ML-Guided Conflict-Aware Scheduling for FPGA-Based Acceleration in the Cloud-Edge Continuum

Juan Encinas, Alfonso Rodríguez and Andrés Otero
Centro de Electrónica Industrial, Universidad Politécnica de Madrid
Madrid, Spain
{juan.encinas, alfonso.rodriguezm, joseandres.otero}@upm.es

***Abstract*—This paper presents a conflict-aware, Machine Learning (ML)-guided workload optimization methodology for FPGA-based Acceleration-as-a-Service (AaaS) platforms operating in dynamic cloud-edge environments. Traditional scheduling techniques rely on static, design-time profiling, which is not well suited to changing workloads and unpredictable interference between hardware accelerators. To address this limitation, a run-time scheduling strategy is proposed that uses incrementally trained, lightweight ML-based models to characterize workload behavior from execution traces. These predictions are then used to drive a population-based metaheuristic search, implemented with the Crow Search Algorithm, in order to select task configurations and replication levels that reduce resource conflicts and shorten workload execution time. Experimental results show that the proposed methodology achieves significant reductions in workload makespan compared to a baseline First come, first served approach, even when the overhead of model training and optimization is taken into account.**

***Index Terms*—Machine Learning for Tuning Hardware, Run-Time Workload Optimization, Conflict-Aware Scheduling, Acceleration-as-a-Service, Cloud-Edge Continuum**

I. Introduction

As cloud and edge computing infrastructures continue to grow [1], the cloud-edge continuum is receiving increasing attention [2]. In this setting, centralized cloud platforms and distributed edge devices are combined into a unified computing substrate, so that applications can exploit low latency and improved data locality at the edge while still benefiting from the scalability and large compute capacity of the cloud.

Field-Programmable Gate Arrays (FPGAs) are becoming important components across this continuum, from edge devices [3] to large cloud deployments [4]. They provide high computational throughput with significantly lower power consumption than general-purpose Central Processing Units (CPUs) and Graphics Processing Units (GPUs), while preserving some flexibility thanks to Dynamic and Partial Reconfiguration (DPR). In many systems, FPGAs are exposed as AaaS devices [5], so that heterogeneous workloads can be offloaded to reconfigurable accelerators.

Taking full advantage of these reconfigurable resources in the cloud-edge continuum requires solving a challenging scheduling problem. The system must decide which tasks are offloaded to the FPGA fabric, how many replicas of each task are instantiated (i.e., the level of parallelism), and when they are executed [6]. This problem is NP-hard [7] and belongs to the class of combinatorial optimization problems [8]. In slot-based acceleration frameworks such as the one considered in this work, the number of possible configurations grows rapidly with the number of tasks and reconfigurable regions, making exhaustive search infeasible under strict latency constraints.

Most existing approaches address this issue by relying on design-time workload profiling [9]. Task characteristics, such as execution time, power consumption, or communication overhead, are measured in advance under controlled conditions. The profiling data are then used either to build fixed, precomputed schedules, or to guide run-time heuristics that operate assuming static workloads that are known beforehand.

In contrast, workloads in the cloud-edge continuum are typically dynamic and often unpredictable [2]. Tasks can change their behavior due to input or environmental variations and may target heterogeneous platforms with different capabilities. In this context, extensive design-time profiling is not only impractical but may also fail to capture the variability of real deployments, for instance the interaction between different accelerator flavors that share the FPGA fabric.

This work proposes a data-driven workload optimization methodology for reconfigurable multi-accelerator systems that aims to reduce workload makespan and improve energy efficiency. Previous results have shown that conflicts between tasks executed in parallel can increase execution time by up to 500% due to resource contention [10]. Based on that observation, a conflict-aware scheduling approach guided by run-time ML-based workload characterization is introduced. In particular, lightweight predictive models are leveraged, which are trained incrementally using execution time measurements collected during system operation, also capturing the interaction between concurrently running accelerators [11]. These models provide online estimates of performance and interference that the scheduler exploits to adapt its decisions to changing workloads and operating conditions without prior knowledge or manual tuning, effectively enabling a self-adaptive behavior.

Efficient exploration of the scheduling solution space remains a key difficulty. To tackle it, this work employs the Crow Search Algorithm (CSA) metaheuristic [12] instead of traditional heuristic-based strategies. Metaheuristics offer high-level search schemes suitable for complex optimization problems and are largely problem-independent [13], [14]. This makes them attractive in cloud-edge scenarios, where scheduling and resource management policies should be portable across diverse platforms.

The rest of this paper is organized as follows. Section II reviews related work. Section III summarizes the workload management infrastructure and the run-time ML-based characterization used in this study. The proposed conflict-aware scheduling strategy is described in Section IV. Section V presents the experimental evaluation, and Section VI concludes the paper and outlines future work.

II. State of the Art

The predominant strategy to address workload optimization in FPGAs is to perform an a priori characterization of the tasks to be accelerated and then use this information at run time to guide scheduling decisions [9]. Most existing proposals focus on design-time knowledge of primarily Directed Acyclic Graph (DAG)-based workloads and target objectives such as makespan reduction [15]–[21], power minimization [22], or fair resource allocation [23], [24].

Although makespan optimization is the most common goal, the specific strategies differ. In [15] the authors apply reconfiguration prefetching, where the configuration of a future task is overlapped with the execution time of the current one [25]. They extend this idea to several DAGs, prefetching tasks from future graphs while the current graph is running, which requires prior knowledge of multiple DAGs but yields an improvement of almost 22% in makespan. In [16], latency information for each task in a DAG is used by a heuristic that partitions the tasks into groups. Each group contains one longer task that hides the execution time of the remaining ones when they are executed together on the FPGA. In [17], memory-related information per task (e.g., required bandwidth and number of memory accesses) is exploited to identify sets of tasks with complementary memory access patterns that can run simultaneously in the FPGA fabric with reduced interference. Their results report up to 65% improvement over earlier approaches.

Other works focus on heterogeneous systems and the decision of whether a task should run on the FPGA or remain in software. In [18], the scheduler offloads tasks to CPUs, GPUs, or FPGAs depending on the expected benefit, using predefined task profiles combined with run-time resource information. Their strategy selects the set of computing nodes that best suits each workload and achieves a speedup of $1.25\times$ in workload execution. A similar philosophy is followed in [19], where a heuristic selects both the target device (e.g., FPGA or CPU) and the number of parallel instances per task based on design-time profiling, obtaining a 20% reduction in makespan with respect to conventional scheduling techniques.

Varying the task capabilities has also been explored as a way to improve performance. The work in [20] constructs, at design time, a solution space that includes multiple configurations for each task, which trade performance against Quality of Service (QoS). At run time, if the application deadline is not satisfied, the heuristic switches to versions with lower QoS (e.g., lower video quality, since they target video processing) and faster execution. Along the same line, the proposal in [21] uses task configurations with different degrees of parallelism. Their goal is to avoid starvation while maximizing overall performance. To that end, tasks are initially executed with high parallelism, but can be replaced by less parallel and less resource-intensive variants when additional tasks need to be admitted. As before, this approach relies on prior knowledge of task behavior and requires profiling of all configurations.

Power optimization has also been considered. In [22], design-time profiling is used to select, at run time, combinations of tasks that maximize the number of parallel accelerators per task while minimizing the overall power consumption when those tasks run concurrently.

Several works address fairness in resource usage through heuristic algorithms that explicitly consider the FPGA resources consumed by each task at run time. In [23], the scheduler tracks the utilization of the FPGA fabric per task and assigns priorities inversely proportional to this utilization metric, updating them periodically. This approach leads to higher FPGA utilization and performance than round-robin counterparts, while still promoting shared use of resources. A similar idea is applied in [24], although the focus is placed on different tenants rather than individual tasks, again obtaining better utilization ratios.

In summary, most scheduling strategies for FPGAs, regardless of whether they target performance, power consumption, or fairness, depend mainly on design-time information that is later used at run time. While this can be sufficient in static scenarios, it overlooks critical run-time aspects such as the dynamic interaction between tasks that share the FPGA fabric. For dynamic workloads, such as those found in the AaaS paradigm, some form of run-time adaptivity is required.

The approach proposed in this paper follows a different direction and targets dynamic workloads. Instead of extensive design-time profiling, which assumes deep prior knowledge of the workload, it relies on ML-based models to characterize the workload at run time. This characterization is then used on the fly to guide the optimization process. The scheduler explicitly focuses on reducing the interaction between kernels running in parallel, which has been largely neglected by state-of-the-art solutions even though its impact on performance is significant [10]. Finally, the proposed method is built around a metaheuristic algorithm rather than a heuristic one, since metaheuristics provide problem-independent search strategies that are more easily generalizable and therefore better suited to the heterogeneous cloud-edge continuum [14].

III. Background

The main contribution of this paper is the proposed workload optimization method. However, it relies on two pre-existing components: a workload management infrastructure that offloads tasks to the FPGA fabric and records their behavior, and an incremental modeling methodology for run-time workload characterization. Both are summarized in this section to provide context. A detailed description can be found in [10] and [11].

A. Workload Management Infrastructure

The infrastructure has two main roles: offloading tasks from the workload to the FPGA and collecting power and performance traces during system operation.

a) Workload Offloading: The workloads are handled as collections of monolithic tasks rather than DAGs, although DAGs could also be supported. A two-queue structure is used. Incoming tasks are first placed in a waiting queue and then moved to a scheduling queue when they become schedulable according to the scheduling policy and resource availability. Each schedulable task is implemented on the FPGA fabric using the ARTICo^3 hardware acceleration framework [26], which partitions the reconfigurable area of the device into independent reconfigurable slots. This structure enables task-level parallelism (different tasks in different slots), data-level parallelism (several replicas of the same task mapped to multiple slots), or a combination of both, thanks to DPR.

b) Workload Registration: While tasks are running in the different reconfigurable slots, a monitoring process periodically records performance traces from the accelerators, for example start and stop signals, similar in spirit to a Xilinx Integrated Logic Analyzer (ILA) [27] but implemented on device and without the need for an external host. In addition, an external board with Analog to Digital Converter (ADC) capabilities is used to measure power consumption during operation. The result is a set of synchronized power and performance traces that capture the behavior of the various task combinations running concurrently on the FPGA.

B. Run-Time Workload Characterization

The traces provided by the workload management infrastructure serve as input to build predictive ML-based models. These models are trained on the recorded power and performance data and updated incrementally at run time. As tasks are offloaded to the FPGA, new measurements are collected and incorporated into the models, so that they follow possible changes in the system or environment, which is common in cloud-edge scenarios. The models implement lightweight regression algorithms with specific learning procedures (described in [11]) that keep training overhead low and make run-time prediction of power and execution time feasible even on modest platforms.

In this work, these two building blocks are combined as follows. The workload management infrastructure offloads tasks to the FPGA while incrementally feeding power and performance traces to the data-driven models. On top of this, a scheduler based on a metaheuristic algorithm is used to close the loop. It queries the models at run time to obtain predictions and uses them to guide task scheduling decisions with the goal of reducing workload makespan and improving energy efficiency. An overview of the complete methodology is shown in Figure 1.

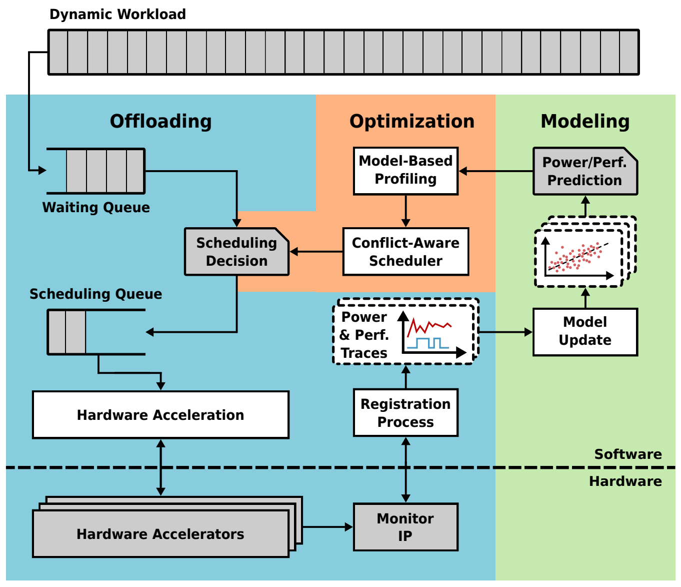


Fig. 1: Overview of the workload optimization methodology.

IV. Conflict-Aware Scheduling

This section describes the scheduling problem addressed in this work, the proposed solution, and its implementation.

A. Optimization Problem Description

The goal is to schedule hardware acceleration tasks on the FPGA. In particular, the scheduler must decide, at run time and whenever new tasks arrive or resources become available, which combination of tasks should be offloaded to the FPGA fabric. In a slot-based reconfigurable architecture such as the one used here, this is equivalent to selecting a configuration of slots, i.e., choosing which tasks run together and how many replicas of each task are instantiated, exploiting both data-level and task-level parallelism through DPR.

Task execution on the FPGA is assumed to be non-preemptive, since the overhead of preemption would generally offset its benefits. Once a task starts, it runs until completion before the corresponding slots can be reused. As a result, each scheduling decision must take into account both tasks that are already running and tasks in the waiting queue, and select a subset of waiting tasks that yields a good long-term outcome.

The same procedure is repeated every time a new task appears or a reconfigurable slot is freed in the FPGA fabric.

B. Proposed Solution

To determine suitable configurations, this work exploits the run-time workload characterization described in Section III. Each candidate schedule (i.e., each configuration of tasks and replicas) is evaluated using predictions provided by the ML-based models. In particular, the models estimate the performance degradation caused by resource contention among tasks that run in parallel, which can significantly affect overall execution time [10]. The scheduler then seeks configurations that reduce this interaction, with the expectation that consistently choosing such configurations will lead to shorter workload makespans in the long run.

The solution space, however, grows quickly with the number of tasks and available slots. A greedy method that exhaustively evaluates all possible configurations is therefore not feasible [8]. Instead, a metaheuristic optimization algorithm is used to explore only a subset of the solution space. Among the many metaheuristic families (e.g., population-based, evolutionary, or physics-inspired algorithms), this work focuses on population-based methods, which have shown good results in complex scheduling problems of this type [28].

The specific metaheuristic selected is the CSA [12]. It has been successfully applied to different optimization domains and has shown competitive performance compared to alternatives such as Particle Swarm Optimization (PSO) [29] and Ant Colony Optimization (ACO) [30]. In addition, its implementation is relatively simple and lightweight [31], which is an important property in the target scenario, where run-time scheduling optimization, workload characterization, and hardware acceleration already coexist on the same platform.

C. Scheduler Implementation

The CSA is inspired by the behavior of crows [12]. Crows hide food in specific locations and remember those locations. They also observe other crows to locate their hiding places, and may react by moving their own food if they notice that they are being followed. A useful feature of the algorithm is that it only requires two tunable parameters: the flight length fl, which controls how far a crow can move in one iteration, and the awareness probability AP, which determines how likely a crow is to detect that it is being followed. Together, these parameters control the exploration/exploitation ratio.

In qualitative terms, the algorithm works as follows. In each iteration, each crow decides whether to follow another crow based on the awareness probability. If it does not follow any crow, it moves to a random position (exploration). If it follows another crow, it moves to a position near the other crow's hiding place, visiting areas that are already known to be promising (exploitation).

The pseudocode of CSA is shown in Algorithm 1 and can be summarized as follows:

1) The initial positions of the crows are randomly generated in a d-dimensional space, where d is the number of variables to optimize (line 1).
2) Each initial position is evaluated, and the result is stored as the crow's memory (line 2). The memory m_i of crow i keeps the best position that crow has found so far.
3) To update its position, each crow x_i randomly selects another crow x_j to follow (line 5) and generates a random number r_i. If this random value is greater than the awareness probability AP, crow x_i moves towards the hiding place m_j of crow x_j (line 6).
   The new position of crow x_i is computed as (line 7):

$$x_{i,\text{iter}+1} = \begin{cases} x_{i,\text{iter}} + \\ r_i \times fl_{i,\text{iter}} \times \\ (m_{j,\text{iter}} - x_{i,\text{iter}}) & \text{if } r_j \geq AP_{j,\text{iter}} \\ \\ \text{a random position} & \text{otherwise} \end{cases} \quad (1)$$

   where $iter$ is the iteration index, $AP_{j,\text{iter}}$ is the awareness probability of crow x_j, r_j and r_i are random numbers that control, respectively, whether crow x_i follows crow x_j and how close it lands to m_j, and $fl_{i,\text{iter}}$ is the flight length of crow x_i.
4) The new position is then checked against the boundaries of the search space (line 9), and the objective function is evaluated (line 10). The objective or fitness function is defined according to the optimization problem under consideration.
5) Finally, the memory is updated (line 11) according to

$$m_{i,\text{iter}+1} = \begin{cases} x_{i,\text{iter}+1} & \text{if } f(x_{i,\text{iter}+1}) \leq f(m_{i,\text{iter}}) \\ m_{i,\text{iter}} & \text{otherwise} \end{cases} \quad (2)$$

   where $f(x_{i,\text{iter}+1})$ and $f(m_{i,\text{iter}})$ denote the fitness of the new position of crow x_i and its current memory, respectively.

Algorithm 1 CSA: Crow Search Algorithm

```
Input: $n$ Number of crows in the population
    $\text{iter}_{\max}$ Maximum number of iterations
Output: Best crow position found
 1: Initialize positions of crows
 2: Initialize crows' memory
 3: for $\text{iter} = 1$ to $\text{iter}_{\max}$ do
 4:     for each crow $i$ in the population do
 5:         Choose a random crow $j$
 6:         Determine the awareness probability $AP$
 7:         Update $x_{i,\text{iter}+1}$ using Equation 1
 8:     end for
 9:     Check solution boundaries
10:     Evaluate the fitness of each crow
11:     Update crows' memory using Equation 2
12: end for
```

Exploration occurs when a crow moves to a random position instead of following another crow. Exploitation happens when a crow moves towards the hiding place of another crow. The degree of exploitation depends on how close the new position is to the other crow's memory and is controlled by the flight length parameter fl. This is illustrated in Figure 2. If $fl < 1$, the new position lies between the current position (blue dot) and the hiding place of the other crow (orange dot). If $fl > 1$, the new position can be located beyond the other crow's memory. The random value r_i determines where the crow lands along the green arrow in Equation 1. This introduces

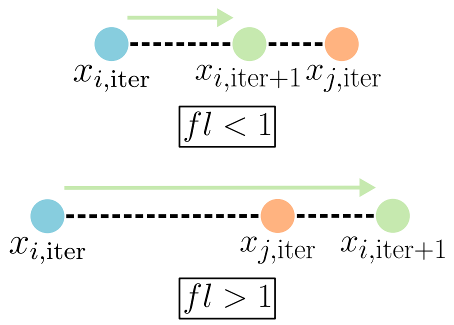


Fig. 2: Exploitation process of the crows.

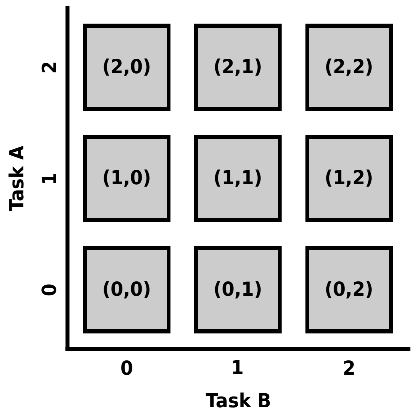


Fig. 3: Example of scheduling solution space (2 tasks, 3 slots).

a smooth transition between exploitation and exploration and helps reduce the risk of getting trapped in local minima.

D. Adaptation of CSA

To apply CSA to the considered scheduling, a few adaptations to the original formulation are required:

- The optimization variables correspond to the tasks that can be accelerated. For each task, the variable value is the number of parallel replicas to instantiate. This value ranges from zero to the total number of reconfigurable slots in the FPGA. Figure 3 illustrates a simplified example with two tasks and at most three available slots. In this case, the crows move in a two-dimensional solution space whose points represent all possible distributions of the three slots between tasks A and B.
- In contrast to the continuous formulation in the original CSA, the solution space here is discrete, since the number of replicas per task must be an integer. To deal with this, the boundary check step (line 9 in Algorithm 1) rounds each coordinate of the new crow position to the nearest integer. Furthermore, the sum of all replicas cannot exceed the number of available slots, so the boundary check also enforces this constraint by decrementing the largest variable until the total number of replicas fits within the available slots.

The scheduling process on the FPGA operates as follows. For clarity, the description starts from an arbitrary instant in which some tasks are already executing, some others are waiting, and a set of slots is free:

1) If there are free slots available in the FPGA fabric, the scheduler inspects the waiting queue to identify the tasks that can be offloaded and starts the scheduling process.
2) The tasks in the waiting queue define the optimization variables, and thus a d-dimensional solution space, where d is the number of waiting kernels.
3) The CSA runs for a fixed number of iterations with a given population size, exploring a subset of the solution space. Each crow position is evaluated by means of a fitness function, described in the next subsection.
4) The crow position that yields the best fitness value is selected, and the corresponding configuration of tasks and replicas is implemented on the FPGA.
5) The same procedure is re-executed every time a new task arrives or a slot becomes free.

E. Fitness Function

The fitness function quantifies the quality of the solutions found by CSA and is central to the effectiveness of the scheduler. In this work, a multi-objective fitness function is defined to capture two goals commonly addressed in the literature: reduction of workload makespan (and, indirectly, energy efficiency, given its direct relation to execution time) and fair allocation of resources. Lower fitness values correspond to better solutions.

a) Workload Makespan Optimization: As outlined in Section III, the scheduler uses the run-time ML-based workload characterization to predict the execution time of different combinations of tasks. When a crow position is evaluated, the models provide for each task in the waiting queue: (i) the predicted execution time if offloaded to the FPGA under the current configuration, and (ii) the predicted execution time if executed in isolation. The relative difference between these two values captures the expected impact of interaction on execution time for that task. This interaction is computed for all waiting tasks, assuming the scenario defined by the crow position (running tasks and number of accelerators per task). Since several waiting tasks may be scheduled simultaneously, the mean interaction across all scheduled tasks is used as a scheduling conflict term.

Scheduling no tasks would trivially result in no interaction. To avoid such degenerate solutions, an activity reward term is introduced that favors configurations using a higher fraction of available accelerators. The global conflict term is defined in Equation 3. Larger values correspond to worse fitness.

$$\text{Conflict} = \underbrace{\frac{1}{N}\sum_{i=1}^{N}\left(\frac{T_i^{\text{shared}} - T_i^{\text{alone}}}{T_i^{\text{alone}}}\right)}_{\text{Average Interaction Impact}} - \underbrace{\frac{Slots_{used}}{Slots_{total}}}_{\text{Activity Reward Term}} \tag{3}$$

b) Fair Resource Allocation: The second term in the fitness function captures how evenly resources are distributed among tasks. For each crow position, the standard deviation of the vector of assigned replicas per task is computed, as shown in Equation 4. A high standard deviation indicates that some tasks receive significantly more replicas than others, which represents an unbalanced situation. Again, higher values correspond to worse fitness.

$$\text{Fairness} = \text{std}\left(\{Slots_{T_1}, Slots_{T_2}, \ldots, Slots_{Tn}\}\right) \quad (4)$$

c) Weighted Combination: The final fitness value is obtained as a weighted linear combination of the normalized conflict and fairness terms, controlled by parameter α, as shown in Equation 5. Normalization enables a meaningful combination of both terms and allows the user to tune the relative importance of each objective. Larger values of α emphasize minimization of conflict, while smaller values give more weight to fairness.

$$\text{Fitness} = \alpha \cdot \text{Conflict} + (1 - \alpha) \cdot \text{Fairness} \quad (5)$$

V. Experimental Evaluation

This section evaluates the proposed conflict-aware scheduler using a set of hardware acceleration benchmarks. First, the experimental setup is described, then the configuration of the scheduler parameters, and finally the main results.

A. Experimental Setup

The proposed system is deployed on an AMD Zynq UltraScale+ ZCU102 board (XCZU9EG-FFVB1156-2-I), with the Programmable Logic (PL) running at 100 MHz and the Processing System (PS) at 1.2 GHz. Hardware accelerators are implemented using the workload management infrastructure described in Section III, with up to 8 simultaneous accelerators mapped to the FPGA fabric. The evaluation uses the MachSuite benchmark suite [32], which provides 19 algorithm variants across 12 Dwarf patterns [33]. Due to resource limitations, 8 kernels are excluded, but 9 of the 12 Dwarfs remain represented.

A synthetic workload of 60,000 acceleration requests is generated in three phases. The first 20,000 requests involve 4 MachSuite benchmarks, the next 20,000 involve 8 benchmarks, and the last 20,000 include all 11 selected benchmarks. Each phase contains an equal number of requests per benchmark, and the sequence of phases exercises the ability of the scheduler to adapt to evolving workloads. Task arrival times are drawn from a Poisson distribution, and the job size (i.e., the number of executions) is chosen as a power of two in the range from 512 to 2,048, with all sizes equally represented.

B. Scheduler Parameter Selection

The scheduler described in Section IV depends on several parameters that affect both the convergence of the optimization and the quality of the scheduling decisions. These parameters are the population size (number of crows), the maximum number of iterations, and the internal CSA parameters, namely the awareness probability AP and the flight length fl, which control the exploration and exploitation balance.

TABLE I: CSA parameter selection

CSA Parameter	**Selected Value**
Number of Crows	4
Number of Iterations	4
Awareness Probability (AP)	0.6
Flight Length (fl)	1.5

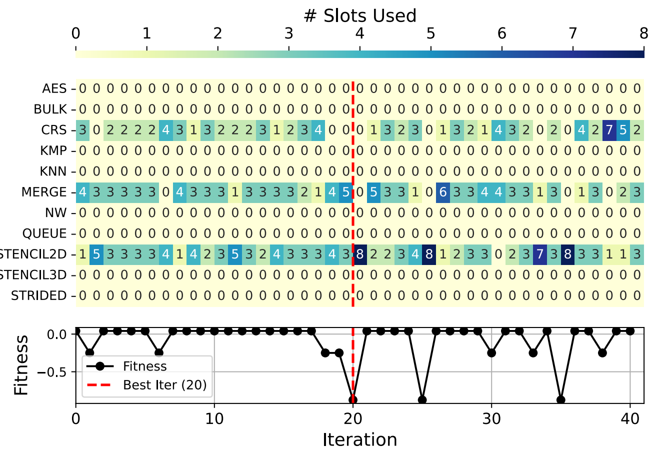
To select these parameters, a software simulation environment was developed that replays workload executions using the predictions produced by the run-time workload characterization models introduced in Section III. Since the goal is to tune CSA rather than to perform on-line scheduling, an already trained model is used. The simulator reconstructs an execution timeline for each kernel, taking into account overlaps and interaction. Whenever a kernel starts or finishes, the completion times of other running kernels are updated using the predictive models. This reproduces resource contention in a realistic way and allows different parameter combinations to be compared under similar conditions.

Using this simulator, a brief sensitivity analysis was carried out to identify robust configurations. Different combinations of AP and fl, together with several population sizes and iteration counts, were tested in terms of convergence speed and resulting fitness. As an illustrative example, Figure 4 shows the trajectory of a single crow during the optimization process. In this scenario, the scheduler must assign accelerators to three pending tasks (CRS, MERGE, and STENCIL2D) on an FPGA with eight available slots. The upper plots show, for different AP values, how the number of accelerators assigned to each task evolves per iteration, while the lower plots show the corresponding fitness values (only a representative case is depicted). High AP values such as 0.9 (Figure 4a) encourage exploration, since crows are more likely to avoid being followed, whereas low AP values such as 0.1 (Figure 4b) favor faster convergence. This behavior directly affects the ability of the scheduler to find good configurations, which justifies the preliminary parameter analysis.

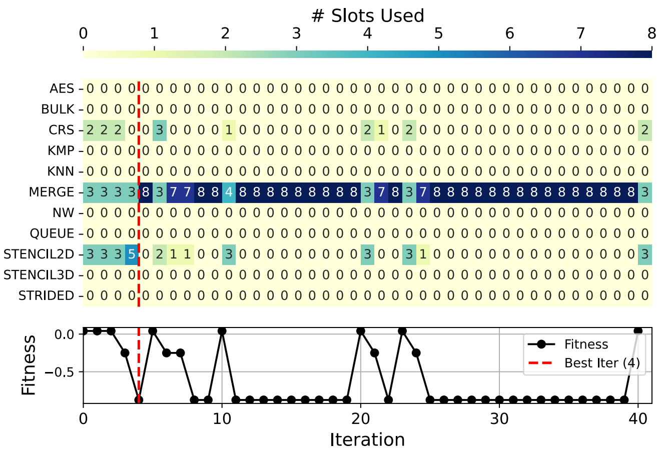
Table I summarizes the parameter values selected after evaluating several alternatives. The number of crows and iterations is kept small to limit the overhead introduced by the scheduler. In addition, to keep the optimization cost manageable, each CSA run only considers the first three tasks in the waiting queue. This choice, also explored in the sensitivity analysis, represents a trade-off between search depth and overhead.

C. Experimental Results

This subsection evaluates the conflict-aware workload optimization methodology and compares it with a baseline. The scheduler is configured with the parameters in Table I and integrated into the workload management infrastructure described in Section III. The synthetic workload defined in the experimental setup is executed on the ZCU102 platform, and the scheduler operates as described in Section IV, using run-time predictions from the ML-based models to guide its decisions. In this experimental campaign the fitness function



(a) $AP = 0.9$



(b) $AP = 0.1$

Fig. 4: Impact of the awareness probability (AP) on the evolution of the CSA search process.

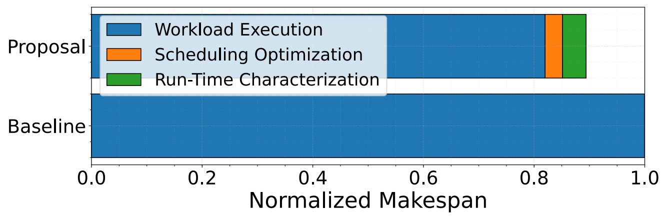


Fig. 5: Normalized makespan comparison between the proposed workload optimization and the baseline approach.

(see Equation 5) is configured with $\alpha = 1$ in order to focus on makespan reduction.

The proposed method is compared with a First come, first served (FCFS) baseline. In this baseline, tasks are scheduled strictly in order of arrival, and the number of replicas for each task is chosen randomly from a uniform distribution between 1 and the maximum number of available slots (8 in this setup).

Figure 5 presents a graphical comparison of the workload makespan obtained with the proposed methodology and with the baseline, normalized to the baseline values. For the proposed method, the reported makespan includes not only the execution time of the workload on the FPGA fabric but also the time spent on run-time workload characterization and on the optimization process itself, which are absent in the baseline.

The results show that the proposed methodology achieves a total workload makespan equal to 89.40% of the baseline, which corresponds to a 1.12× speedup. If a steady-state situation is considered where the workload is already characterized and no further changes occur, i.e., excluding characterization overhead and counting only pure execution plus scheduling optimization, the makespan is reduced to 85.15% of the baseline (1.17× speedup). In practice, workloads in cloud-edge environments will eventually change, so re-characterization will be needed from time to time, but the frequency of these updates depends on the specific deployment.

If the focus is placed only on the execution of the workload, ignoring both characterization and optimization overheads, the makespan drops to 81.99% of the baseline, which corresponds to a 1.22× speedup. These values indicate that task interaction has a significant impact on the total makespan. In this experimental setup, that impact accounts for at least 18.01% of the baseline execution time and is effectively mitigated by the proposed conflict-aware scheduler. Since energy consumption is proportional to execution time and power is not significantly affected by interaction [10], this improvement in performance also translates into better energy efficiency.

Even when characterization and optimization overheads are included, the proposed methodology still reduces workload makespan by 10.60%. At the same time, it provides an up-to-date run-time workload characterization free of extra cost, which can be used not only by the local scheduler but also by higher level orchestrators in the cloud-edge continuum to perform global workload distribution and resource management based on local ML-derived models.

VI. Conclusions

This work has presented a conflict-aware workload optimization methodology for FPGA-based acceleration in dynamic environments such as the cloud-edge continuum. In contrast to traditional design-time strategies based on static profiling and fixed heuristics, the proposed approach introduces a data-driven scheduler that operates at run time. By incrementally training lightweight predictive ML-based models on real execution traces, the system is able to capture and exploit dynamic effects such as task interaction without requiring prior knowledge of the workload.

To explore the large scheduling solution space, the methodology relies on the CSA, a population-based and lightweight metaheuristic suitable for complex optimization problems. The scheduler uses the predictions provided by the run-time workload characterization models to guide its decisions, targeting configurations that reduce interference between tasks and, when required, promote a fair allocation of FPGA resources.

The experimental evaluation shows that the proposed workload optimization methodology reduces workload makespan significantly with respect to a FCFS baseline, even when the overhead introduced by run-time modeling and optimization is included. Under steady-state conditions, where the workload is already characterized, the improvement is even more pronounced, which confirms the benefit of explicitly accounting for conflicts between parallel accelerators. Since execution time reductions directly translate into energy savings and power is not strongly affected by interaction, the approach is particularly attractive for energy-constrained edge deployments running dynamic workloads.

Future work will consider porting and evaluating the methodology on a broader set of platforms and application domains, in order to assess its generality and scalability. Another line of work will explore hierarchical workload management, where information derived from local run-time characterizations is propagated to higher-level managers to enable continuum-wide optimization. In that context, techniques such as federated learning could be used to aggregate local models while preserving autonomy and privacy and limiting communication overhead.

Acknowledgment

Funded by the European Union, by grant No. 101135183. Views and opinions expressed are however those of the author(s) only and do not necessarily reflect those of the European Union. Neither the European Union nor the granting authority can be held responsible for them.

References

[1] A. T. Atieh, "The next generation cloud technologies: A review on distributed cloud, fog and edge computing and their opportunities and challenges," *RRST*, vol. 1, no. 1, pp. 1–15, 2021.

[2] D. Khalyeyev, T. Bureš, and P. Hnětynka, "Towards characterization of edge-cloud continuum," in *Software Architecture. ECSA 2022 Tracks and Workshops*. Springer, 2023, pp. 215–230.

[3] C. Xu *et al.*, "The case for fpga-based edge computing," *IEEE Trans. Mobile Computing*, vol. 21, no. 7, 2022.

[4] M. Leeser, S. Handagala, and M. Zink, "Fpgas in the cloud," *Computing in Science & Engineering*, vol. 23, no. 6, pp. 72–76, 2021.

[5] C. Bobda, J. M. Mbongue, P. Chow, M. Ewais, N. Tarafdar, J. C. Vega, K. Eguro, D. Koch, S. Handagala, M. Leeser, M. Herbordt, H. Shahzad, P. Hofste, B. Ringlein, J. Szefer, A. Sanaullah, and R. Tessier, "The future of fpga acceleration in datacenters and the cloud," *ACM Trans. Reconfigurable Technol. Syst.*, vol. 15, no. 3, 2022.

[6] O. Diessel and H. Elgindy, "On dynamic task scheduling for fpga-based systems," *Int. J. Foundations of Computer Science*, vol. 12, no. 5, pp. 645–669, 2001.

[7] J. Blazewicz, J. K. Lenstra, and A. H. G. Rinnooy Kan, "Scheduling subject to resource constraints: Classification and complexity," *Discrete Applied Mathematics*, vol. 5, no. 1, pp. 11–24, 1983.

[8] A. Adhi, B. Santosa, and N. Siswanto, "A meta-heuristic method for solving scheduling problem: Crow search algorithm," *IOP Conf. Ser.: Materials Science and Engineering*, vol. 337, no. 1, p. 012003, 2018.

[9] L. Tianyang, Z. Fan, G. Wei, S. Mingqian, and C. Li, "A survey: Fpga-based dynamic scheduling of hardware tasks," *Chinese Journal of Electronics*, vol. 30, no. 6, pp. 991–1007, 2021.

[10] J. Encinas, A. Rodríguez, A. Otero, and E. de la Torre, "Data-driven modeling of reconfigurable multi-accelerator systems under dynamic workloads," *Microprocessors and Microsystems*, vol. 107, p. 105050, 2024.

[11] J. Encinas, A. Rodríguez, and A. Otero, "Leveraging incremental machine learning for reconfigurable systems modeling under dynamic workloads," *ACM Trans. Reconfigurable Technol. Syst.*, vol. 18, no. 1, 2025.

[12] A. Askarzadeh, "A novel metaheuristic method for solving constrained engineering optimization problems: Crow search algorithm," *Computers & Structures*, vol. 169, pp. 1–12, 2016.

[13] X.-S. Yang, *Engineering Optimization: An Introduction with Metaheuristic Applications*. Wiley, 2010.

[14] E. H. Houssein, A. G. Gad, Y. M. Wazery, and P. N. Suganthan, "Task scheduling in cloud computing based on meta-heuristics: Review, taxonomy, open challenges, and future trends," *Swarm and Evolutionary Computation*, vol. 62, p. 100841, 2021.

[15] R. Ramezani, "A prefetch-aware scheduling for fpga-based multi-task graph systems," *The Journal of Supercomputing*, vol. 76, no. 9, pp. 7140–7160, 2020.

[16] M. Bertolino, R. Pacalet, L. Apvrille, and A. Enrici, "Efficient scheduling of fpgas for cloud data center infrastructures," in *Proc. Euromicro Conf. on Digital System Design (DSD)*, 2020, pp. 57–64.

[17] S. Alismail and D. Koch, "Efficient resource scheduling for run-time reconfigurable systems on fpgas," in *Proc. Int. Conf. on Field-Programmable Logic and Applications (FPL)*, 2023, pp. 123–129.

[18] H. Khetawat and F. Mueller, "Workload scheduling on heterogeneous devices," in *Proc. ISC High Performance*, 2024, pp. 1–11.

[19] A. Vaishnav, K. D. Pham, and D. Koch, "Heterogeneous resource-elastic scheduling for cpu+fpga architectures," in *Proc. Int. Symp. on Highly-Efficient Accelerators and Reconfigurable Technologies (HEART)*, 2019.

[20] A. Duhamel and S. Pillement, "Qos aware design-time/run-time manager for fpga-based embedded systems," in *Design and Architecture for Signal and Image Processing*. Springer, 2022, pp. 96–107.

[21] A. Vaishnav, K. D. Pham, J. Powell, and D. Koch, "Fos: A modular fpga operating system for dynamic workloads," *ACM Trans. Reconfigurable Technol. Syst.*, vol. 13, no. 4, 2020.

[22] R. Paul and M. Danelutto, "Power aware scheduling of tasks on fpgas in data centers," in *Proc. Euromicro Int. Conf. on Parallel, Distributed and Network-Based Processing (PDP)*, 2024, pp. 148–152.

[23] A. Mehrabi, D. J. Sorin, and B. C. Lee, "Spatiotemporal strategies for long-term fpga resource management," in *Proc. IEEE Int. Symp. on Performance Analysis of Systems and Software (ISPASS)*, 2022, pp. 198–209.

[24] E. Karabulut, A. A. Malik, A. Awad, and A. Aysu, "Themis: Time, heterogeneity, and energy minded scheduling for fair multi-tenant use in fpgas," *IEEE Trans. Computers*, pp. 1–14, 2025.

[25] Z. Li and S. Hauck, "Configuration prefetching techniques for partial reconfigurable coprocessor with relocation and defragmentation," in *Proc. ACM/SIGDA Int. Symp. on Field-Programmable Gate Arrays (FPGA)*, 2002, pp. 187–195.

[26] A. Rodríguez, J. Valverde, J. Portilla, A. Otero, T. Riesgo, and E. De la Torre, "Fpga-based high-performance embedded systems for adaptive edge computing in cyber-physical systems: The artico3 framework," *Sensors*, vol. 18, no. 6, p. 1877, 2018.

[27] Xilinx, "System Integrated Logic Analyzer Product Guide," PG-261, 2021.

[28] R. Aron and A. Abraham, "Resource scheduling methods for cloud computing environment: The role of meta-heuristics and artificial intelligence," *Engineering Applications of Artificial Intelligence*, vol. 116, p. 105345, 2022.

[29] J. Kennedy and R. Eberhart, "Particle swarm optimization," in *Proc. Int. Conf. on Neural Networks (ICNN)*, vol. 4, 1995, pp. 1942–1948.

[30] M. Dorigo, M. Birattari, and T. Stutzle, "Ant colony optimization," *IEEE Computational Intelligence Magazine*, vol. 1, no. 4, pp. 28–39, 2006.

[31] H. Singh, S. Tyagi, P. Kumar, S. S. Gill, and R. Buyya, "Metaheuristics for scheduling of heterogeneous tasks in cloud computing environments: Analysis, performance evaluation, and future directions," *Simulation Modelling Practice and Theory*, vol. 111, p. 102353, 2021.

[32] B. Reagen, R. Adolf, Y. S. Shao, G.-Y. Wei, and D. Brooks, "Machsuite: Benchmarks for accelerator design and customized architectures," in *Proc. IEEE Int. Symp. on Workload Characterization (IISWC)*, 2014, pp. 110–119.

[33] K. Asanovic, R. Bodik, B. Catanzaro, J. Gebis, P. Husbands, K. Keutzer, D. Patterson, W. Plishker, J. Shalf, and S. W. Williams, "The landscape of parallel computing research: A view from berkeley," 2006.