled considering several aspects of the problem, like the task and target architecture properties. Although achieving important advantages in several scenarios compared to traditional strategies, the GMK could not cover all workload behaviors and also presented disadvantages due to its long convergence time.

This first study push our investigation to detect the factors that affected the efficiency of the provisioning strategies. We concluded that workload properties, target architecture, optimization goals (e.g., energy and performance), and strategy convergence time were the key factors. This observation showed us a need for resource provisioning adaptability, as these factors dynamically and drastically vary in the Cloud scope. Therefore, we developed a solution capable of detecting the most suitable strategies for a given workload. Our solution relied on a decision tree classification method that could either select strategies for best performance or energy results. This method could be constantly upgraded with new provisioning strategies, also opening up space for new optimization metrics.

Even though reaching a dynamic and end-user transparent resource provisioning, we observed that the CPU-FPGA environment potential could be further exploited by leveraging each device's intrinsic capabilities. On the CPU side, DVFS was an alternative to improve energy efficiency further. Therefore, we investigated the influence of this optimization technique alongside our resource provisioning method. At first glance, the traditional use of DVFS (i.e., through static DVFS levels) revealed a negative impact on our solution's performance and little energy improvements for some DVFS levels. Therefore, the challenge was to find a way of inserting this technique into our framework without affecting the advantages brought by our resource provisioning. For that, we de-

signed a DVFS optimization step capable of leveraging per-core DVFS without affecting the allocation makespan.

As we produced a synergistic solution that involved CPU optimization, there was still a need to better explore the FPGA side. We found this opportunity on HLS-Versioning, which opened up space to explore designs optimized for different optimization targets. Here, the first challenge lay in producing specialized task designs. For this challenge, we developed an automatic approach that explores multiple HLS pragma optimizations to generate designs with variant task properties. The second challenge was to take advantage of these variant designs at runtime. We found this answer through an optimization tuple search algorithm that enables the cloud administrator, at runtime, to select the most suitable designs based on the desired optimization target (i.e., performance/energy/area). The last investigation was to check the impact of the variant designs on our resource provisioning method. From this study, we detected that selecting different designs would affect the workload properties, also affecting the provisioning strategy produced solutions. This conclusion reinforced the significance of our decision trees, which could handle the variant workload behaviors, selecting the strategies that could leverage the advantages of a particular design.

The union of these optimization axes consolidated RAHD, which extracts the benefits of each technique without lessening each other effectiveness. Our results point out that RAHD can achieve energy and performance with only 7% and 4% worse than an Oracle, which always selects the best resource provisioning for a given workload. Finally, we conclude that only the synergistic exploitation of CPU and FPGA optimization techniques and resource provisioning can extract the maximum potential of CPU-FPGA Cloud environments, which is only provided by RAHD.

6.1 Limitations

This Section highlights the constraints and boundaries of this thesis.

Adopted Execution Model. Due to time constraints, this work embraces only the OpenCL execution model, producing provisioning solutions for scenarios where the FPGA is fully reconfigured with bitstream containers, which can be composed of one or multiple task designs. In future works, we aim to explore Dynamic Partial Reconfiguration as this technique can bring huge flexibility for multiple-task execution.

Regarding HLS-Versioning Exploration. This research limited the multiple-

design version generation in frequency and processing loop sizes due to time constraints. All tasks were synthesized with the same target frequency, and loop sizes were not varied during HLS-Versioning. We have limited our scope due to time constraints (see more in Section 4.1.1.3).

Optimization Techniques. As aforementioned, the focus of this work lies in exploiting DVFS on the CPU. We understand that power gating could also be considered. However, it requires an in-depth analysis of involved latencies (e.g., wake-up delay) and tools that could not be covered due to time constraints. Other power optimization techniques such as thread throttling were already explored in publications related to this proposal. The use of these techniques in future works will be discussed in the next Section.

Other Optimization Objectives. This proposal currently focuses on makespan and energy. However, other objectives are useful for the Cloud area, like optimizing cost and quality of service. The use of other optimization metrics in future works will be discussed in the next Section.

6.2 Future Works

This thesis has opened many opportunities for future work. In the scope of FPGA optimizations, we still glance at opportunities to find higher DSE in terms of optimized designs. For that, a great possibility lies in exploring designs with variant frequency. Recent FPGA models and tools already support setting tasks with multiple frequency levels at the same FPGA Configuration, which unlocks the possibility for us to expand design versions and control at finer granularity the FPGA performance and energy consumption. Furthermore, other FPGA techniques, such as FPGA overlay (SILVA et al., 2019; LI; MASKELL, 2019), could be potentially adopted to bring additional flexibility to our scope.

In the context of CPU optimization, we are currently investigating DVFS alongside DCT (Dynamic Concurrency Throttling) (SCHWARZROCK et al., 2021) in our scope. Collaborative computing and DCT were already explored by the works related to this thesis in (KNORST et al., 2021b) and (KNORST et al., 2021a). However, both alternatives were never explored alongside CPU-FPGA collaborative computing. By using both possibilities, we could better control the execution of multi-thread applications. In this sphere, the challenge sits in finding DVFS configurations that, combined with DCT profiles, can exploit the potential of each task without lessening task request parallelism

and affecting FPGA execution.

We also aim to explore uncore DVFS to further improve energy benefits. However, in contrast to per-core DVFS, few OS tools enable managing uncore frequency and voltage scaling, and most of them enable configuring these parameters in a coarse granularity. Currently, the technique is usually employed statically, using BIOS configurations. Other techniques, like power gating, which has been employed in CPU-based Clouds, are also the target of future works.

With respect to the explored architectures, we target expanding our provisioning over other systems, like CPU-GPU, multi-GPU, multi-FPGA, and recent architectures, such as Intel Alder Lake heterogeneous multi-core. We believe that these systems may bring new opportunities to accelerate application behaviors with efficiency, and the exploitation of their particular optimization techniques brings new challenges to our scope. Moreover, we work towards studying the correlation between intra-node (i.e., current work) and inter-node provisioning and how we can integrate both approaches to maximize each other effectiveness, considering multiple heterogeneous architectures (i.e., the reality of current Cloud warehouses).

We also aim to cover other optimization metrics (i.e., other than makespan and energy). In the Cloud area, cost optimization (i.e., maximizing Cloud profit) and quality of service (i.e., serving a tenant request under a given time budget) are also the focus. Our future works also investigate provisioning strategies focused on both metrics and how DVFS and HLS-Versioning can be employed with minimum or no penalties to the target optimizations.

6.3 List of publications

6.3.1 Publications related to this thesis

The following works have been published/submitted in peer-reviewed venues:

- **Jordan, M. G.**, Korol, G., Rutzig, Knorst, T., M. B., & Beck, A. C. S. (2023). Energy-Aware Fully-Adaptive Resource Provisioning in Collaborative CPU-FPGA Cloud Environments. Journal of Parallel and Distributed Computing. (JORDAN et al., 2023).
- **Jordan, M. G.**, Korol, G., Rutzig, Knorst, T., M. B., & Beck, A. C. S. (2022).

ERIN: Energy-Aware Resource Provisioning Framework for CPU-FPGA Multi-tenant Environments. IEEE Design & Test. (JORDAN et al., 2022).

- **Jordan, M. G.**, Korol, G., Rutzig, M. B., & Beck, A. C. S. (2021). Resource-Aware Collaborative Allocation for CPU-FPGA Cloud Environments. IEEE Transactions on Circuits and Systems II: Express Briefs, 68(5), 1655-1659. (JORDAN et al.,).
- **Jordan, M. G.**, Korol, G., Rutzig, M. B., Beck, A. C. S. (2021, October). FAIR: Fully-Adaptive Framework for Improving Resource Provisioning in Collaborative CPU-FPGA Cloud Environments. In 2021 IEEE 33rd International Symposium on Computer Architecture and High-Performance Computing (SBAC-PAD) (pp. 147-156). IEEE. (JORDAN et al., 2021a).
- **Jordan, M. G.**, Korol, G., Rutzig, M. B., & Beck, A. C. S. (2021, August). MUTECO: A Framework for Collaborative Allocation in CPU-FPGA Multi-tenant Environments. In 2021 34th SBC/SBMicro/IEEE/ACM Symposium on Integrated Circuits and Systems Design (SBCCI) (pp. 1-6). IEEE. (Honorable Mention Paper). (JORDAN et al., 2021b).
- Lignati, B. N., **Jordan, M. G.**, Korol, G., Rutzig, M. B., & Beck, A. C. S. (2021, January). Exploiting HLS-Generated Multi-Version Kernels to Improve CPU-FPGA Cloud Systems. In 2021 26th Asia and South Pacific Design Automation Conference (ASP-DAC) (pp. 536-541). IEEE. (LIGNATI et al., 2021b).
- Knorst, T., **Jordan, M. G.**, Lorenzen, A. F., Rutzig, M. B., & Beck, A. C. S. (2021, August). ETCG: Energy-Aware CPU Thread Throttling for CPU-GPU Collaborative Environments. In 2021 34th Symposium on Integrated Circuits and Systems Design (SBCCI) (pp. 1-6). IEEE. (KNORST et al., 2021a).
- Knorst, T., **Jordan, M. G.**, Lorenzon, A. F., Rutzig, M. B., & Beck, A. C. S. (2021, November). ETCF–Energy-Aware CPU Thread Throttling and Workload Balancing Framework for CPU-FPGA Collaborative Environments. In 2021 XI Brazilian Symposium on Computing Systems Engineering (SBESC) (pp. 1-8). IEEE. (KNORST et al., 2021b).
- Vicenzi, J. C., Knorst, T., **Jordan, M. G.**, Korol, G., Beck, A. C. S., & Rutzig, M. B. (2021, November). TRIPP: Transparent Resource Provisioning for Multi-Tenant CPU-GPU based Cloud Environments. In 2021 XI Brazilian Symposium on Computing Systems Engineering (SBESC) (pp. 1-8). IEEE. (VICENZI et al., 2021).

6.3.2 Publications as a Result from Collaborations

Besides the above-mentioned publications directly related to this thesis work, the following works were also published during the author's time as a Ph.D. student/candidate. These are the results of collaboration between other students in the group:

- **Jordan, M. G.**, Brandalero, M., Malfatti, G. M., Oliveira, G. F., Lorenzon, A. F., da Silva, B. C., ... I& Beck, A. C. S. (2020). Data clustering for efficient approximate computing. Design Automation for Embedded Systems (DAES), 24(1), 3-22. (SCHWARZROCK et al., 2021).
- Korol, G., **Jordan, M. G.**, Rutzig, M. B., & Beck, A. C. S. (2022). AdaFlow: a framework for adaptive dataflow CNN acceleration on FPGAs. In: 2022 Design, Automation Test in Europe Conference Exhibition (DATE). IEEE, 2022. p. 244-249. (KOROL et al., 2022a).
- Korol, G., **Jordan, M. G.**, Rutzig, M. B., & Beck, A. C. S. (2022). ConfAx: Exploiting Approximate Computing for Configurable FPGA CNN Acceleration at the Edge. IEEE International Symposium on Circuits and Systems (ISCAS). (KOROL et al., 2022b).
- Korol, G., **Jordan, M. G.**, Rutzig, M. B., & Beck, A. C. S. (2021). Synergistically Exploiting CNN Pruning and HLS Versioning for Adaptive Inference on Multi-FPGAs at the Edge. ACM Transactions on Embedded Computing Systems (TECS), 20(5s), 1-26. (KOROL et al., 2021)
- Korol, G., **Jordan, M. G.**, Brandalero, M., Hübner, M., Rutzig, M. B., I& Beck, A. C. S. (2020, August). MCEA: A Resource-Aware Multicore CGRA Architecture for the Edge. In 2020 30th International Conference on Field-Programmable Logic and Applications (FPL) (pp. 33-39). IEEE. (KOROL et al., 2020).
- Schwarzrock, J., **Jordan, M. G.**, Korol, G., de Oliveira, C. C., Lorenzon, A. F., Rutzig, M. B., I& Beck, A. C. S. (2021). Dynamic concurrency throttling on numa systems and data migration impacts. Design Automation for Embedded Systems (DAES), 25(2), 135-160. (SCHWARZROCK et al., 2021).
- Silva, R., Korol, G., **Jordan, M. G.**, Brandalero, M., Hübner, M., Pereira, M., ... I& Beck, A. C. S. (2020, August). A Management Technique for Concurrent Access to a Reconfigurable Accelerator. In 2020 33rd Symposium on Integrated Circuits and Systems Design (SBCCI) (pp. 1-6). IEEE. (SILVA et al., 2020).

- Knorst, T., Vicenzi, J., **Jordan, M. G.**, de Almeida, J. H., Korol, G., Beck, A. C. S., I& Rutzig, M. B. (2020, August). Unlocking the Full Potential of Heterogeneous Accelerators by Using a Hybrid Multi-Target Binary Translator. In 2020 33rd Symposium on Integrated Circuits and Systems Design (SBCCI) (pp. 1-6). IEEE. (KNORST et al., 2020).
- Korol, G., **Jordan, M.**, Brandalero, M., Rutzig, M. B., I& Beck, A. C. S. (2019, November). Power-aware phase oriented reconfigurable architecture. In 2019 26th IEEE International Conference on Electronics, Circuits and Systems (ICECS) (pp. 626-629). IEEE. (KOROL et al., 2019a).
- Korol, G., **Jordan, M.**, Silva, R. S., Pereira, M. M., Brandalero, M., Rutzig, M. B., I& Beck, A. C. S. (2019, December). A runtime power-aware phase predictor for cgras. In 2019 International Conference on ReConFigurable Computing and FPGAs (ReConFig) (pp. 1-8). IEEE. (KOROL et al., 2019b).
- Schwarzrock, J., **Jordan, M. G.**, Korol, G., de Oliveira, C. C., Lorenzon, A. F., I& Beck, A. C. S. (2019, November). On the influence of data migration in dynamic thread management of parallel applications. In 2019 IX Brazilian Symposium on Computing Systems Engineering (SBESC) (pp. 1-8). IEEE. (SCHWARZROCK et al., 2019).
- Rocha, H., Korol, G., **Jordan, M.**, Krause, A., Silveira, R., Vieira, C., ... I& Beck, A. C. S. (2020, August). Firefly: An Open-source Rocket-based Intermittent Framework. In 2020 33rd Symposium on Integrated Circuits and Systems Design (SBCCI) (pp. 1-6). IEEE. (ROCHA et al., 2020).

REFERENCES

ALTERA. Fpga architecture white paper. In: . [S.l.: s.n.], 2006.

AYDIN, H. et al. Power-aware scheduling for periodic real-time tasks. **IEEE Transactions on computers**, IEEE, v. 53, n. 5, p. 584–600, 2004.

BACIS, M.; BRONDOLIN, R.; SANTAMBROGIO, M. D. Blastfunction: an fpga-as-a-service system for accelerated serverless computing. In: IEEE. **2020 Design, Automation & Test in Europe Conference & Exhibition (DATE)**. [S.l.], 2020. p. 852–857.

BANERJEE, S.; BOZORGZADEH, E.; DUTT, N. D. Integrating physical constraints in hw-sw partitioning for architectures with partial dynamic reconfiguration. **IEEE Transactions on Very Large Scale Integration (VLSI) Systems**, IEEE, v. 14, n. 11, p. 1189–1202, 2006.

BELOGLAZOV, A.; ABAWAJY, J.; BUYYA, R. Energy-aware resource allocation heuristics for efficient management of data centers for cloud computing. **Future generation computer systems**, Elsevier, v. 28, n. 5, p. 755–768, 2012.

BENDOU, Y. **FPGA Dynamic Function eXchange**. Thesis (PhD) — Politecnico di Torino, 2020.

BERTOLINO, M. et al. Efficient scheduling of fpgas for cloud data center infrastructures. In: IEEE. **DSD**. [S.l.], 2020. p. 57–64.

BHATTI, M. K.; BELLEUDY, C.; AUGUIN, M. An inter-task real time dvfs scheme for multiprocessor embedded systems. In: IEEE. **2010 Conference on Design and Architectures for Signal and Image Processing (DASIP)**. [S.l.], 2010. p. 136–143.

BYMA, S. et al. Fpgas in the cloud: Booting virtualized hardware accelerators with openstack. In: IEEE. **FCCM**. [S.l.], 2014. p. 109–116.

CADAMBI, S. et al. Managing pipeline-reconfigurable fpgas. In: **Proceedings of the 1998 ACM/SIGDA sixth international symposium on Field programmable gate arrays**. [S.l.: s.n.], 1998. p. 55–64.

CATTANEO, R. et al. Para-sched: A reconfiguration-aware scheduler for reconfigurable architectures. In: IEEE. **2014 IEEE International Parallel & Distributed Processing Symposium Workshops**. [S.l.], 2014. p. 243–250.

CHANG, L.-W. et al. Collaborative computing for heterogeneous integrated systems. In: **Proceedings of the 8th ACM/SPEC on International Conference on Performance Engineering**. [S.l.: s.n.], 2017. p. 385–388.

CHEN, F. et al. Enabling fpgas in the cloud. In: **CF**. [S.l.: s.n.], 2014. p. 1–10.

CHEN, G.; HUANG, K.; KNOLL, A. Energy optimization for real-time multiprocessor system-on-chip with optimal dvfs and dpm combination. **ACM Transactions on Embedded Computing Systems (TECS)**, ACM New York, NY, USA, v. 13, n. 3s, p. 1–21, 2014.

CHENG, H.-Y. et al. Core vs. uncore: The heart of darkness. In: IEEE. **2015 52nd ACM/EDAC/IEEE Design Automation Conference (DAC)**. [S.l.], 2015. p. 1–6.

CHOI, Y.-k.; CONG, J. Hls-based optimization and design space exploration for applications with variable loop bounds. In: IEEE. **2018 IEEE/ACM International Conference on Computer-Aided Design (ICCAD)**. [S.l.], 2018. p. 1–8.

CONG, J. et al. Cpu-fpga coscheduling for big data applications. **IEEE Design & Test**, IEEE, v. 35, n. 1, p. 16–22, 2017.

CONG, J. et al. Understanding performance differences of fpgas and gpus. In: IEEE. **2018 IEEE 26th Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM)**. [S.l.], 2018. p. 93–96.

CONG, J. et al. Automated accelerator generation and optimization with composable, parallel and pipeline architecture. In: IEEE. **2018 55th ACM/ESDA/IEEE Design Automation Conference (DAC)**. [S.l.], 2018. p. 1–6.

CORDONE, R. et al. Partitioning and scheduling of task graphs on partially dynamically reconfigurable fpgas. **IEEE transactions on computer-aided design of integrated circuits and systems**, IEEE, v. 28, n. 5, p. 662–675, 2009.

DAI, G. et al. Online scheduling for fpga computation in the cloud. In: IEEE. **2014 International Conference on Field-Programmable Technology (FPT)**. [S.l.], 2014. p. 330–333.

DAMIANI, A. et al. Blastfunction: A full-stack framework bringing fpga hardware acceleration to cloud-native applications. **ACM Transactions on Reconfigurable Technology and Systems (TRETS)**, ACM New York, NY, v. 15, n. 2, p. 1–27, 2022.

DEIANA, E. A. et al. A multiobjective reconfiguration-aware scheduler for fpga-based heterogeneous architectures. In: IEEE. **2015 International Conference on ReConFigurable Computing and FPGAs (ReConFig)**. [S.l.], 2015. p. 1–6.

DHAR, A. et al. Dml: Dynamic partial reconfiguration with scalable task scheduling for multi-applications on fpgas. **IEEE Transactions on Computers**, IEEE, 2021.

DIESSEL, O. et al. Dynamic scheduling of tasks on partially reconfigurable fpgas. **IEE Proceedings-Computers and Digital Techniques**, IET, v. 147, n. 3, p. 181–188, 2000.

DORFLINGER, A. et al. Hardware and software task scheduling for arm-fpga platforms. In: IEEE. **2018 NASA/ESA Conference on Adaptive Hardware and Systems (AHS)**. [S.l.], 2018. p. 66–73.

FAHMY, S. A.; VIPIN, K.; SHREEJITH, S. Virtualized fpga accelerators for efficient cloud computing. In: IEEE. **CloudCom**. [S.l.], 2015. p. 430–435.

FERRANDI, F. et al. Ant colony optimization for mapping, scheduling and placing in reconfigurable systems. In: IEEE. **2013 NASA/ESA Conference on Adaptive Hardware and Systems (AHS-2013)**. [S.l.], 2013. p. 47–54.

GAO, Y. et al. An energy and deadline aware resource provisioning, scheduling and optimization framework for cloud systems. In: IEEE. **2013 International Conference on Hardware/Software Codesign and System Synthesis (CODES+ ISSS)**. [S.l.], 2013. p. 1–10.

GERARDS, M. E.; HURINK, J. L.; KUPER, J. On the interplay between global dvfs and scheduling tasks with precedence constraints. **IEEE Transactions on Computers**, IEEE, v. 64, n. 6, p. 1742–1754, 2014.

GOLDBERG, D. E. **Genetic algorithms in search, optimization & machine learning, third impression**. [S.l.]: Pearson education Inc, 2008.

GUO, Z. et al. Energy-efficient multi-core scheduling for real-time dag tasks. In: SCHLOSS DAGSTUHL-LEIBNIZ-ZENTRUM FUER INFORMATIK. **29th Euromicro conference on real-time systems (ECRTS 2017)**. [S.l.], 2017.

GUPTA, V. et al. The forgotten {'Uncore'}: On the {Energy-Efficiency} of heterogeneous cores. In: **2012 USENIX Annual Technical Conference (USENIX ATC 12)**. [S.l.: s.n.], 2012. p. 367–372.

HAN, R. et al. Edgetuner: Fast scheduling algorithm tuning for dynamic edge-cloud workloads and resources. In: IEEE. **IEEE INFOCOM 2022-IEEE Conference on Computer Communications**. [S.l.], 2022. p. 880–889.

HANDA, M.; VEMURI, R. An efficient algorithm for finding empty space for online fpga placement. In: **Proceedings of the 41st annual Design Automation Conference**. [S.l.: s.n.], 2004. p. 960–965.

HEMDAN, E. E.-D.; SHOUMAN, M. A.; KARAR, M. E. Covidx-net: A framework of deep learning classifiers to diagnose covid-19 in x-ray images. **arXiv preprint arXiv:2003.11055**, 2020.

HUANG, S. et al. Analysis and modeling of collaborative execution strategies for heterogeneous cpu-fpga architectures. In: ACM. **ICPE**. [S.l.], 2019. p. 79–90.

JING, C.; ZHU, Y.; LI, M. Energy-efficient scheduling on multi-fpga reconfigurable systems. **Microprocessors and Microsystems**, Elsevier, v. 37, n. 6-7, p. 590–600, 2013.

JORDAN, M. G. et al. Erin: Energy-aware resource-provisioning framework for cpu-fpga multitenant environment. **IEEE Design & Test**, IEEE, v. 39, n. 6, p. 138–146, 2022.

JORDAN, M. G. et al. Energy-aware fully-adaptive resource provisioning in collaborative cpu-fpga cloud environments. **Journal of Parallel and Distributed Computing**, Elsevier, 2023.

JORDAN, M. G. et al. Resource-aware collaborative allocation for CPU-FPGA cloud environments. **TCAS-II**, p. 1–1. ISSN 1549-7747, 1558-3791. Available from Internet: <https://ieeexplore.ieee.org/document/9380748/>.

JORDAN, M. G. et al. Fair: Fully-adaptive framework for improving resource provisioning in collaborative cpu-fpga cloud environments. In: IEEE. **2021 IEEE 33rd International Symposium on Computer Architecture and High Performance Computing (SBAC-PAD)**. [S.l.], 2021. p. 147–156.

JORDAN, M. G. et al. Muteco: A framework for collaborative allocation in cpu-fpga multi-tenant environments. In: IEEE. **SBCCI**. [S.l.], 2021. p. 1–6.

JORDAN, M. G. et al. Resource-aware collaborative allocation for cpu-fpga cloud environments. **TCAS-II**, IEEE, v. 68, n. 5, p. 1655–1659, 2021.

JUAREZ, F.; EJARQUE, J.; BADIA, R. M. Dynamic energy-aware scheduling for parallel task-based application in cloud computing. **Future Generation Computer Systems**, Elsevier, v. 78, p. 257–271, 2018.

JUNGBLUT, P.; KRANZLMÜLLER, D. Dynamic spatial multiplexing on fpgas with opencl. In: SPRINGER. **International Symposium on Applied Reconfigurable Computing**. [S.l.], 2021. p. 265–274.

KELLERER, H.; PFERSCHY, U.; PISINGER, D. Multidimensional knapsack problems. In: **Knapsack problems**. [S.l.]: Springer, 2004. p. 235–283.

KHAN, A. et al. An approach to realize time-sharing of flip-flops in time-multiplexed fpgas. In: IEEE. **Proceedings. 2004 IEEE International Conference on Field-Programmable Technology (IEEE Cat. No. 04EX921)**. [S.l.], 2004. p. 351–354.

KHANH, P. N. et al. Exploiting loop-array dependencies to accelerate the design space exploration with high level synthesis. In: **DATE**. [S.l.: s.n.], 2015. p. 157–162.

KHATTAR, N.; SIDHU, J.; SINGH, J. Toward energy-efficient cloud computing: a survey of dynamic power management and heuristics-based optimization techniques. **The Journal of Supercomputing**, Springer, v. 75, n. 8, p. 4750–4810, 2019.

KIM, W. et al. System level analysis of fast, per-core dvfs using on-chip switching regulators. In: IEEE. **2008 IEEE 14th International Symposium on High Performance Computer Architecture**. [S.l.], 2008. p. 123–134.

KNODEL, O.; GENSSLER, P. R.; SPALLEK, R. G. Virtualizing reconfigurable hardware to provide scalability in cloud architectures. In: **International Conference on Advances in Circuits, Electronics and Micro-electronics (CENICS)**. [S.l.: s.n.], 2017.

KNODEL, O.; LEHMANN, P.; SPALLEK, R. G. Rc3e: Reconfigurable accelerators in data centres and their provision by adapted service models. In: IEEE. **2016 IEEE 9th International Conference on Cloud Computing (CLOUD)**. [S.l.], 2016. p. 19–26.

KNORST, T. et al. Etcg: Energy-aware cpu thread throttling for cpu-gpu collaborative environments. In: IEEE. **SBCCI**. [S.l.], 2021. p. 1–6.

KNORST, T. et al. Etcf–energy-aware cpu thread throttling and workload balancing framework for cpu-fpga collaborative environments. In: IEEE. **2021 XI Brazilian Symposium on Computing Systems Engineering (SBESC)**. [S.l.], 2021. p. 1–8.

KNORST, T. et al. On the benefits of collaborative thread throttling and hls-versioning in cpu-fpga environments. In: IEEE. **2022 35th SBC/SBMicro/IEEE/ACM Symposium on Integrated Circuits and Systems Design (SBCCI)**. [S.l.], 2022. p. 1–6.

KNORST, T. et al. Unlocking the full potential of heterogeneous accelerators by using a hybrid multi-target binary translator. In: IEEE. **2020 33rd Symposium on Integrated Circuits and Systems Design (SBCCI)**. [S.l.], 2020. p. 1–6.

KONG, F. et al. Energy-efficient scheduling for parallel real-time tasks based on level-packing. In: **Proceedings of the 2011 ACM Symposium on Applied Computing**. [S.l.: s.n.], 2011. p. 635–640.

KOROL, G. et al. Power-aware phase oriented reconfigurable architecture. In: IEEE. **2019 26th IEEE International Conference on Electronics, Circuits and Systems (ICECS)**. [S.l.], 2019. p. 626–629.

KOROL, G. et al. A runtime power-aware phase predictor for cgras. In: IEEE. **2019 International Conference on ReConFigurable Computing and FPGAs (ReConFig)**. [S.l.], 2019. p. 1–8.

KOROL, G. et al. Mcea: A resource-aware multicore cgra architecture for the edge. In: IEEE. **2020 30th International conference on field-programmable logic and applications (FPL)**. [S.l.], 2020. p. 33–39.

KOROL, G. et al. Synergistically exploiting cnn pruning and hls versioning for adaptive inference on multi-fpgas at the edge. **ACM Transactions on Embedded Computing Systems (TECS)**, ACM New York, NY, v. 20, n. 5s, p. 1–26, 2021.

KOROL, G. et al. Adaflow: a framework for adaptive dataflow cnn acceleration on fpgas. In: IEEE. **2022 Design, Automation & Test in Europe Conference & Exhibition (DATE)**. [S.l.], 2022. p. 244–249.

KOROL, G. et al. Confax: Exploiting approximate computing for configurable fpga cnn acceleration at the edge. In: IEEE. **2022 IEEE International Symposium on Circuits and Systems (ISCAS)**. [S.l.], 2022. p. 1650–1654.

KUMAR, H.; CHAWLA, N.; MUKHOPADHYAY, S. Biasp: a dvfs based exploit to undermine resource allocation fairness in linux platforms. In: **ISLPED**. [S.l.: s.n.], 2020. p. 223–228.

KUMARI, A.; MEHTA, A. K. A hybrid intrusion detection system based on decision tree and support vector machine. In: IEEE. **2020 IEEE 5th International conference on computing communication and automation (ICCCA)**. [S.l.], 2020. p. 396–400.

LAU, J. et al. Heterorefactor: refactoring for heterogeneous computing with fpga. In: IEEE. **2020 IEEE/ACM 42nd International Conference on Software Engineering (ICSE)**. [S.l.], 2020. p. 493–505.

LE, D.-C. et al. Performance analysis of adaptive resource allocation scheme for opencl-based fpga virtualization system. In: IEEE. **2019 International Conference on Information and Communication Technology Convergence (ICTC)**. [S.l.], 2019. p. 392–397.

LI, J.; MARTINEZ, J. F. Dynamic power-performance adaptation of parallel computation on chip multiprocessors. In: IEEE. **The Twelfth International Symposium on High-Performance Computer Architecture, 2006.** [S.l.], 2006. p. 77–87.

LI, X.; MASKELL, D. L. Time-multiplexed fpga overlay architectures: A survey. **ACM Transactions on Design Automation of Electronic Systems (TODAES)**, ACM New York, NY, USA, v. 24, n. 5, p. 1–19, 2019.

LI, X. et al. Dhl: Enabling flexible software network functions with fpga acceleration. In: IEEE. **2018 IEEE 38th International Conference on Distributed Computing Systems (ICDCS)**. [S.l.], 2018. p. 1–11.

LIANG, H.; SINHA, S.; ZHANG, W. Parallelizing hardware tasks on multicontext fpga with efficient placement and scheduling algorithms. **IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems**, IEEE, v. 37, n. 2, p. 350–363, 2017.

LIGNATI, B. N. et al. Exploiting HLS-generated multi-version kernels to improve CPU-FPGA cloud systems. In: **ASPDAC**. ACM, 2021. p. 536–541. ISBN 978-1-4503-7999-1. Available from Internet: <https://dl.acm.org/doi/10.1145/3394885.3431557>.

LIGNATI, B. N. et al. Exploiting hls-generated multi-version kernels to improve cpu-fpga cloud systems. In: IEEE. **ASP-DAC**. [S.l.], 2021. p. 536–541.

LIN, C.-C. et al. Energy-efficient task scheduling for multi-core platforms with per-core dvfs. **Journal of Parallel and Distributed Computing**, Elsevier, v. 86, p. 71–81, 2015.

LIU, J.; BAYLISS, S.; CONSTANTINIDES, G. A. Offline synthesis of online dependence testing: Parametric loop pipelining for hls. In: IEEE. **FCCM**. [S.l.], 2015. p. 159–162.

LIU, X. et al. Energy-aware task scheduling strategies with qos constraint for green computing in cloud data centers. In: **Proceedings of the 2018 Conference on Research in Adaptive and Convergent Systems**. [S.l.: s.n.], 2018. p. 260–267.

MAJUMDER, A. et al. Energy-aware real-time tasks processing for fpga based heterogeneous cloud. **IEEE Transactions on Sustainable Computing**, IEEE, 2021.

MAK, W.-K.; YOUNG, E. F. Temporal logic replication for dynamically reconfigurable fpga partitioning. **IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems**, IEEE, v. 22, n. 7, p. 952–959, 2003.

MARCH, J. L. et al. Power-aware scheduling with effective task migration for real-time multicore embedded systems. **Concurrency and Computation: Practice and Experience**, Wiley Online Library, v. 25, n. 14, p. 1987–2001, 2013.

MARCONI, T. et al. Intelligent merging online task placement algorithm for partial reconfigurable systems. In: IEEE. **2008 Design, Automation and Test in Europe**. [S.l.], 2008. p. 1346–1351.

MAVROIDIS, I. et al. Ecoscale: Reconfigurable computing and runtime system for future exascale systems. In: IEEE. **2016 Design, Automation & Test in Europe Conference & Exhibition (DATE)**. [S.l.], 2016. p. 696–701.

MEI, B.; SCHAUMONT, P.; VERNALDE, S. A hardware-software partitioning and scheduling algorithm for dynamically reconfigurable embedded systems. In: **Proceedings of ProRISC**. [S.l.: s.n.], 2000. p. 405–411.

MELONI, P. et al. Neuraghe: exploiting cpu-fpga synergies for efficient and flexible cnn inference acceleration on zynq socs. **ACM Transactions on Reconfigurable Technology and Systems (TRETS)**, ACM New York, NY, USA, v. 11, n. 3, p. 1–24, 2018.

MENG, Y.; KUPPANNAGARI, S.; PRASANNA, V. Accelerating proximal policy optimization on cpu-fpga heterogeneous platforms. In: IEEE. **2020 IEEE 28th Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM)**. [S.l.], 2020. p. 19–27.

MINHAS, U. I. et al. Efficient, dynamic multi-task execution on fpga-based computing systems. **IEEE Transactions on Parallel and Distributed Systems**, IEEE, v. 33, n. 3, p. 710–722, 2021.

MOHANTY, A. et al. High-performance face detection with cpu-fpga acceleration. In: IEEE. **2016 IEEE International Symposium on Circuits and Systems (ISCAS)**. [S.l.], 2016. p. 117–120.

MOODY, K. P. **FPGA-Accelerated Digital Signal Processing for UAV Traffic Control Radar**. Thesis (PhD) — Brigham Young University, 2021.

MORALES-VILLANUEVA, A.; KUMAR, R.; GORDON-ROSS, A. Configuration prefetching and reuse for preemptive hardware multitasking on partially reconfigurable fpgas. In: IEEE. **2016 Design, Automation & Test in Europe Conference & Exhibition (DATE)**. [S.l.], 2016. p. 1505–1508.

NAFKHA, A.; LOUET, Y. Accurate measurement of power consumption overhead during fpga dynamic partial reconfiguration. In: IEEE. **2016 International Symposium on Wireless Communication Systems (ISWCS)**. [S.l.], 2016. p. 586–591.

NATALE, M. D.; BINI, E. Optimizing the fpga implementation of hrt systems. In: IEEE. **13th IEEE Real Time and Embedded Technology and Applications Symposium (RTAS'07)**. [S.l.], 2007. p. 22–31.

NGUYEN, T. D.; KUMAR, A. Maximizing the serviceability of partially reconfigurable fpga systems in multi-tenant environment. In: **FPGA**. [S.l.: s.n.], 2020. p. 29–39.

PAOLILLO, A. et al. Power minimization for parallel real-time systems with malleable jobs and homogeneous frequencies. In: IEEE. **2014 IEEE 20th International Conference on Embedded and Real-Time Computing Systems and Applications**. [S.l.], 2014. p. 1–10.

PARSA, S.; ENTEZARI-MALEKI, R. Rasa: a new grid task scheduling algorithm. **JDCTA**, v. 3, n. 4, p. 91–99, 2009.

PELLIZZONI, R.; CACCAMO, M. Adaptive allocation of software and hardware real-time tasks for fpga-based embedded systems. In: IEEE. **12th IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS'06)**. [S.l.], 2006. p. 208–220.

PERINA, A. B.; BECKER, J.; BONATO, V. Lina: Timing-constrained high-level synthesis performance estimator for fast dse. In: IEEE. **2019 International Conference on Field-Programmable Technology (ICFPT)**. [S.l.], 2019. p. 343–346.

PILLAI, P.; SHIN, K. G. Real-time dynamic voltage scaling for low-power embedded operating systems. In: **Proceedings of the eighteenth ACM symposium on Operating systems principles**. [S.l.: s.n.], 2001. p. 89–102.

PURGATO, A. et al. Resource-efficient scheduling for partially-reconfigurable fpga-based systems. In: IEEE. **2016 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW)**. [S.l.], 2016. p. 189–197.

PUTNAM, A. et al. A reconfigurable fabric for accelerating large-scale datacenter services. In: IEEE. **2014 ACM/IEEE 41st International Symposium on Computer Architecture (ISCA)**. [S.l.], 2014. p. 13–24.

QIN, Y. et al. Energy-efficient intra-task dvfs scheduling using linear programming formulation. **IEEE Access**, IEEE, v. 7, p. 30536–30547, 2019.

RABOZZI, M. et al. Floorplanning automation for partial-reconfigurable fpgas via feasible placements generation. **IEEE Transactions on Very Large Scale Integration (VLSI) Systems**, IEEE, v. 25, n. 1, p. 151–164, 2016.

RIHANI, M. A. et al. Dynamic and partial reconfiguration power consumption runtime measurements analysis for zynq soc devices. In: IEEE. **2016 International Symposium on Wireless Communication Systems (ISWCS)**. [S.l.], 2016. p. 592–596.

ROCHA, H. et al. Firefly: An open-source rocket-based intermittent framework. In: IEEE. **2020 33rd Symposium on Integrated Circuits and Systems Design (SBCCI)**. [S.l.], 2020. p. 1–6.

RODRÍGUEZ, A. et al. Parallel multiprocessing and scheduling on the heterogeneous xeon+ fpga platform. **The Journal of Supercomputing**, Springer, v. 76, n. 6, p. 4645–4665, 2020.

RODRÍGUEZ, A. et al. Lightweight asynchronous scheduling in heterogeneous reconfigurable systems. **Journal of Systems Architecture**, Elsevier, v. 124, p. 102398, 2022.

SADEGHI, M.; RAZAVI, S. A.; ZAMANI, M. S. Reducing reconfiguration time in fpgas. In: IEEE. **2019 27th Iranian Conference on Electrical Engineering (ICEE)**. [S.l.], 2019. p. 1844–1848.

SAFE, M. et al. On stopping criteria for genetic algorithms. In: SPRINGER. **SBIA**. [S.l.], 2004. p. 405–413.

SALAMI, B. et al. An experimental study of reduced-voltage operation in modern fpgas for neural network acceleration. In: IEEE. **2020 50th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)**. [S.l.], 2020. p. 138–149.

SCHWARZROCK, J. et al. On the influence of data migration in dynamic thread management of parallel applications. In: IEEE. **2019 IX Brazilian Symposium on Computing Systems Engineering (SBESC)**. [S.l.], 2019. p. 1–8.

SCHWARZROCK, J. et al. Dynamic concurrency throttling on numa systems and data migration impacts. **Design Automation for Embedded Systems**, Springer, v. 25, p. 135–160, 2021.

SEO, E. et al. Energy efficient scheduling of real-time tasks on multicore processors. **IEEE transactions on parallel and distributed systems**, IEEE, v. 19, n. 11, p. 1540–1552, 2008.

SEO, J.; KIM, T.; LEE, J. Optimal intratask dynamic voltage-scaling technique and its practical extensions. **IEEE transactions on computer-aided design of integrated circuits and systems**, IEEE, v. 25, n. 1, p. 47–57, 2005.

SEYOUM, B. et al. Automating the design flow under dynamic partial reconfiguration for hardware-software co-design in fpga soc. In: **Proceedings of the 36th Annual ACM Symposium on Applied Computing**. [S.l.: s.n.], 2021. p. 481–490.

SEYOUM, B. B.; BIONDI, A.; BUTTAZZO, G. C. Flora: Floorplan optimizer for reconfigurable areas in fpgas. **ACM Transactions on Embedded Computing Systems (TECS)**, ACM New York, NY, USA, v. 18, n. 5s, p. 1–20, 2019.

SHA, S. et al. On fundamental principles for thermal-aware design on periodic real-time multi-core systems. **ACM Transactions on Design Automation of Electronic Systems (TODAES)**, ACM New York, NY, USA, v. 25, n. 2, p. 1–23, 2020.

SHEIKH, S. Z.; PASHA, M. A. Energy-efficient real-time scheduling on multicores: A novel approach to model cache contention. **ACM Transactions on Embedded Computing Systems (TECS)**, ACM New York, NY, USA, v. 19, n. 4, p. 1–25, 2020.

SHIN, D.; KIM, J. A profile-based energy-efficient intra-task voltage scheduling algorithm for real-time applications. In: **Proceedings of the 2001 international symposium on Low power electronics and design**. [S.l.: s.n.], 2001. p. 271–274.

SHIN, D.; KIM, J.; LEE, S. Intra-task voltage scheduling for low-energy hard real-time applications. **IEEE Design & Test of Computers**, IEEE, v. 18, n. 2, p. 20–30, 2001.

SHIN, Y.; CHOI, K. Power conscious fixed priority scheduling for hard real-time systems. In: IEEE. **Proceedings 1999 Design Automation Conference (Cat. No. 99CH36361)**. [S.l.], 1999. p. 134–139.

SHIN, Y.; CHOI, K. System-level power optimization of embedded systems. **Rapport nSNU-EE-TR-2000-3, School of Electrical Engineering, Seoul National University**, 2000.

SHIN, Y.; CHOI, K.; SAKURAI, T. Power optimization of real-time embedded systems on variable speed processors. In: IEEE. **IEEE/ACM International Conference on Computer Aided Design. ICCAD-2000. IEEE/ACM Digest of Technical Papers (Cat. No. 00CH37140)**. [S.l.], 2000. p. 365–368.

SIDIROPOULOS, H. et al. The vineyard framework for heterogeneous cloud applications: The brainframe case. In: IEEE. **2018 Conference on Design and Architectures for Signal and Image Processing (DASIP)**. [S.l.], 2018. p. 70–75.

SILVA, L. B. D. et al. Ready: A fine-grained multithreading overlay framework for modern cpu-fpga dataflow applications. **ACM Transactions on Embedded Computing Systems (TECS)**, ACM New York, NY, USA, v. 18, n. 5s, p. 1–20, 2019.

SILVA, R. et al. A management technique for concurrent access to a reconfigurable accelerator. In: IEEE. **2020 33rd Symposium on Integrated Circuits and Systems Design (SBCCI)**. [S.l.], 2020. p. 1–6.

SKHIRI, R. et al. From fpga to support cloud to cloud of fpga: State of the art. **International Journal of Reconfigurable Computing**, Hindawi, v. 2019, 2019.

STAVRINIDES, G. L.; KARATZA, H. D. Energy-aware scheduling of real-time workflow applications in clouds utilizing dvfs and approximate computations. In: IEEE. **2018 IEEE 6th international conference on future internet of things and cloud (FiCloud)**. [S.l.], 2018. p. 33–40.

STEIGER, C. et al. Online scheduling and placement of real-time tasks to partially reconfigurable devices. In: IEEE. **RTSS 2003. 24th IEEE Real-Time Systems Symposium, 2003**. [S.l.], 2003. p. 224–225.

STONE, J. E.; GOHARA, D.; SHI, G. Opencl: A parallel programming standard for heterogeneous computing systems. **Computing in science & engineering**, IEEE Computer Society, v. 12, n. 3, p. 66, 2010.

TANG, Q.; GUO, B.; WANG, Z. Sw/hw partitioning and scheduling on region-based dynamic partial reconfigurable system-on-chip. **Electronics**, Multidisciplinary Digital Publishing Institute, v. 9, n. 9, p. 1362, 2020.

TATEMATSU, T. et al. Checkpoint extraction using execution traces for intra-task dvfs in embedded systems. In: IEEE. **2011 Sixth IEEE International Symposium on Electronic Design, Test and Application**. [S.l.], 2011. p. 19–24.

TRIMBERGER, S. Scheduling designs into a time-multiplexed fpga. In: **Proceedings of the 1998 ACM/SIGDA sixth international symposium on Field programmable gate arrays**. [S.l.: s.n.], 1998. p. 153–160.

TRIMBERGER, S. et al. A time-multiplexed fpga. In: IEEE. **Proceedings. The 5th Annual IEEE Symposium on Field-Programmable Custom Computing Machines Cat. No. 97TB100186)**. [S.l.], 1997. p. 22–28.

ULLMANN, M. et al. On-demand fpga run-time system for dynamical reconfiguration with adaptive priorities. In: SPRINGER. **International Conference on Field Programmable Logic and Applications**. [S.l.], 2004. p. 454–463.

VAISHNAV, A.; PHAM, K. D.; KOCH, D. A survey on fpga virtualization. In: IEEE. **2018 28th International Conference on Field Programmable Logic and Applications (FPL)**. [S.l.], 2018. p. 131–1317.

VICENZI, J. C. et al. Tripp: Transparent resource provisioning for multi-tenant cpu-gpu based cloud environments. In: IEEE. **2021 XI Brazilian Symposium on Computing Systems Engineering (SBESC)**. [S.l.], 2021. p. 1–8.

VIPIN, K.; FAHMY, S. A. Architecture-aware reconfiguration-centric floorplanning for partial reconfiguration. In: SPRINGER. **International symposium on applied reconfigurable computing**. [S.l.], 2012. p. 13–25.

VISALAKSHI, S.; RADHA, V. A literature review of feature selection techniques and applications: Review of feature selection in data mining. In: IEEE. **2014 IEEE International Conference on Computational Intelligence and Computing Research**. [S.l.], 2014. p. 1–6.

WALDER, H.; PLATZNER, M. Non-preemptive multitasking on fpgas: Task placement and footprint transform. In: CSREA PRESS. **Proceedings of the 2nd International Conference on Engineering of Reconfigurable Systems and Architectures (ERSA)**. [S.l.], 2002. p. 24–30.

WANG, S.; ANANTHANARAYANAN, G.; MITRA, T. Optic: Optimizing collaborative cpu–gpu computing on mobile devices with thermal constraints. **IEEE TCAD**, IEEE, v. 38, n. 3, p. 393–406, 2018.

WANG, X. et al. When fpga meets cloud: A first look at performance. **IEEE Transactions on Cloud Computing**, IEEE, 2020.

WANG, Z. et al. Pipefl: Hardware/software co-design of an fpga accelerator for federated learning. **IEEE Access**, IEEE, v. 10, p. 98649–98661, 2022.

WANG, Z. et al. Melia: A mapreduce framework on opencl-based fpgas. **IEEE Transactions on Parallel and Distributed Systems**, IEEE, v. 27, n. 12, p. 3547–3560, 2016.

WASSI, G. et al. Multi-shape tasks scheduling for online multitasking on fpgas. In: IEEE. **2014 9th International Symposium on Reconfigurable and Communication-Centric Systems-on-Chip (ReCoSoC)**. [S.l.], 2014. p. 1–7.

WEI, X. et al. Throughput optimization for streaming applications on cpu-fpga heterogeneous systems. In: IEEE. **ASP-DAC**. [S.l.], 2017. p. 488–493.

WEISER, M. et al. Scheduling for reduced cpu energy. In: **Mobile Computing**. [S.l.]: Springer, 1994. p. 449–471.

WIRBEL, L. Xilinx sdaccel. 2014.

XIAN, C.; LU, Y.-H.; LI, Z. Energy-aware scheduling for real-time multiprocessor systems with uncertain task execution time. In: IEEE. **2007 44th ACM/IEEE Design Automation Conference**. [S.l.], 2007. p. 664–669.

XIAO, Z.; SONG, W.; CHEN, Q. Dynamic resource allocation using virtual machines for cloud computing environment. **IEEE transactions on parallel and distributed systems**, IEEE, v. 24, n. 6, p. 1107–1117, 2012.

XILINX. **Get Moving with Alveo**. 2019. <https://www.xilinx.com/developer/articles/example-0-loading-an-alveo-image.html>. Accessed: 2022-01-08.

XILINX. **PetaLinux Tools Documentation**. 2020. <http://xilinx.eetrend.com/files/2020-06/wen_zhang_/100049850-99702-petalinuxgongjuwendangcankaozhinan.pdf>. Accessed: 2022-01-08.

XILINX, A. **Managing Clock Frequencies - 2022.2 English - Xilinx**. 2023. (accessed: 10-Jan-2023). Available from Internet: <https://docs.xilinx.com/r/en-US/ug1393-vitis-application-acceleration/Managing-Clock-Frequencies>.

XILINX Vision Benchmarks. 2019. <https://github.com/Xilinx/SDAccel_Examples/tree/master/vision>. Accessed: 2022-01-08.

YANG, J. et al. A task scheduling method for energy-performance trade-off in clouds. In: IEEE. **2016 IEEE 18th International Conference on High Performance Computing and Communications; IEEE 14th International Conference on Smart City; IEEE 2nd International Conference on Data Science and Systems (HPCC/SmartCity/DSS)**. [S.l.], 2016. p. 1029–1036.

ZENG, G. et al. Practical energy-aware scheduling for real-time multiprocessor systems. In: IEEE. **2009 15th IEEE International Conference on Embedded and Real-Time Computing Systems and Applications**. [S.l.], 2009. p. 383–392.

ZENG, H.; PRASANNA, V. Graphact: Accelerating gcn training on cpu-fpga heterogeneous platforms. In: **Proceedings of the 2020 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays**. [S.l.: s.n.], 2020. p. 255–265.

ZHA, Y.; LI, J. Virtualizing fpgas in the cloud. In: **Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems**. [S.l.: s.n.], 2020. p. 845–858.

ZHANG, X. et al. Exploring hw/sw co-design for video analysis on cpu-fpga heterogeneous systems. **IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems**, IEEE, 2021.

ZHANG, Y.; HU, X.; CHEN, D. Z. Task scheduling and voltage selection for energy minimization. In: IEEE. **Proceedings 2002 Design Automation Conference (IEEE Cat. No. 02CH37324)**. [S.l.], 2002. p. 183–188.

ZHAO, J. et al. Performance modeling and directives optimization for high level synthesis on fpga. **IEEE TCAD**, IEEE, 2019.

ZHENG, L. A task migration constrained energy-efficient scheduling algorithm for multiprocessor real-time systems. In: IEEE. **2007 International Conference on Wireless Communications, Networking and Mobile Computing**. [S.l.], 2007. p. 3055–3058.

ZHONG, G. et al. Lin-analyzer: a high-level performance analysis tool for fpga-based accelerators. In: IEEE. **(DAC)**. [S.l.], 2016. p. 1–6.

ZHONG, G. et al. Design space exploration of fpga-based accelerators with multi-level parallelism. In: IEEE. **(DATE)**. [S.l.], 2017. p. 1141–1146.

ZHOU, P. et al. Mocha: Multinode cost optimization in heterogeneous clouds with accelerators. In: **The 2021 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays**. [S.l.: s.n.], 2021. p. 273–279.

ZHOU, S.; PRASANNA, V. K. Accelerating graph analytics on cpu-fpga heterogeneous platform. In: IEEE. **(SBAC-PAD)**. [S.l.], 2017. p. 137–144.

ZHOU, Y. et al. Rosetta: A realistic high-level synthesis benchmark suite for software programmable fpgas. In: ACM. **FPGA**. [S.l.], 2018. p. 269–278.

ZHU, J.; SUTTON, P. Fpga implementations of neural networks–a survey of a decade of progress. In: SPRINGER. **International Conference on Field Programmable Logic and Applications**. [S.l.], 2003. p. 1062–1066.

APPENDIX A — PROVISIONING STRATEGIES DESCRIPTION

This appendix Section outlines all provisioning strategies, including a discussion of their advantages and disadvantages that influenced their selection.

Genetic Multidimensional Knapsack: We have adopted the GMK to solve the Collaborative CPU-FPGA resource provisioning for the following reasons: a) GMK is capable of maximizing an objective given multiple FPGA resource constraints (BRAMs, LUTs, DSPs, FFs, and I/O) (KELLERER; PFERSCHY; PISINGER, 2004); b) the genetic properties of the algorithm allow tuning the quality of solution or algorithm execution time depending on the warehouse requirements; and c) the GMK relies on a genetic mutation property that increases the delivered Quality of Solution compared to the greedy approaches. In our case, we adapted the GMK to minimize the number of *FPGA reconfigurations* (knapsacks). *For that, it will maximize the FPGA resources provisioning for tasks that at the same time: I) are the most significant in terms of makespan time or energy; II) are benefited the most from FPGA acceleration; and III) require less resource provisioning, which enables more tasks to be executed in parallel.* Tasks that do not match these constraints, and can be executed collaboratively, will be dispatched to the CPU.

From that, we designed two versions of the GMK: one to reduce makespan (GMK-P) and the other to optimize energy consumption (GMK-E). Equation A.1 models the profit for makespan acceleration considering the evaluated task execution time relevance (k_{TR}), the acceleration of the task when executed in the FPGA (k_A), and its resource consumption (r_{ck}). The task execution time relevance (k_{TR}) is given by comparing the task CPU time (k_{TC}) with the task with the longest CPU time in the batch (k_{LTC}). The task acceleration (k_A) is given by equation A.3, which is the division of the task execution time in the CPU (k_{TC}) by the task execution time in the FPGA (k_{TF}). Equation A.4 gives the worst case in resource utilization (among BRAM/LUT/DSP/FF/IO). In other words, we take the maximum ratio between available and used resources of each type. For instance, if a task consumes 50% of FPGA BRAM resources and 20% of LUT/DSP/FF/IO, the chosen value is 50%. The main goal of such modeling is to maximize the number of tasks allocated in the FPGA, prioritizing the execution of tasks considering the aforementioned items (I, II, and III).

$$\textit{GMK-P profit} \rightarrow 1/(k_{TR} + 1/k_A + r_{ck}) \tag{A.1}$$

$$k_{TR} \rightarrow (k_{LTC} - k_{TC})/k_{LTC} \tag{A.2}$$

$$k_A \rightarrow k_{TC}/k_{TF} \tag{A.3}$$

$$r_{ck} \rightarrow max(k_{(RES)}/F_{(RES)}) \tag{A.4}$$

The profit modeling for energy optimization follows the same idea as the former model, focusing on energy consumption instead of makespan acceleration. Where (k_{ER}) represents the energy consumption relevance and (k_{ED}) represents the energy decrease when executing the task in the FPGA, respectively. The energy profit modeling is shown in Equation A.5.

$$\textit{GMK-E profit} \rightarrow 1/(k_{ER} + 1/k_{ED} + r_{ck}) \tag{A.5}$$

With the proper formulation at hand, we detail the two main phases of the GMK strategy: **FPGA Configuration Generation** (Phase 1) and **Collaborative Optimization** (Phase 2). Phase 1 uses the genetic algorithm to generate a near-optimal *FPGA Configuration*, which is represented by a set of tasks that, based on constraints mentioned above (LUTs, DSPs, BRAMs, FFs, and IO), are allocated to the FPGA. Precisely, Phase 1 follows the steps below:

1. generates a random initial population represented by a *FPGA configuration* set;
2. recombines the configurations from the current set to produce new configurations;
3. applies the fitness function to evaluate the fitness of the new configurations. The fitness function selects configurations that maximize the profit given by equations A.1 or A.5 while respecting FPGA resource constraints;
4. applies mutation in the selected configurations and replaces all configurations from the initial set with the new ones;
5. checks if a near-optimal *FPGA configuration* was achieved and evokes Phase 2. Else, it returns to step 2.

Phase 1 stops when both number of generations and convergence are reached (SAFE et al., 2004). The GMK executes Phase 1 for a given number of generations. Then it converges to a satisfactory FPGA configuration (near-optimal profit) after no profit improvements for the last n iterations.

Phase 2 evaluates tasks that were not selected to be executed over the FPGA. These tasks are, then, allocated to the CPU such that the makespan found in Phase 1's *FPGA Configurations* is not increased. For that purpose, the algorithm follows the steps below:

1. evaluates whether the FPGA makespan is higher than the one presented by each CPU core. If that is the case, and the strategy is targeting makespan (GMK-P), it goes to step 2. Else, the strategy targets energy (GMK-E), it goes to step 3;
2. allocates the remaining tasks to the CPU cores such that makespan is not increased. It is accomplished by allocating each task to the CPU with lowest makespan. It goes to step 4;
3. assigns tasks to CPU if they consume less energy than when allocated in the next *FPGA configuration*;
4. builds a *Task Arrangement* and removes the already assigned tasks from the initial batch. If there are still tasks to be allocated, the algorithm evokes Phase 1 to produce new *FPGA configurations*. Else, the *Collaborative Solution* is generated.

We have configured the GMK genetic parameters with values that have already shown to be efficient in similar problems from the literature (GOLDBERG, 2008). We have set the GMK crossover probability to 0.8, mutation to 0.08, population to 200, and the max number of generations to 200. The number of iterations to evaluate the profits improvement is given by 30% of the actual generation after a minimum number of 10 generations.

Advantageous Corner Scenario - (a) Workload Property: Suitable for heterogeneous scenarios, as it considers task acceleration, execution time, and resource consumption when distributing tasks over the FPGA. Presents good solutions (i.e., near-optimal, close to an exhaustive search [7]) when the batch is mostly composed of long-execution time tasks. (b) Target Architecture: All.

Disadvantageous Corner Scenario - (a) Workload Property: The batch is composed of short-execution time tasks. Its convergence time can represent a huge portion of the total execution time (strategy convergence time + batch execution time). (b) Target Architecture: All.

Algorithm 1 First-Come First-Served

Input: batch, $fpga_{resources}$
Output: $collaborative_{solution}$

1: **procedure** $BuildFPGAConfiguration$(batch)
2: **for** $task \in batch$ **do**
3: **if** $fpga_{resources}$ **then** ▷ available resources
4: $fpga_{config} \leftarrow task$
5: **end if**
6: **end for**
7: `call` $BuildCPUConfiguration$
8: **end procedure**
9: **procedure** $BuildCPUConfiguration$(batch)
10: **for** $task \in batch$ **do**
11: **if** $fpga_{time} < cpu_{time} + task_{time}$ **then**
12: $cpu_{config} \leftarrow task$ ▷ balance workload over available cores
13: **else**
14: $break$
15: **end if**
16: **end for**
17: `call` $BuildTaskArrangement$
18: **end procedure**
19: **procedure** $BuildTaskArrangement$(batch)
20: $task_{arrangement} \leftarrow fpga_{config}, cpu_{config}$
21: $collaborative_{solution} \leftarrow task_{arrangement}$
22: **if** $batch$ **then**
23: `call` $BuildFPGAConfiguration$
24: **else**
25: `return` $collaborative_{solution}$
26: **end if**
27: **end procedure**

Advantageous Corner Scenario - (a) Workload Property: The batch is composed of a few tasks. These tasks present similar performance when executed over FPGA or CPU and are not resource-hungry. (b) Target Architecture: Robust FPGA and robust CPU so tasks can be executed in a single Task Arrangement.

Disadvantageous Corner Scenario - (a) Workload Property: The batch comprises many tasks. Most of them are heterogeneous, which may benefit more from a specific architecture. As the strategy does not consider task characteristics, it may distribute resource-hungry tasks to the FPGA architecture, generating many Task Arrangements. (b) Target Architecture: With less resourceful FPGA and less resourceful CPU, many Task Arrangements are generated with low task parallelism.

Algorithm 2 First-Fit

Input: batch, n, $fpga_{resources}$
Output: $collaborative_{solution}$

procedure $BuildFPGAConfiguration$(batch)
  for $task \in batch$ **do**
    if $fpga_{resources}$ & $task_{Accel} \geq n$ **then** ▷ available resources and accel. $\geq$ n
      $fpga_{config} \leftarrow task$
    end if
  end for
  `call` $BuildCPUConfiguration$
end procedure
procedure $BuildCPUConfiguration$(batch)
  for $task \in batch$ **do**
    if $fpga_{config}$ **then**
      if $fpga_{time} < cpu_{time} + task_{time}$ & $task_{Accel} < n$ **then**
        $cpu_{config} \leftarrow task$ ▷ balance workload over available cores
      else
        $break$
      end if
    else
      $cpu_{config} \leftarrow task$ ▷ balance workload over available cores
    end if
  end for
  `call` $BuildTaskArrangement$
end procedure
procedure $BuildTaskArrangement$(batch)
  $task_{arrangement} \leftarrow fpga_{config}, cpu_{config}$
  $collaborative_{solution} \leftarrow task_{arrangement}$
  if $batch$ **then**
    `call` $BuildFPGAConfiguration$
  else
    `return` $collaborative_{solution}$
  end if
end procedure

Advantageous Corner Scenario - (a) Workload Property: tasks are not resource-hungry; there are sufficient FPGA-oriented (i.e., high acceleration when executed over the FPGA) and CPU-oriented tasks so that both architectures' utilization is maximized. (b) Target Architecture: Balanced architecture (similar number of CPU-oriented and FPGA-oriented tasks). Unbalanced architecture (i.e., robust CPU if there are more CPU-oriented tasks or robust FPGA if there are more FPGA-oriented tasks).

Disadvantageous Corner Scenario - (a) Workload Property: Resource-hungry tasks, generating many Task Arrangements. Most tasks are either CPU-oriented or FPGA-oriented, making one of the architectures idle. (b) Target Architecture: Balanced architecture (different number of CPU-oriented and FPGA-oriented tasks). Unbalanced architecture (e.g., less resourceful FPGA may lead to many tasks being dispatched to FPGA if most of them are FPGA-oriented, overloading the device).

Algorithm 3 MaxMin

Input: batch, $fpga_{resources}$, weight
Output: $collaborative_{solution}$

```
1: Sort batch by FPGA Time in descending order.
2: procedure BuildFPGAConfiguration(batch)
3:     for task ∈ batch do
4:         if fpga_resources then                    ▷ available resources
5:             fpga_config ← task
6:         end if
7:     end for
8:     call BuildCPUConfiguration
9: end procedure
10: procedure BuildCPUConfiguration(batch)
11:     for task ∈ batch do
12:         if fpga_time < cpu_time + task_time then
13:             cpu_config ← task                 ▷ balance workload over available cores
14:         else
15:             break
16:         end if
17:     end for
18:     call BuildTaskArrangement
19: end procedure
20: procedure BuildTaskArrangement(batch)
21:     task_arrangement ← fpga_config, cpu_config
22:     collaborative_solution ← task_arrangement
23:     if batch then
24:         call BuildFPGAConfiguration
25:     else
26:         return collaborative_solution
27:     end if
28: end procedure
```

Advantageous Corner Scenario - (a) Workload Property: The long execution time tasks are FPGA-oriented and require few FPGA resources (producing few Task Arrangements). (b) Target Architecture: FPGA is robust.

Disadvantageous Corner Scenario - (a) Workload Property: The long execution time tasks are CPU-oriented and require many FPGA resources (producing many Task Arrangements). (b) Target Architecture: FPGA with few resources.

Algorithm 4 MinMin

Input: batch, $fpga_{resources}$, weight
Output: $collaborative_{solution}$

```
1: Sort batch by FPGA Time in ascending order.
2: procedure BuildFPGAConfiguration(batch)
3:     for task ∈ batch do
4:         if fpga_resources then                    ▷ available resources
5:             fpga_config ← task
6:         end if
7:     end for
8:     call BuildCPUConfiguration
9: end procedure
10: procedure BuildCPUConfiguration(batch)
11:     for task ∈ batch do
12:         if fpga_time < cpu_time + task_time then
13:             cpu_config ← task                 ▷ balance workload over available cores
14:         else
15:             break
16:         end if
17:     end for
18:     call BuildTaskArrangement
19: end procedure
20: procedure BuildTaskArrangement(batch)
21:     task_arrangement ← fpga_config, cpu_config
22:     collaborative_solution ← task_arrangement
23:     if batch then
24:         call BuildFPGAConfiguration
25:     else
26:         return collaborative_solution
27:     end if
28: end procedure
```

Advantageous Corner Scenario - (a) Workload Property: The short execution time tasks are FPGA-oriented and require few FPGA resources, while the long ones benefit more from CPU execution. (b) Target Architecture: CPU is robust.

Disadvantageous Corner Scenario - (a) Workload Property: The short execution time tasks are resource-hungry and are CPU-oriented; the long execution time ones are FPGA-oriented. (b) Target Architecture: CPU with few resources.

Algorithm 5 RASA

Input: batch, $fpga_{resources}$, weight
Output: $collaborative_{solution}$

```
1: Sort batch by FPGA Time in descending order.
2: procedure BuildFPGAConfiguration(batch)
3:     counter ← 0
4:     while batch do
5:         if fpga_resources then                          ▷ available resources
6:             if counter is even then
7:                 fpga_config ← task_1                    ▷ gets first task from list
8:             else
9:                 fpga_config ← task_n                    ▷ gets last task from list
10:            end if
11:        else
12:            break
13:        end if
14:        counter ← counter + 1
15:    end while
16:    call BuildCPUConfiguration
17: end procedure
18: procedure BuildCPUConfiguration(batch)
19:    while task ∈ batch do
20:        counter ← 0
21:        if fpga_time < cpu_time + task_time then
22:            if counter is even then
23:                cpu_config ← task_1                     ▷ gets first task from list
24:            else
25:                cpu_config ← task_n                     ▷ gets last task from list
26:            end if
27:        else
28:            break
29:        end if
30:        counter ← counter + 1
31:    end while
32:    call BuildTaskArrangement
33: end procedure
34: procedure BuildTaskArrangement(batch)
35:    task_arrangement ← fpga_config, cpu_config
36:    collaborative_solution ← task_arrangement
37:    if batch then
38:        call BuildFPGAConfiguration
39:    else
40:        return collaborative_solution
41:    end if
42: end procedure
```

Advantageous Corner Scenario - Workload Property: Suitable for both long and short-execution time tasks that are FPGA-oriented. It benefits from the MaxMin intrinsic characteristic by executing many short execution time tasks in parallel with the longest ones. At the same time, similarly to MinMin, it gives the opportunity for short execution time tasks to execute over the FPGA. Target Architecture: FPGA is robust.

Disadvantageous Corner Scenario - (a) Workload Property: Resource-hungry tasks with low FPGA acceleration. (b) Target Architecture: CPU and FPGA with few resources.

Algorithm 6 Round-Robin

Input: batch, $fpga_{resources}$
Output: $collaborative_{solution}$

1: **procedure** $BuildFPGAConfiguration$(batch)
2: **for** $task \in batch$ **do**
3: **if** $fpga_{resources}$ **then** ▷ available resources
4: $fpga_{config} \leftarrow task$
5: $break$
6: **else**
7: `call` $BuildTaskArrangement$
8: **end if**
9: **end for**
10: `call` $BuildCPUConfiguration$
11: **end procedure**
12: **procedure** $BuildCPUConfiguration$(batch)
13: **if** $batch$ **then**
14: $cpu_{config} \leftarrow task$ ▷ balance workload over available cores
15: `call` $BuildFPGAConfiguration$
16: **else**
17: `call` $BuildTaskArrangement$
18: **end if**
19: **end procedure**
20: **procedure** $BuildTaskArrangement$(batch)
21: $task_{arrangement} \leftarrow fpga_{config}, cpu_{config}$
22: $collaborative_{solution} \leftarrow task_{arrangement}$
23: **if** $batch$ **then**
24: `call` $BuildFPGAConfiguration$
25: **else**
26: `return` $collaborative_{solution}$
27: **end if**
28: **end procedure**

Advantageous Corner Scenario - (a) Workload Property: The batch is composed of tasks that present similar performance when executed over FPGA or CPU. (b) Target Architecture: Balanced architecture (e.g., robust FPGA, robust CPU).

Disadvantageous Corner Scenario - (a) Workload Property: The batch is composed of heterogeneous tasks, which may benefit more from a specific architecture. (b) Target Architecture: For an unbalanced architecture (e.g., robust FPGA, less resourceful CPU), one of the devices may get overloaded.

Algorithm 7 Weighted Round-Robin

Input: batch, $fpga_{resources}$, $weight_{fpga}$, $weight_{cpu}$
Output: $collaborative_{solution}$

procedure $BuildFPGAConfiguration$(batch)
  $counter \leftarrow 0$
  while $batch \ \& \ counter < weight_{fpga}$ **do**
    if $fpga_{resources}$ **then** ▷ available resources
      $fpga_{config} \leftarrow task$
    else
      `call` $BuildTaskArrangement$
    end if
    $counter \leftarrow counter + 1$
  end while
  if $batch$ **then**
    `call` $BuildCPUConfiguration$
  else
    `call` $BuildTaskArrangement$
  end if
end procedure
procedure $BuildCPUConfiguration$(batch)
  $counter \leftarrow 0$
  while $batch \ \& \ counter < weight_{cpu}$ **do**
    $cpu_{config} \leftarrow task$ ▷ balance workload over available cores
    $counter \leftarrow counter + 1$
  end while
  `call` $BuildFPGAConfiguration$
end procedure
procedure $BuildTaskArrangement$(batch)
  $task_{arrangement} \leftarrow fpga_{config}, cpu_{config}$
  $collaborative_{solution} \leftarrow task_{arrangement}$
  if $batch$ **then**
    `call` $BuildFPGAConfiguration$
  else
    `return` $collaborative_{solution}$
  end if
end procedure

Advantageous Corner Scenario - (a) Workload Property: If n $>$ m, most tasks in the batch must benefit more from the FPGA. If m $<$ n, most tasks must benefit more from the CPU architecture. (b) Target Architecture: Balanced architecture. Unbalanced architecture - if n $>$ m (e.g., robust FPGA, less resourceful CPU), if n $<$ m (e.g., robust CPU, less resourceful FPGA).

Disadvantageous Corner Scenario - (a) Workload Property: The opposite of the advantageous scenario (e.g., n $>$ m and most tasks benefit from the CPU). (b) Target Architecture: Balanced or unbalanced. One of the architectures may get overloaded, while the other architecture is inactive, or its resources are not fully used.

APPENDIX B — BENCHMARKS' DESCRIPTION

Application	Categorization	Description	References
3D Rendering	Video Processing	Renders 2D images from 3D models	Zhou et al. (2018)
ADI	Mathematical	Alternating Direction Implicit Solver (Stencil)	Liu, Bayliss and Constantinides (2015)
Atax	Mathematical	Matrix Transpose and Vector Multiplication	Perina, Becker and Bonato (2019)
BackPropagation	Machine Learning	A machine-learning algorithm that trains the weights of connecting nodes on a layered neural network	Cong et al. (2018a)
Bicg	Mathematical	Biconjugate gradient method Algorithm to solve systems of linear equations	Perina, Becker and Bonato (2019)
CFD	Mathematical	Grid finite volume solver for the three-dimensional Euler equations for compressible flow	Cong et al. (2018a)
Convolve	Machine Learning	Convolutional image filtering	Xilinx... (2019)
DCT	Image Processing	Discrete Cosine Transform	Perina, Becker and Bonato (2019)
Digit Recognition	Machine Learning	Hand-Written digit classification using the KNN algorithm	Zhou et al. (2018)
Face Detection	Video Processing	Detects human faces from images using the Viola Jones algorithm	Zhou et al. (2018)
Floyd-Warshal	Graph Processing	Floyd-Warshal Shortest Path	Zhong et al. (2016)
Gesummv	Mathematical	Scalar, Vector and Matrix Multiplication	Perina, Becker and Bonato (2019)
Heat 3D	Mathematical	Heat equation over 3D data domain	Perina, Becker and Bonato (2019)
Histogram	Image Processing	Histogram Equalization to improve contrast in images	Xilinx... (2019)
IDCT	Image Processing	Inverse Discrete Cosine Transform	Xilinx... (2019)
Jacobi 1D	Mathematical	1-D Jacobi stencil computation	Perina, Becker and Bonato (2019)
Jacobi 2D	Mathematical	2-D Jacobi stencil computation	Perina, Becker and Bonato (2019)
Kmeans	Machine Learning	Clustering algorithm used in data-mining and machine learning	Cong et al. (2018a)
KNN	Machine Learning	Finds the k-nearest neighbors from an unstructured data set	Cong et al. (2018a)
MD5	Cryptography	MD5 hashing algorithm	Liu, Bayliss and Constantinides (2015)
Median Filter	Image Processing	Removes noises in images	Xilinx... (2019)
MxM	Mathematical	Linear Algebra Matrix Multiplication	Zhong et al. (2016)
NW	Genomics	Needleman-Wunsch Nonlinear global optimization method for DNA sequence alignments.	Cong et al. (2018a)
Optical Flow	Video Processing	Computes the movement of pixels in image frames.	Zhou et al. (2018)
Pathfinder	Graph Processing	Dynamic programming to find a path on a 2-D grid	Cong et al. (2018a)
Pivot	Mathematical	Pivot operation in gaussian elimination	Liu, Bayliss and Constantinides (2015)
RowCol	Mathematical	2D row column multiplication	Liu, Bayliss and Constantinides (2015)
Seidel	Mathematical	2-D Seidel Stencil Computation	Zhong et al. (2016)
Spam Filter	Machine Learning	Classifies emails as "spam" or "ham". Trains a Logistic Regression Model, using a Stochastic Gradient Descent Model	Zhou et al. (2018)
SRAD	Image Processing	Speckle Reducing Anisotropic Diffusion. Diffusion method for ultrasonic and radar imaging applications	Cong et al. (2018a)
Syr2k	Mathematical	Symmetric rank-2k operations	Perina, Becker and Bonato (2019)
Syrk	Mathematical	Symmetric rank-k operations	Perina, Becker and Bonato (2019)
Trisolv	Mathematical	Linear Algebra Triangular Matrix Solver	Liu, Bayliss and Constantinides (2015)
Watermarking	Image Processing	Adds watermarking to images	Xilinx... (2019)

APPENDIX C — BENCHMARKS' ADDITIONAL DATA

Table C.1: Benchmark set 1 FPGA power consumption and acceleration.

Benchmark Set 1	Power (W)					Accel. (Max)	Benchmark Set 1	Power (W)					Accel. (Max)
	U200	U50	1140T	870T	410T			U200	U50	1140T	870T	410T	
3DRendering	3.9W	3.6W	2.5W	3.1W	2.1W	3.42x	MD5	4.7W	4.4W	3.3W	3.8W	2.7W	1.14x
ADI	5.3W	4.9W	3.2W	3.8W	2.7W	1.10x	Median Filter	5.6W	5.3W	3.4W	4.0W	2.9W	42.01x
Convolve	5.4W	5.1W	3.4W	3.9W	2.9W	3.82x	MxM	7.9W	7.5W	4.8W	5.3W	4.2W	2.38x
Digit Rec.	4.6W	4.4W	3.3W	3.8W	2.9W	9.85x	Optical Flow	9.0W	8.6W	5.4W	5.9W	4.9W	3.16x
FIR	4.4W	4.1W	2.8W	3.4W	2.3W	2.33x	Pivot	9.9W	9.5W	6.6W	7.1W	5.9W	0.56x
Face Detection	5.1W	4.8W	3.5W	3.9W	2.8W	1.92x	RowCol	5.6W	5.3W	3.3W	3.9W	2.8W	1.84x
FloydWarshall	3.6W	3.4W	2.3W	2.9W	1.8W	9.36x	Seidel Filter	4.3W	4.0W	2.7W	3.3W	2.2W	15.52x
Histogram	7.4W	7.0W	4.7W	5.2W	4.2W	4.77x	Spam Filter	10.9W	10.4W	6.5W	7.1W	6.0W	1.52x
IDCT	14.2W	13.7W	8.2W	8.7W	-	3.18x	TriSolv	3.7W	3.5W	2.3W	2.9W	1.9W	0.21x
KMeans	11.7W	11.2W	7.0W	7.5W	6.4W	3.83x	Watermark	4.0W	3.7W	2.5W	3.1W	2.0W	18.81x

Table C.2: Benchmark set 2 FPGA power consumption and acceleration.

Benchmark Set 2		Power (W)					Accel. (Max)	Benchmark Set 2		Power (W)					Accel. (Max)
		U200	U50	1140T	870T	410T				U200	U50	1140T	870T	410T	
ADI	Full Energy	4.3W	4.0W	2.7W	3.2W	2.2W	1.05x	KNN	Full Energy	3.9W	3.6W	2.4W	2.9W	1.9W	0.05x
	Full Performance	5.3W	4.9W	3.2W	3.8W	2.7W	1.10x		Full Performance	7.3W	6.9W	4.8W	5.4W	4.3W	0.06x
Atax	Full Energy	4.6W	4.3W	3.2W	3.8W	2.8W	1.44x	MD5	Full Energy	3.7W	3.4W	2.3W	2.9W	1.9W	1.88x
	Full Performance	7.1W	6.8W	5.3W	5.8W	4.1W	1.68x		Full Performance	4.7W	4.4W	3.3W	3.8W	2.7W	2.05x
Backp.	Full Energy	4.4W	4.1W	3.0W	3.6W	2.6W	0.58x	NW	Full Energy	4.4W	4.1W	3.0W	3.5W	2.5W	4.36x
	Full Performance	7.8W	7.4W	6.5W	6.9W	5.8W	0.71x		Full Performance	7.2W	6.9W	4.8W	5.3W	4.3W	6.53x
Bicg	Full Energy	3.7W	3.4W	2.5W	3.0W	2.1W	1.96x	Pathf.	Full Energy	5.8W	5.4W	3.5W	4.0W	-	0.47x
	Full Performance	4.2W	3.9W	2.9W	3.5W	2.4W	2.04x		Full Performance	8.8W	8.5W	5.2W	5.8W	-	0.57x
CFD	Full Energy	4.0W	3.7W	2.5W	3.1W	2.0W	4.89x	Pivot	Full Energy	3.8W	3.5W	2.4W	3.0W	2.0W	0.39x
	Full Performance	13.3W	12.8W	8.5W	9.0W	7.8W	6.88x		Full Performance	9.9W	9.5W	6.6W	7.1W	5.9W	0.56x
DCT	Full Energy	4.1W	3.8W	2.6W	3.1W	2.1W	1.62x	RowCol	Full Energy	3.8W	3.4W	2.4W	3.0W	2.0W	1.56x
	Full Performance	4.3W	3.9W	2.6W	3.2W	2.1W	1.68x		Full Performance	5.6W	5.3W	3.3W	3.9W	2.8W	1.84x
Ges.	Full Energy	4.0W	3.7W	2.5W	3.1W	2.1W	1.08x	Seidel	Full Energy	3.7W	3.4W	2.3W	2.9W	1.9W	13.86x
	Full Performance	5.8W	5.5W	3.8W	4.4W	3.3W	1.10x		Full Performance	4.3W	4.0W	2.7W	3.3W	2.2W	15.52x
$Heat_{3D}$	Full Energy	3.7W	3.5W	2.3W	2.9W	1.9W	1.09x	Srad	Full Energy	9.5W	9.0W	7.0W	7.5W	6.5W	20.28x
	Full Performance	5.9W	5.6W	3.9W	4.4W	3.3W	1.25x		Full Performance	17.7W	17.0W	12.2W	12.6W	11.3W	23.87x
Jac_{1D}	Full Energy	3.7W	3.4W	2.3W	2.9W	1.9W	2.82x	Syr2k	Full Energy	4.9W	4.6W	3.5W	4.1W	3.0W	1.52x
	Full Performance	4.3W	4.0W	2.9W	3.4W	2.4W	3.09x		Full Performance	7.7W	7.3W	5.7W	6.2W	5.1W	1.60x
Jac_{2D}	Full Energy	4.1W	3.8W	2.8W	3.4W	2.3W	1.18x	Syrk	Full Energy	3.7W	3.4W	2.5W	3.0W	2.0W	1.56x
	Full Performance	9.3W	9.0W	7.0W	7.5W	6.4W	1.48x		Full Performance	5.5W	5.2W	4.1W	4.6W	3.5W	1.81x

Figure C.1: Benchmarks energy consumption over different FPGA architectures.

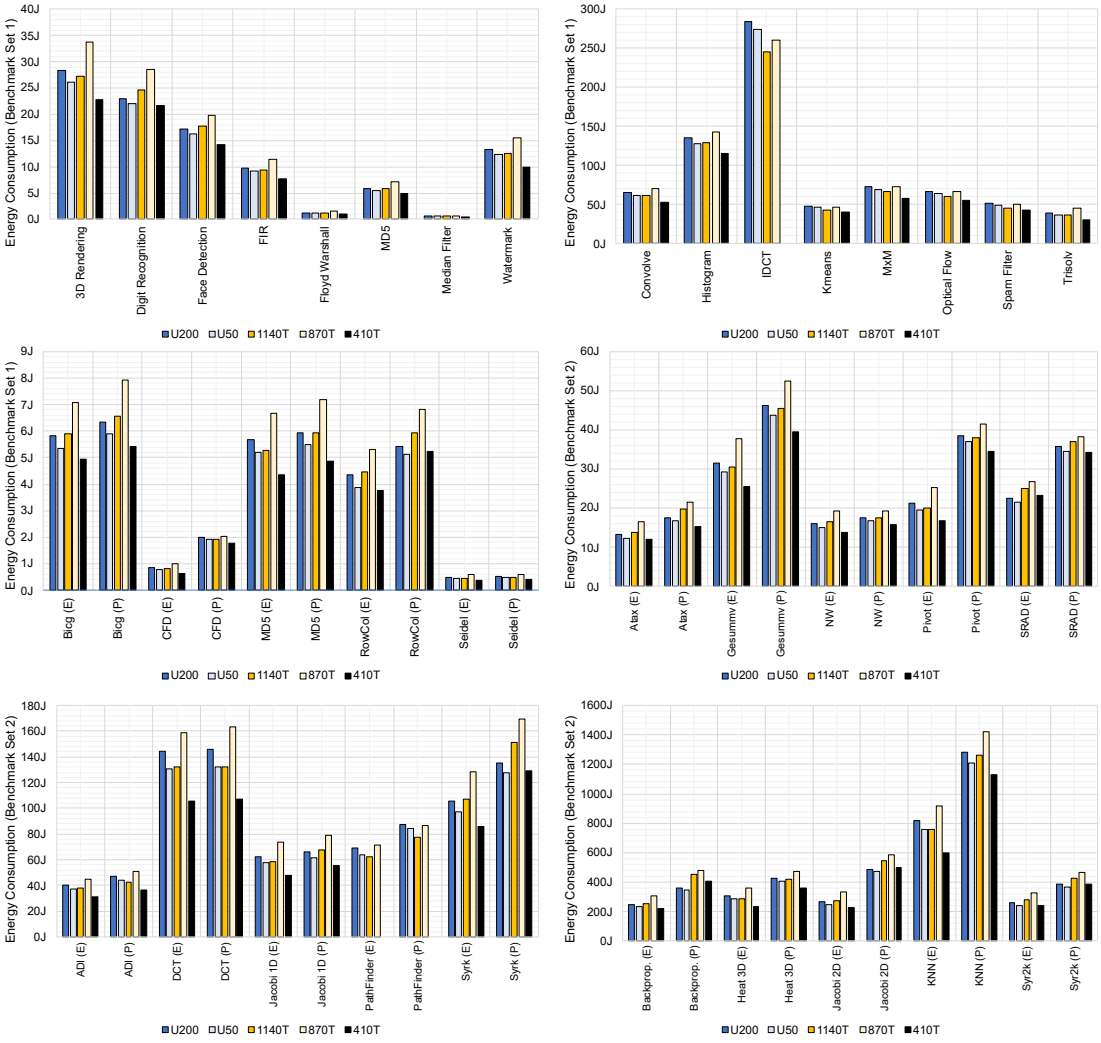


Figure C.2: Benchmarks execution time over variant DVFS levels for the AMD 3800x and AMD 3990x architectures.

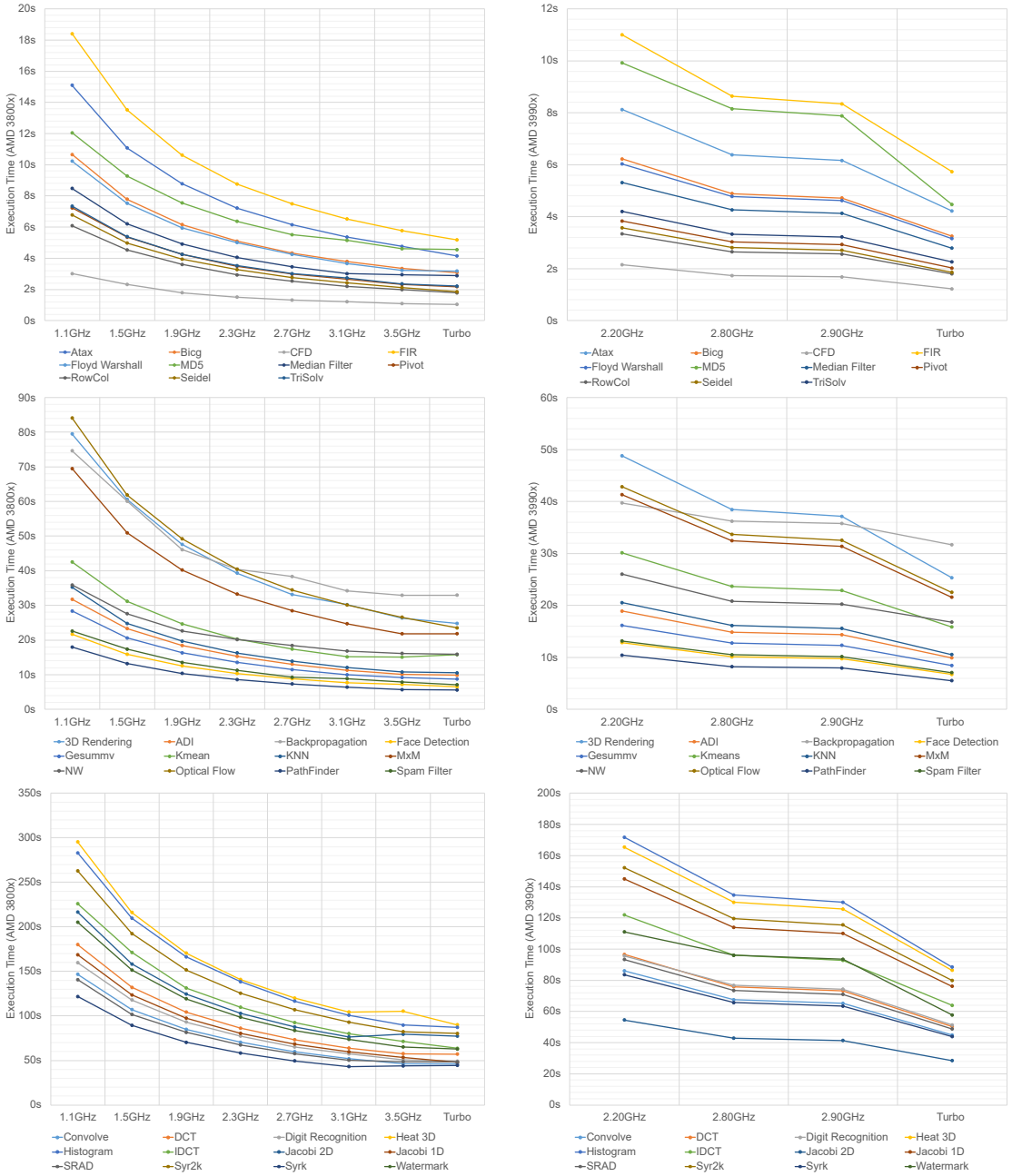
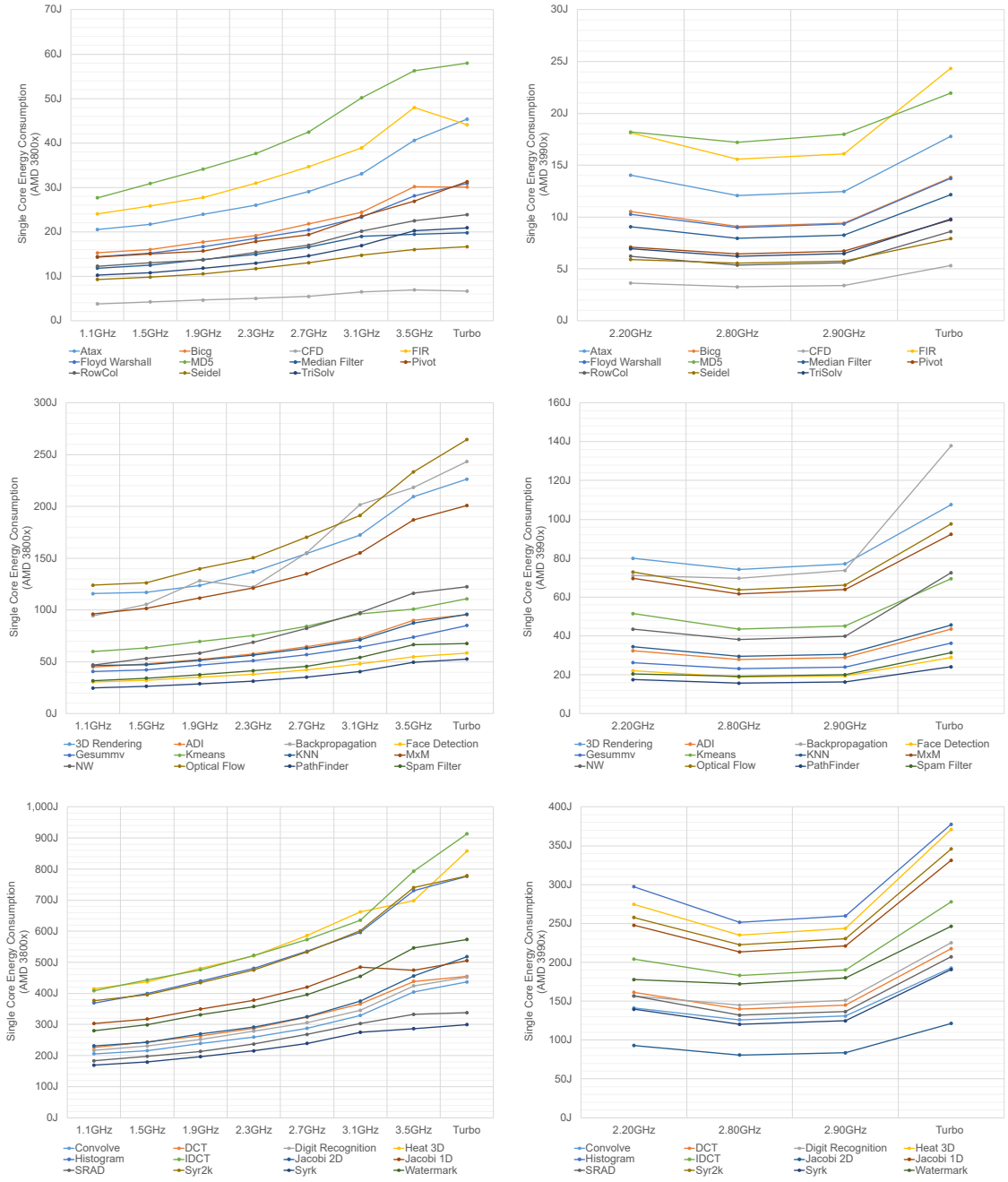


Figure C.3: Single core energy over variant DVFS levels for the AMD 3800x and AMD 3990x architectures.



APPENDIX D — RESUMO EXPANDIDO EM PORTUGUÊS

Este apêndice apresenta de forma resumida esta tese de doutorado, intitulada "Framework de provisionamento de recursos para ambientes CPU-FPGA com uso adaptativo e sinérgico de HLS-Versioning e DVFS".

D.1 Introdução

Com a crescente demanda por offloading e software-as-a-service, empresas como Microsoft, Amazon, Alibaba e Huawei tem investido em aceleradores FPGA para atender às necessidades de alta performance com eficiência energética em diversas aplicações, como redes neurais, análise de *big data* e computação de alto desempenho. Esses aceleradores são geralmente implantados junto de dispositivos CPU, expandindo ainda mais as possibilidades de otimização de software por meio de computação colaborativa, que permite que a CPU e o FPGA trabalhem juntos para executar tarefas, aproveitando as forças de cada dispositivo para obter uma execução eficiente. Nas infraestruturas colaborativas modernas de Nuvem, a execução multi-inquilino (do Inglês *multi-tenant*) tem sido empregada para otimizar o uso de recursos, em que vários clientes (inquilinos) compartilham os recursos da infraestrutura. Neste cenário o provisionamento de recursos deve ser bem equilibrado, explorando as capacidades de otimização oferecidas pelas arquiteturas CPU e FPGA para otimizar desempenho e energia. Para isso, diversos trabalhos adotam técnicas de gerenciamento de energia, como DVFS, ao lado da CPU, enquanto outros usam HLS para explorar os benefícios de diferentes otimizações de hardware no lado do FPGA, tornando possível gerar diferentes versões do mesmo projeto com a mesma funcionalidade, porém com características diferentes de desempenho, energia e consumo de recursos.

D.2 Motivação e Contribuições

Embora as técnicas de DVFS e HLS-Versioning tenham sido amplamente utilizadas de forma independente, elas nunca foram exploradas cooperativamente para melhorar ainda mais a eficiência de provisionamento de recursos. Portanto, esta tese propõe a ferramenta RAHD (abreviatura do Inglês ***R***esource Provisioning Framework for CPU-FPGA Environments with ***A***daptive ***H***LS-Versioning and ***D***VFS) que une as técnicas de

DVFS, HLS-Versioning e computação colaborativa CPU-FPGA. Nosso método é projetado para ambientes da Nuvem, onde várias solicitações de tarefas devem ser provisionadas nas arquiteturas CPU e FPGA. Ao contrário de todas as pesquisas anteriores, esse provisionamento colaborativo é usado juntamente com técnicas de otimização sem reduzir o potencial do provisionamento. No lado da FPGA, o HLS-Versioning é usado para permitir a seleção de designs otimizados para a execução FPGA em tempo de execução (por exemplo, orientado para desempenho ou energia). O uso de uma determinada versão do HLS implica na alteração das propriedades da carga de trabalho. Nesse cenário, nossa solução é a única a fornecer a adaptabilidade necessária para extrair o máximo dos benefícios das versões otimizadas, uma vez que nosso provisionamento adapta-se dependendo das características da carga de trabalho. Por outro lado, nossa tese emprega sinergicamente DVFS na CPU para reduzir ainda mais o consumo de energia sem prejudicar o tempo total do provisionamento de tarefas.

Além disso, nosso provisionamento é sempre executado em um tempo viável, levando em consideração a duração da carga de trabalho (ou seja, cargas de trabalho de curta ou longa duração). Para cobrir todos os comportamentos, este framework conta com múltiplas estratégias de provisionamento. Em tempo de execução, as técnicas mais adequadas são selecionadas por meio de um árbitro implementado através de árvores de decisão que consideram as propriedades da carga de trabalho e da arquitetura nas suas escolhas. Em resumo, destacamos as seguintes contribuições:

- Mostramos que diferentes estratégias de provisionamento de recursos (com diferentes compensações entre qualidade de solução e tempo de convergência) são mais adequadas dependendo das características da arquitetura e carga de trabalho;
- Mostramos que HLS-Versioning é obrigatório para maximizar o desempenho ou a eficiência energética, afetando também o potencial das estratégias de provisionamento de recursos, uma vez que altera as características da carga de trabalho;
- Mostramos que DVFS deve ser empregado de forma sinérgica para trazer ganhos de energia sem prejudicar os benefícios de desempenho/energia trazidos pelo HLS-Versioning e otimização do provisionamento de recursos;
- Mostramos que o provisionamento de recursos deve ser totalmente adaptativo, já que as propriedades das cargas de trabalho recebidas mudam devido à heterogeneidade inerente das cargas de trabalho e ao uso do HLS-Versioning;
- Propomos uma ferramenta capaz de explorar todos os pontos supracitados, ex-

traindo o máximo potencial de ambientes CPU-FPGA através de um provisionamento de tasks totalmente adaptativo que potencializa o uso da técnica HLS-Versioning e expande os ganhos de energia através do uso de DVFS.

D.3 Resultados Experimentais

Resultados experimentais foram obtidos através do conjunto de aplicações listadas nas Tabelas 5.2 e 5.3, utilizadas para criar sete cenários de carga de trabalho da Nuvem (também listadas nas tabelas). Além disso, foram consideradas dez diferentes combinações arquiteturais CPU-FPGA, através dos dispositivos apresentados na Tabela 5.1. Através destes múltiplos cenários, visamos reproduzir a alta heterogeneidade das cargas e arquiteturas encontradas na Nuvem. Os experimentos avaliaram os eixos de otimização do RAHD individualmente e como um todo.

Em um primeiro momento, nós avaliamos os benefícios do uso de um provisionamento colaborativo multi-inquilino. Estes experimentos mostraram as vantagens do compartilhamento dos recursos CPU-FPGA, possibilitando a maximização dos mesmos. Também destacaram a importância de se usar computação colaborativa para otimizar ainda mais desempenho e energia. Comparado a um provisionamento não-colaborativo inquilino-único, onde as cargas de cada inquilino são distribuídas ao dispositivo FPGA sequencialmente (isto é, as cargas de inquilinos diferentes não são executadas em paralelo), o ambiente colaborativo multi-inquilino atingiu ganhos de energia de mais de 10x.

Em um segundo momento, estudamos a importância da escolha pelas estratégias de provisionamento mais adequadas para cada carga de trabalho. Para tais experimentos, todas estratégias de provisionamento do RAHD foram testadas considerando os diversos cenários de carga de trabalho listadas nas Tabelas 5.2 e 5.3. Nossos resultados apontaram que diferentes estratégias eram mais adequadas para determinadas cargas de trabalho, dependendo do objetivo de otimização (isto é, desempenho ou energia), das características das cargas, da arquitetura alvo, e do tempo de convergência das estratégias. Além disso, observamos que a escolha da estratégia não adequada poderia resultar em degradações de desempenho de até 6x e de energia de até 9x. Esses experimentos nos ensinaram a importância de adaptar a estratégia de provisionamento para cargas de trabalho recebidas. Para isso, propomos árvores de decisão que detectam a melhor estratégia de provisionamento para cada carga de trabalho para maximizar o desempenho ou minimizar a energia.

A partir disto, no experimento seguinte estudamos a eficácia das árvores de de-

cisão usadas pelo RAHD para seleção dinâmica de estratégias de provisionamento. Nestes experimentos, nenhum dos eixos de otimização foram habilitados (isto é, nem DVFS e nem HLS-Versioning). Para tal estudo, comparamos o RAHD com o uso de estratégias fixas (isto é, uma única estratégia utilizada para todas cargas, sem adaptabilidade) e um oráculo que sempre seleciona as melhores estratégias para uma determinada carga de trabalho. Os resultados apontaram que o RAHD consegue atingir soluções muito próximas do oráculo, ficando apenas 1% atrás do oráculo em termos de desempenho, e 2% em termos de energia. Em comparação ao uso de estratégias fixas, este ficou, em média, 27% a frente da estratégia fixa que apresentou os melhores resultados gerais para energia.

Durante o terceiro experimento, exploramos os efeitos da otimização DVFS quando aplicada de maneira não adaptativa e adaptativa. Em um primeiro momento, utilizamos os perfis de DVFS com a frequência mais baixa e de forma fixa, para ver o impacto do uso de um perfil não adaptativo. Tal abordagem mostrou penalidades de mais de 2x em termos de desempenho e 1.2x em energia. Por outro lado, o uso adaptativo dos perfis DVFS providos pelo RAHD reduziram a energia dos provisionamentos em até 22%, sem nenhuma penalidade ao tempo total de execução da carga. Desta maneira, os experimentos nos mostraram que o DVFS deve ser adaptado sinergicamente para fornecer benefícios de energia sem prejudicar o desempenho e a eficiência energética do provisionamento.

No quarto experimento, foi explorado o HLS-Versioning no ambiente proposto, onde os resultados mostraram que o RAHD gerou versões especializadas que aumentaram significativamente a eficiência energética e de desempenho dos designs individuais. Utilizando uma estratégia sofisticada junto com designs especializados, foi possível obter ganhos de energia de cerca de 30x em comparação com a estratégia FCFS com designs não especializados e 6x em comparação com a estratégia FCFS com designs especializados. Isto destacou a importância de um provisionamento robusto para extrair o máximo benefícios desses designs.

Finalmente, o último experimento avaliou o RAHD como um todo, mostrando sua eficiência em unir HLS-Versioning, DVFS e provisionamento adaptativo de recursos. Nossos experimentos mostraram que o RAHD pode extrair os benefícios fornecidos por cada otimização sem nunca reduzir a eficácia do provisionamento. Comparado a estratégia FCFS sem o uso de otimizações, RAHD apresentou, em média, ganhos de desempenho de 15x e de energia de 50x. Com relação a um oráculo, que sempre seleciona a melhor estratégia e considera DVFS e HLS-Versioning, este ficou apenas 4% atrás em termos de desempenho, e 7% atrás em termos de energia.