On Exploiting Patterns For Robust FPGA-based Multi-accelerator Edge Computing Systems

Seyyed Ahmad Razavi, Hsin-Yu Ting, Tootiya Giyahchi, Eli Bozorgzadeh
University of California, Irvine
{srazavim, hting1, tgiyahch}@uci.edu, eli@ics.uci.edu

***Abstract*—Edge computing plays a key role in providing services for emerging compute-intensive applications while bringing computation close to end devices. FPGAs have been deployed to provide custom acceleration services due to their reconfigurability and support for multi-tenancy in sharing the computing resource. This paper explores an FPGA-based Multi-Accelerator Edge Computing System, that serves various DNN applications from multiple end devices simultaneously. To dynamically maximize the responsiveness to end devices, we propose a system framework that exploits the characteristic of applications in patterns and employs a staggering module coupled with a mixed offline/online multi-queue scheduling method to alleviate resource contention, and uncertain delay caused by network delay variation. Our evaluation shows the framework can significantly improve responsiveness and robustness in serving multiple end devices.**

I. Introduction

Edge computing systems have become an unavoidable scheme to enable almost real-time processing of neural network applications in the era of IoTs [1]–[6]. In various CPS and IoT applications, end devices such as small drones or mini-robots are not mostly equipped with computational resources to process compute-intensive applications. Hence, they seek access to computational resources at the edge node for immediate processing. Edges are mostly equipped with accelerators such as FPGAs and GPUs to provide custom computations [7]. An FPGA edge can provide reconfigurable custom hardware acceleration for neural network (e.g., binarized DNNs like BCNN [8], or FINN [9]) and host multiple DNN accelerators accessed by applications concurrently [1], [2].

At the FPGA edge node, serving a queue of requests from various end devices for DNN acceleration, the FPGA device can execute a set of DNN accelerators from a library of DNN accelerators, each optimized for a certain objective, such as accuracy and performance. The system software at the edge can process the requests on a first-come, first-serve (FCFS) basis. However, this can incur dynamic reconfiguration overhead and degrade the system responsiveness. Researchers have proposed batching the requests during runtime to reduce the overhead [3], but this can lead to starvation and timed-out requests. In many edge computing platforms, there is only an online scheduler without any prior knowledge on any periodic nature or some regularity in arrival time of the tasks such as [1], [4], [5]. However, in various CPS and IoT monitoring applications, sensing occurs at a fixed rate, and hence, the data is sent to the edge for acceleration periodically. When multiple end devices continuously send acceleration requests at a fixed rate, a pattern in the sequence of requests can be observed. [6] adopts the ideal

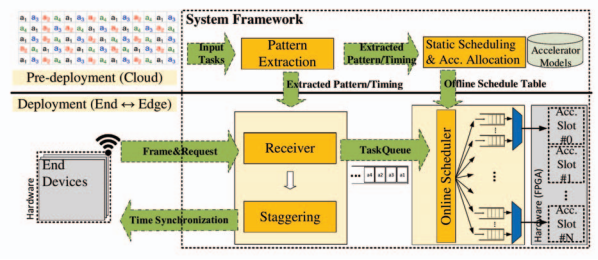

Fig. 1: System Overview

fixed pattern of arrival times of tasks under a set of periodic tasks for scheduling. However, this pattern is not unique and might change over time due to noises and uncertainties in the environment.

In this paper, we present a systematic approach to extract the pattern in a sequence of requests to access various accelerators at the pre-deployment phase and exploit it for pre-deployment planning on accelerator allocation and tasks' execution ordering. In addition, at the deployment phase, we propose a runtime end-device data-transfer staggering method, and an online scheduler to accommodate the network delay variation and enhance the regularity of patterns in the queue of requests, and also mitigate the effect of task arrival time noise. The experimental results show that using our proposed framework, the average response/wait time is improved by up to 36%/60% compared to the framework where a fixed pattern is assumed for static task scheduling and accelerator allocation [6].

II. System Framework

Figure 1 shows our proposed framework. At the pre-deployment phase, pattern extraction is presented to observe the behavior of the tasks' arrival at the edge. Then, we propose a static task scheduling and accelerator allocation, based on the extracted pattern, which provides the tentative ordering of the task and selected DNN accelerators to be implemented on the FPGA. Strictly following the Offline Schedule Table might lead to system performance degradation due to random parameters such as network jitter. Therefore, for the deployment phase, we propose a soft real-time scheduler that uses the Offline Schedule Table as a guideline, but properly adapts the schedule based on the task arrival times and task queue status. Also, tightly coupled with our online scheduler, our proposed runtime arrival time staggering module seeks to make the task arrival times close to the extracted pattern by staggering the arrival times using a feedback loop.

978-3-9819263-6-1/DATE22/©2022 EDAA

III. PRE-DEPLOYMENT PHASE

To extract system characteristics and exploit it to improve the system performance, we propose a pre-deployment phase including two steps 1-pattern extraction, and 2- static scheduling. The output of this phase is expected arrival times and a static schedule that will be used in the deployment phase. The static schedule is an optimized schedule based on the calculated timings from the pattern extraction step.

A. Pattern Extraction

To exploit any regularity in receiving acceleration requests at the edge, we use a pattern extraction module to understand the characteristics of our network's behavior. Due to noises and uncertainties in the environment, such a pattern is challenging to extract. Figure 2 shows an overview of our setup and where pattern extraction plays its role. a_1 to a_n are different applications represented as multiple end devices that simultaneously send tasks to an edge node for acceleration. Each end device periodically sends tasks (t_i) to the edge node continuously, and the edge node receives a queue of incoming tasks (Q). We propose an episode-mining-based approach to find a representative pattern of how tasks arrive at the edge.

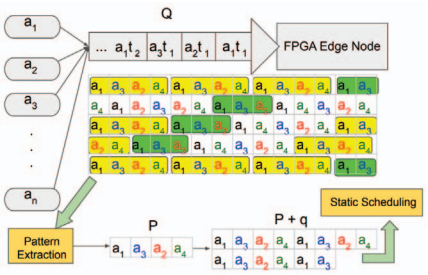

Fig. 2: The pattern extraction module extract the dominant pattern P and cluster remaining tasks in the hyperperiod as q and output $P+q$.

Algorithm 1 explains our method to find the dominant pattern over Q. We use the episode mining algorithm, EMMA, proposed in [10], inside our pattern extraction module. *EMMA* receives a sequence of events and returns all patterns within a window size of $maxwin$ that their occurrence in the input sequence is greater than $minsup$. We initially set $maxwin$ to be the size of hyperperiod based on the expected tasks' rates. To remove partially ordered patterns, if the maximum length of patterns is smaller than $maxwin$, $maxwin$ would be decreased to the maximum pattern size step by step until the condition is met (Line 3-5 in Algorithm 1). Then, *maxsup_patterns* finds and returns a dominant pattern P with the maximum support within the $maxwin$-size patterns (Line 6 in Algorithm 1). Since all tasks within the hyperperiod should be taken into account for the static scheduling and accelerator allocation, we cluster the remaining irregular tasks within the hyperperiod in timestamp (q, line 7 in Algorithm 1) and attach them to the dominant pattern to create the representative pattern $P+q$ (Line 8 in Algorithm 1).

Figure 2 includes an example of the incoming task IDs sequence of 4 devices with sampling rates of 2,3,4 and 5 fps

Algorithm 1 Pattern Extraction

1: $maxwin$ = $hyperperiod$
2: $patterns$ = EMMA($Q, maxwin, minsup$)
3: **while** maxlen($patterns$) $< maxwin$ **do**
4: $maxwin$ = maxlen($patterns$)
5: $patterns$ = EMMA($Q, maxwin, minsup$)
6: P = maxsup_patterns(maxlen_patterns($patterns$))
7: q = calc_irregular_tasks(Q, P)
8: Return $P+q$

below the Q, every row is an interval of size 14 (=hyperperiod). $P = a_1, a_3, a_2, a_4$ is the output of line 6 in Algorithm 1 and exists in all intervals, however, except for the first and last intervals, all intervals in between, have some partial deviations from P for the remaining tasks. Pattern P's size is within the hyperperiod and represents an ordered subsequence that occurs frequently in our input sequence. The remaining out-of-order application IDs are tasks that arrive closely to each other and due to network congestion, slightly deviate from P. We average the relative arrival times over all of them to find q and create the representative of our input sequence as $P+q$.

B. Static Scheduling and Accelerator Allocation

Each task in the representative pattern is a request for inference acceleration under certain requirements, such as applications (i.e. dataset), accuracy, and performance. Given these requirements, tasks might be compatible with various types of accelerators, and mapping them to these accelerators depends on the schedule and available resources.

An FPGA can be partitioned into multiple reconfiguration regions, called accelerator slots, that are able to execute accelerators simultaneously and independently. In each slot, there is a time overhead for reloading (and potentially partial reconfiguration) if the accelerator needs to be changed during runtime. Therefore, for each task, the system must decide when (timestamp) to start the task on which slot using what accelerator. As a target application, we applied the accelerator models in [6] that adopts the Binarized Neural Network accelerator, e.g. [9], with different factors in DNN models and hardware optimization so they can be accommodated properly given the size of the slots and application's requirements.

We formulate the above temporal and spatial scheduling and accelerator allocation problem for tasks in the extracted pattern as a mixed-integer linear programming (MILP) problem, and optimize it to maximize the system responsiveness, i.e. minimize the average response time among all tasks in the extracted pattern. Response time includes time to wait for available slots, time for execution on FPGA, and time for reconfigure/reload accelerators if necessary. Due to the lack of space, we only summarize the adapted key constraints, and omit basic scheduling equations [6]: **Cross-Pattern Constraint:** since the schedule repeats for each hyperperiod, the potential reconfiguration/reload overhead between these repeated hyperperiod's schedules has to be considered. **Load-Balance Constraint:** the schedule will balance the load among slots; thus, preventing growing tasks in any single queue and risks of timeout.

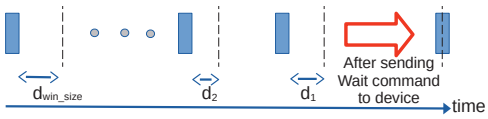

Fig. 3: time staggering for an end device: Rectangles are arrived tasks at the edge, dotted lines are expected arrival times, and d_i is their gap.

IV. Deployment Phase

The Offline Schedule Table determines the schedule of accelerators on each slot of the FPGA based on the expected arrival times of tasks. However, because of various factors such as network delay variation, the task arrival times will drift from the expected arrival times after a while which will negatively affect the response time. To handle these uncertainties during runtime, we propose two lightweight online modules: 1- Staggering module, and 2- Online scheduler.

A. Task Arrival Time Staggering Module

In this section, we propose a lightweight method to adjust the end device sending time, so the edge receives tasks close to expected arrival times. This method just requires the task arrival times. The proposed module is a feedback loop that for each device, during runtime, monitors the task arrival times, and guides the end device to adjust its sending time, so the arrival times of the incoming future tasks get close to the expected arrival times. For each end device, as it is shown in Figure 3, the sliding window average (m) of the difference (d) between the task arrival time and the expected arrival times with a window size of win_size is calculated on the edge. Then, the edge will send a 'Wait command' to the end-node to adjust the sending time by the calculated average (m), if it is more than a certain threshold. In many periodic applications, such as monitoring, the end nodes can simply adjust their sampling time rather than holding sampled data on the end device. Calculating the average is incremental and lightweight with time complexity of $O(1)$ for a streaming data. Note that the 'Wait command' only contains an integer number, which fits in one network packet. This method does not need a synchronized internal clock, nor explicitly measuring the network delay. Hence, it reduces the development time, eases the system deployment, and can detect/adapt to the environmental changes without requiring complex estimation methods.

B. Online Scheduler

Even with staggering, task arrival time varies from one task to another due to random parameters such as network jitter. On the other side, we have obtained extracted patterns and timing of tasks in the pre-deployment phase. To exploit this knowledge, and also reduce the effect of tasks' arrival time fluctuation on the response time, we propose a lightweight Online scheduler, which approximately follows the Offline Schedule Table while accommodating for the uncertainties. Our proposed online scheduler uses Offline Schedule Table as a guideline and reacts to the arbitrary arrival of tasks to increase the system's responsiveness. The input to the algorithm is a queue $taskQueue$ which holds all arrived tasks in the order they arrive. It assigns the tasks in a FCFS order to the corresponding accelerator's queue ($AcTskQs$) based on the Offline Schedule Table. Note that each accelerator has its own task queue. Online scheduler has two main rules to mitigate the effect of tasks' arrival time fluctuation: **Rule 1**: it processes all the tasks in the current accelerator's queue, even if the time for the currently loaded accelerator passes the tentative time in Offline Schedule Table. To prevent starvation when 'borrowing' time from the next accelerator in line, if there is any task in the next accelerator's task queue that is being timed out due to overrunning, the scheduler will fetch the next accelerator. **Rule 2**: if the currently loaded accelerator finishes the tasks earlier than it was expected, the next accelerator will be fetched earlier than the tentative time in Offline Schedule Table. Therefore, the slot can start to process the tasks in the next queue earlier.

Using these two rules, Online scheduler uses Offline Schedule Table as a guideline rather than strictly following it, and accommodates for the task arrival time fluctuations. Each slot has its own online scheduler, running on different threads. In the main process, we assign arrived tasks to a proper accelerator's queue($AcTskQs$) based on the $AcSchedule$. Online scheduler is shown in Algorithm 2. We call the accelerator that is currently loaded on the slot as current_accelerator. For each task in the current_accelerator queue $AcTskQs[AcIndx]$, in a FCFS order, the scheduler runs the task on the FPGA's slot if it not timed out. The accelerator call is non-preemptive. The scheduler keeps running the tasks in $AcTskQs[AcIndx]$ on the current_accelerator till there is no task left in the queue, or an overrun happens. Overrun happens when current_accelerator is 'borrowing' time from the next accelerator and there is a task being timed out in the next accelerator task queue. In the case of overrun, scheduler will fetch the next accelerator. After processing the tasks in the $AcTskQs[AcIndx]$ (line $3-8$), the online scheduler checks if it has to fetch the next accelerator using the following two conditions (line 9): 1- pre_fetch_flag indicates if the last served task is the final task in the tentative schedule $AcSchedule$. Using this flag, if the final task execution finishes early, the slot will not remain idle, and it will fetch the next accelerator. 2- $accel\_timed\_out\_flag$ function indicates if the current accelerator's time is over. Scheduler fetches the next scheduler in a round-robin basis based on Offline Schedule Table, which prevents starvation.

Algorithm 2 Online Scheduler

```
1:  assign_task(taskQueue, AcSchedule, AcTskQs)
2:  while True do //multithread: one thread per slot
3:      while not empty(AcTskQs[AcIndx]) do
4:          if accel_timed_out_flag and over_run_flag then
5:              break;
6:          Task=AcTskQs[AcIndx].pop()
7:          if not_timed_out(Task,Thresh) then
8:              run(Task);
9:      if pre_fetch_flag or accel_timed_out_flag then
10:         AcIndx=fetch_next_accel(AcSchedule,'roundrobin')
```

V. Evaluation

Platform Setup: We used a Xilinx Ultrascale+ MPSoC ZCU104 board as the FPGA Edge, partitioned into three

accelerator slots, and multiple Raspberry Pi 3B+/4B as the End devices. The End devices (Edge) are connected to a wireless access point through 5GHz WiFi (Ethernet), and communicate using TCP/IP socket in an office environment. For each experiment, end devices, starting at a random time, periodically send requests for acceleration to the FPGA edge for 10 minutes. We set the timeout threshold for tasks to 1 second. The accelerator models we applied are from [6], which perform DNN inference for vision applications, such as object detection and classifications. We measured and reported the time interval between the task arrival at the edge and the beginning (end) of execution on the FPGA edge as the wait (response) time.

Evaluation Scenario: We explore seven end devices periodically sending requests for acceleration to the Edge. There are three different workloads representing 34/30/26 fps in total. We also explore the variants of fps per end device. For example, in Table I, w34d2, w34d1 and w34d0 gave the total FPS of 34, and for the former one, the FPS for individual end devices are $\{2, 2, 5, 5, 6, 6, 8\}$. Since the WiFi noise level in office environment is limited, during runtime, we add various levels of extra delays with normal distribution [11] ($X \sim \mathcal{N}(\mu, \sigma)$ where $0 < \mu < 70$, and $0 < \sigma < 10$) to the end devices to evaluate the performance of the proposed framework in tackling the changes in network delay and end device timing. We compared our proposed method with two other scheduling methods, $FCFS$ and $Static$, as baseline. In $FCFS$, accelerators and slots are shared among tasks and applications, and processed in a $FCFS$ order. The other baseline method, $Static$, is basically the static scheduling in Section III-B without the proposed deployment phase; therefore, it strictly follows the Offline Schedule Table based on the pre-defined extracted pattern/timing.

Result and Discussion: In Table I, we compare our proposed methods with the baseline methods in terms of response/wait time and the number of timeout tasks. For the $FCFS$, there are large amounts of timeout tasks and high response time. $FCFS$ shares slots to all applications; however, processing tasks in a $FCFS$ order lacks the consideration of overhead in re-allocating accelerators, which results in recurring reconfiguration/reload. $Static$ method uses the schedule and accelerator allocation which considers sharing accelerators, and slots, together with the extracted patterns/timing. Unlike $FCFS$, based on the static schedule, tasks may run on an out-of-order basis to avoid changing accelerators intermittently and achieve the minimum response time. Thus, $Static$ significantly improves both response time and the number of timeout tasks compared to the $FCFS$. Strictly following the Offline Schedule Table makes the system vulnerable to the task arrival time fluctuations. As it is shown in Table I, some tasks have timed out in $Static$ method. In our proposed method, using the deployment phase, the system can tolerate the task arrival time variation. We set the Staggering module in the Edge device to send the 'Wait command', if necessary, with the time interval of at least 2 seconds; thus, the communication overhead in the deployment phase is very low. In addition, the computation overhead of Online scheduler and Staggering module is negligible (the relative CPU utilization on the Edge was under 5% during our experiments). Since the proposed method utilizes the system knowledge and handles the random arrival times by using Online scheduler, the response/wait time has improved. As it is shown in Table I, the response time has reduced from 7% up to 36% compared to $Static$. As the response time includes the execution time, we measured the average wait time to better show the effect of our proposed method. The Online phase has improved the wait time from 12% up to 60%. Note that our proposed method assigns the tasks to the accelerator based on Offline Schedule Table, same as what $Static$ uses. Therefore, execution times are the same for these two methods.

TABLE I: Comparison (Tout/Rsp/Wait denotes the number of timeout tasks/the average response time/the average wait time. Imp is the improvement compared to $Static$.)

		$FCFS$		$Static$ [6]			Proposed method		
WL.	FPS	Tout	Rsp	Tout	Rsp	Wait	Tout	Rsp/Imp	Wait/Imp
w34d2	2,2,5,5,6,6,8	8343	872	0	245	170	0	209/15%	134/21%
w34d1	3,4,5,5,5,6,6	7789	881	4	198	121	0	184/7%	107/12%
w34d0	4,5,5,5,5,5,5	8074	873	57	335	280	1	247/26%	172/39%
w30d2	2,2,3,4,5,7,7	5836	876	4	198	121	0	126/36%	51/58%
w30d1	2,4,4,4,5,5,6	5915	880	0	149	74	2	130/13%	55/26%
w30d0	4,4,4,4,4,5,5	6464	876	0	188	114	0	161/14%	87/24%
w26d2	2,2,3,3,3,5,8	4348	859	0	135	62	0	110/19%	25/60%
w26d1	2,3,3,4,4,5,5	4474	864	15	180	108	0	129/28%	52/52%
w26d0	3,3,4,4,4,4,4	4577	866	0	132	55	0	123/7%	48/13%

$\star$ There are $10mins \times 60 \times FPS$ tasks in total in each experiment.

VI. Conclusion

In this paper, we present a system framework that extracts patterns of tasks at the edge for static scheduling and accelerator allocation, and, during runtime, employs a soft real-time scheduler together with a staggering module to improve the responsiveness on a multi-tenant FPGA-based DNN system.

References

[1] S. Jiang et al., "Scylla: Qoe-aware continuous mobile vision with fpga-based dynamic deep neural network reconfiguration," in IEEE INFOCOM, 2020.

[2] S.-C. Kao et al., "Domain-specific genetic algorithm for multi-tenant dnnaccelerator scheduling," arXiv preprint arXiv:2104.13997, 2021.

[3] Z. Yang et al., "Deeprt: A soft real time scheduler for computer vision applications on the edge," arXiv preprint arXiv:2105.01803, 2021.

[4] J. Meng et al., "Dedas: Online task dispatching and scheduling with bandwidth constraint in edge computing," in IEEE INFOCOM, 2019.

[5] X. Liu et al., "Collaborative edge computing with fpga-based cnn accelerators for energy-efficient and time-aware face tracking system," IEEE Transactions on Computational Social Systems, 2021.

[6] H.-Y. Ting et al., "Dynamic sharing in multi-accelerators of neural networks on an fpga edge device," in IEEE ASAP, 2020.

[7] Z. Zhu et al., "A hardware and software task-scheduling framework based on cpu+ fpga heterogeneous architecture in edge computing," IEEE Access, 2019.

[8] H. Peng et al., "Binary complex neural network acceleration on fpga," in IEEE ASAP, 2021.

[9] M. Blott et al., "Finn-r: An end-to-end deep-learning framework for fast exploration of quantized neural networks," ACM Transactions on Reconfigurable Technology and Systems, 2018.

[10] K.-Y. Huang and C.-H. Chang, "Efficient mining of frequent episodes from complex sequences," Information Systems, 2008.

[11] Y. Zeng et al., "Emuedge: A hybrid emulator for reproducible and realistic edge computing experiments," in IEEE International Conference on Fog Computing (ICFC), 2019.