A Dynamic Distributed Scheduler for Computing on the Edge

Fei Hu *
Dept. of Computer Science
University of Colorado
fei.hu@colorado.edu

Kunal Mehta
Dept. of Computer Science
University of Colorado
kunal.mehta@colorado.edu

Shivakant Mishra
Dept. of Computer Science
University of Colorado
mishras@colorado.edu

Mohammad AlMutawa
Dept. of Computer Science
Kuwait University
almutawa@cs.ku.edu.kw

***Abstract*—Edge computing is crucial for IoT applications, especially those needing quick, private data handling. However, these applications are resource-intensive, and edge computing resources are limited compared to cloud capabilities. Efficiently using these limited resources is essential to meet all application demands, including latency, privacy, and cost. Due to the dynamic and hybrid nature of IoT environments, static scheduling systems often falls short in meeting these diverse constraints. This paper describes the design, implementation, and evaluation of a dynamic distributed scheduler for the edge. This scheduler schedules the execution of various computing tasks across all computing resources available at the edge in order to satisfy the diverse constraints of an IoT application. The key characteristic of this scheduler is that it constantly monitors the current state of the IoT infrastructure and dynamically adjusts the scheduling of various computing tasks based on the current environmental context as well as the current computing and network conditions. The scheduler utilizes a predictive profiling method to manage the hybrid and variable nature of edge computing resources and tasks. A prototype of this scheduler has been implemented and the paper demonstrates its practicality via an augmented reality application.**

***Index Terms*—Dynamic Distributed Scheduler, Parallel Computing, Edge Computing, Augmented Reality Applications, Device Profile**

I. Introduction

Edge AI systems [1], [2] have emerged as a promising solution for handling the growing demand for AI workloads by bringing computation closer to the data sources and reducing latency [3], [4]. A key challenge in building AI applications at the edge is that these applications tend to be highly compute-intensive and computing capabilities are limited at the edge server when compared to the cloud. Developers have been addressing this issue in an ad hoc manner by distributing different computing tasks of applications across different computing nodes available at the edge, including different low-end sensing devices as well as different processing units at the edge server. However, distributing computing tasks across different computing nodes comes with the added cost of having to transfer data between nodes. Thus, a careful balance between the performance improvement due to parallel computation across different computing nodes and the increased delays due to data movement is needed to fully realize the performance benefits of task distribution. This is further complicated by three important factors prevalent in an IoT environment: (1) Network conditions fluctuate, dynamically altering the communication delays associated with data movement; (2) CPU load on computing nodes fluctuate due to dynamically varying contexts, resulting in variable compute times for various tasks; and (3) Compute nodes at the edge are highly diverse with widely varying compute, storage, and communication capabilities [3], [5], [6].

This paper addresses these challenges by building a dynamic distributed scheduler that schedules different computing tasks of an application across different computing nodes based on the current computing and networking conditions to meet all the requirements of the application by parallel computing, including latency, privacy, power, and cost constraints. This system constantly monitors the current compute and network conditions and dynamically adjusts the scheduling of the compute tasks accordingly. It has five important features:

1) Device profiling: To address heterogeneity among different computing devices, each device in the IoT environment is profiled for different computing tasks under different computing loads to enable the scheduler to make optimal scheduling decisions at runtime.
2) Predictive profiling: To address the limitation of pre-profiling every possible device under every possible computing environment, the scheduler incorporates a prediction mechanism to predict the running times of various computing tasks during runtime and adjusts schedules accordingly.
3) Two-level scheduling decision: To minimize the overhead of the scheduler, the scheduling decision is done at two levels, a primitive scheduling decision at the low-end devices and a near-optimal scheduling decision at the edge server.
4) Local compute principle: To address the uncertainty in determining accurate communication and compute latency under every possible computing environment, the scheduler schedules tasks locally as a low-level scheduling decision as long as this scheduling satisfies all application requirements.
5) Predictive compute drops: Based on application guidance, the scheduler selectively drops some compute task instances to maximize meeting the overall application constraints.

A prototype of this dynamic, distributed scheduler has

This research is supported by grants from NSF.

979-8-3503-4965-8/24/$31.00 ©2024 IEEE

been implemented and extensively evaluated under different computing scenarios. Evaluation results demonstrate that this scheduler outperforms current ad hoc scheduling tasks at the edge to satisfy various application requirements. We build an augmented reality application at the edge and demonstrate the usefulness of our scheduler compared to the current state-of-the-art. We also provide a predictive profiling solution where other applications running times are predicted based on existing application profiling and propose further optimizations. In particular, the paper provides the following important contributions:

- The proposed dynamic distributed scheduler addresses the key challenges of device heterogeneity and dynamically changing compute and network environment encountered in an IoT environment at the edge by dynamically adjusting the distributing of tasks based on the current operating conditions.
- The proposed predictive profiling solution simplifies the onerous task of pre-profiling every device under every possible compute scenario.
- The paper provides a prototype implementation of the proposed scheduler and an extensive performance evaluation under different operating conditions. The experiment results indicate that this distributed architecture is efficient at the edge with parallel computing.

II. Related Work

To overcome the challenges in edge environment [7], [8], in recent years, many state-of-the-art scheduling techniques have emerged and those efforts primarily fall into two categories: centralized scheduling and distributed scheduling [9]. Generally, centralized methods mainly include convex optimization [10], approximate algorithm [11], heuristic algorithm [12], [13], and machine learning [14], [15]; distributed methods mainly include game theory [16], matching theory [17], auction [18], federated learning [19], and blockchain [20].

The referenced studies highlight performance improvements in areas like latency, energy use, cost, utility, profit, and resource efficiency. For example, paper [10] leverages Lyapunov optimization to enhance problem-solving efficiency, with a focus on minimizing response times. Meanwhile, paper [13] addresses the distribution challenge as NP-hard, proposing a greedy heuristic algorithm aimed at satisfying latency requirements. Additionally, paper [16] targets energy efficiency with a distributed algorithm derived from Game Theory.

While these studies present significant benefits in performance metrics, their real-world application faces notable challenges. Techniques like Lyapunov optimization and decomposition in complex algorithms may overly complicate implementation in actual edge networks [10]. Similarly, approaches based on approximation algorithms, game theory, and auctions risk converging to local optima [11], [18]. Heuristic and machine learning algorithms often require extensive parameterization, complicating their adaptability [12], [14]. Matching theory, although considered, may not effectively address partial offloading issues, which are central to this paper [17]. Moreover, emerging methods like federated learning and blockchain, while promising, are still nascent in this domain [19], [20].

We propose a distributed architecture with a dynamic distributed scheduler that utilizes device profiling and workload monitoring to address the mentioned challenges. Employing a two-tier scheduling approach, the lower level emphasizes local computing, with comprehensive performance enhancements elucidated in this paper. Additionally, this framework sets the stage for upper-level scheduling, where we can delve into performance enhancement based on current research findings.

III. Scheduler Design

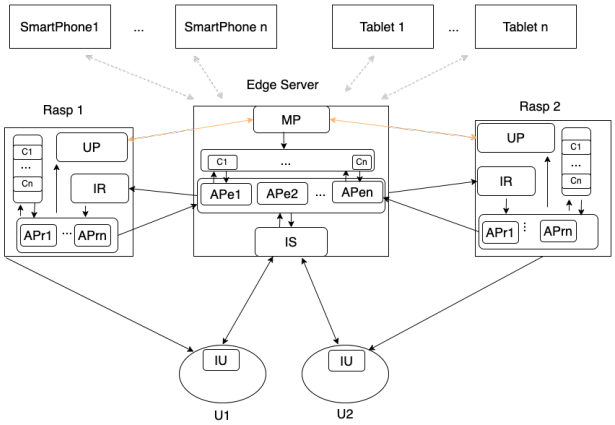


Fig. 1: Architecture

An IoT application is typically a long-running application composed of multiple repetitive compute tasks. For example, an augmented reality application is comprised of a sequence of three different compute tasks (extracting frames from a video stream, detecting objects in a frame and putting labels, and combining the labelled frames to form a video stream) that are repetitively executed on a sequence of frames. The goal of our scheduler is to schedule the execution of these tasks on different devices so as to ensure that all application constraints such as real time latency requirements or privacy are satisfied.

Our scheduler is designed for a typical edge-computing based IoT environment that is comprised of three primary components as illustrated in Figure 1: an edge server, several IoT sensors controlled and accessed via end devices, and (mobile) users. The edge server is the administrative authority of the IoT environment and interacts with the users as well as the end devices. It is relatively resource-rich, authenticates users, supports one or more applications that (authorized) users may invoke, and interacts with the end devices to operate IoT sensors. A user interacts with the edge server to invoke an application and receives the application results from the edge server as well as from one or more end devices. Finally, end devices control the operation of IoT sensors based on the directives given by the edge server, including invoking sensors and receiving sensor data. While the end devices are significantly resource-constrained compared to the edge server,

depending on their capabilities, they may execute scheduler and application tasks based on the directives given by the edge server. Typical end devices are Raspberry Pi's, Arduino boards, smartphones and tablets.

A. Scheduler Design Principles

Edge IoT environments are highly dynamic and hybrid, featuring a range of devices from resource-limited to powerful edge servers. Variability in communication and compute loads, influenced by real-time events, necessitates a scheduler to dynamically adapt to these fluctuations during runtime. However, this need for dynamic adaptation requires the scheduler to constantly monitor the current computing and networking conditions, which adds to the overall performance overhead of an application. Thus, balancing this extra performance overhead with the performance gains that result from near optimal scheduling made possible by having an accurate, up-to-date compute and network state information is critical for optimizing task scheduling and enhancing application the overall application performance.

Our distributed scheduler incorporates *local compute principle* at the end devices to address this challenge: execute a task at the end device that has the data as long as this execution can meet all the application constraints such as real time latency requirements. This entails executing a task at the end device that controls the IoT sensor whose data is needed for that task. On the other hand, if the execution of the task at this end device cannot satisfy all application constraints, the task along with all the relevant sensor data is sent to the edge server that uses a complex multi-objective optimization algorithm for scheduling the execution of the task. In order to determine if this execution can meet all application constraints, our schedulers incorporates a two-level architecture (See Section III-B) and *device profiling* (discussed in Section III-C).

B. Architecture Components

As shown in Figure 1, the edge server has four key components: an Interface Server (IS) component that authenticates and routes user requests, Applications on Edge Server (APe1, ... APen) components that manage different AI applications, Maintain Profile (MP) component that updates a comprehensive device profile table, and a pool of containers—Container Pool (C1, ... Ck) linked to APe's to execute different application tasks.

Each end device has has four key components: an Interface Receiver (IR) to interact with the edge server, a pool of containers to execute different application tasks, an Update Profile (UP) component that periodically sends the current device state (load, battery power, etc.) to the edge server, and Application on End Device (APr1, ... Aprm) components that manage different AI applications and their scheduling. Note that due to the resource constraints of end devices and specific characteristics of the IoT sensors that they manage, the size of their container pool will be significantly smaller than the size of the container pool at the edge server, and they will support only a subset of (relevant) applications from all the applications supported by the edge server.

Our scheduler logically separates the task of determining the current compute and network conditions from making the scheduling decision by incorporating a two-level architecture. The lower level actively monitors and updates current system state (communication latencies, CPU loads, battery power, etc) of all compute devices. Each end device periodically sends its state information (load, battery power, etc.) to the edge server. Further, each end device continually stores and updates its own (current) state information and the edge server stores and updates the state information of all devices.

The upper level of our scheduler architecture makes informed scheduling decisions using the state information being maintained by the lower level. All end devices and the edge server have a separate scheduler component. An end device schedulers make scheduling decisions (execute locally or send the task to the edge server) based on its own state information and profile, while the edge server scheduler makes scheduling decision (which device to execute a task on) using a multi-objective optimization algorithm based on state information of all devices and their profiles.

C. Device Profiling

Every computing task involves constraints like real-time completion, privacy, and cost limits. Determining if a device can satisfy these while running a task requires knowing the time and power consumption under current conditions like CPU load and communication latency. Our scheduler profiles each device in advance for various conditions and tasks. This device profile, along with real-time conditions, guides the scheduler's decisions. This section details the profiling of task runtimes across different conditions and devices.

Equation 1 indicates the total processing time for each task. T_{trans} and T_{reply} represent total data transmission and reply times, while T_{queue} indicates the queuing time on the device, obtained through multi-threading programming. Determining $T_{process}$, or processing time, is more complex due to the diversity of computing tasks, device heterogeneity, and fluctuating workloads.

$$T_{total} = T_{trans} + T_{queue} + T_{process} + T_{reply} \quad (1)$$

Since a microservice-based architecture using containers has become the standard of structuring IoT applications at the edge [22] and due to difficulties in measuring processing time ($T_{process}$) under fluctuating workloads, we recommend using device profiling to establish a detailed profile table, including active container counts to aid the scheduler in decision-making. Further exploration of device profiling will be detailed in Section IV.

D. Predictive Profiling

A key challenge in device profiling is acquiring comprehensive profiling information in advance under every possible workload, which is highly time-consuming and sometimes even infeasible. The goal of predictive profiling is to estimate

the running times of various compute tasks on a device under different workloads (number of containers running concurrently) based on the amount of time it takes to run a single task container with other containers running concurrently. Equation 2 shows how we do this estimation. In this equation, notation T_{ym} indicates the amount of time it would take to run a new container for task *y* while there are *m* containers of task *y* already running at present. The equation illustrates an estimate of the amount of time it would take to run a new task *x* container while there are *m* task *x* containers already running concurrently. This estimate is based on first computing a multiplying factor $(T_{ym} - T_{y1})/T_{y1}$ for a candidate task *y* using actual experiments.

$$T_{xm} = T_{x1} * (T_{ym} - T_{y1})/T_{y1} \quad (2)$$

E. Architecture Workflow

Algorithm 1 The Dynamic Distributed Scheduler Algorithm

Input: profile tables of all computing nodes, time constraint T, original video
Output: number of frames meeting time constraint, video with augmented reality effect

t_{pr1} ▷ predictive running time on R1
t_{pr2} ▷ predictive running time on R2
t_{pes} ▷ predictive running time on the edge server
t_{cr1} ▷ Current time constraint on R1
t_{cr2} ▷ Current time constraint on R2
t_{ces} ▷ Current time constraint on the edge server
t_{q1} ▷ Queuing time on R1
t_{qes} ▷ Queuing time on the edge server
t_{r1es} ▷ Communication delay from R1 to the edge server
t_{esr2} ▷ Communication delay from the edge server to R2
R1_PRO(num) ▷ Function for predictive running time on R1
R2_PRO(num) ▷ Function for predictive running time on R2
ES_PRO(num) ▷ Function for predictive time on the edge server
$t_{pr1} \leftarrow$ R1_PRO(num)
$t_{cr1} \leftarrow T - t_{q1}$
if $t_{pr1} < t_{cr1}$ **then**
    Process this frame locally
else
    Send this frame to the edge server
    $t_{pr2} \leftarrow$ R2_PRO(num)
    $t_{cr2} \leftarrow t_{cr1} - t_{r1es} - t_{esr2}$
    if $t_{pr2} < t_{cr2}$ **then**
        Send this frame to R2
    else
        $t_{pes} \leftarrow$ ES_PRO(num)
        $t_{ces} \leftarrow t_{cr1} - t_{r1es} - t_{qes}$
        if $t_{pes} < t_{ces}$ **then**
            The edge server processes this frame
        else
            Drop this frame
        end if
    end if
end if

Figure 1 shows a distributed scheduler system where the Interface Server (IS) receives user requests and forwards them to the appropriate Application on Edge Server (APe) based on the user's location. The APe identifies the right end device, like a Raspberry Pi with a camera, to fetch data and initiate the requested application. Responses are then sent back to the user. The system uses multi-threading for efficiency and dynamic scheduling based on current loads, monitored by the Update Profile (UP) module, to optimize resource use across devices.

The workflow and pseudo-code outlined in Algorithm 1 detail a two-level decision process for task allocation. At the local level, device R1 evaluates whether to process tasks in-house or offload them to the edge server, based on local computing capabilities and device profiling. At the global level, the edge server uses comprehensive profiling data to either distribute tasks to other end devices or handle them locally, aiming for near-optimal resource utilization.

IV. Device Profile Evaluation

A. Profile Evaluation Scenarios

This section explores four scenarios to simulate real-world application workloads in containers, divided into warm and cold categories. Warm containers are pre-initialized and ready to execute tasks quickly, offering fast response times. Conversely, cold containers require initialization and loading of necessary runtime environments and code upon request, which introduces latency before processing begins.

TABLE I: System Configuration

Component Name	Specifications
Edge Server	2.3GHz Dual-Core Intel Core i5, 8 GB RAM, 256GB Disk
Raspberry Pi	Quad core Cortex-A72 (ARM v8) 64-bit 8GB RAM, 1.8GHz Clock Speed
Smart Phone	Octa-core (4x2.3 GHz Mongoose, 4x1.6 GHz Cortex-A53), 4GB RAM

TABLE II: Profile of Warm Container on the Edge Server

# of containers	1	2	3	4	5	6	7	8
Average Time (ms)	223	273	366	464	540	644	837	947
Total Time (ms)	11193	6930	6216	5951	5794	5507	6020	6099

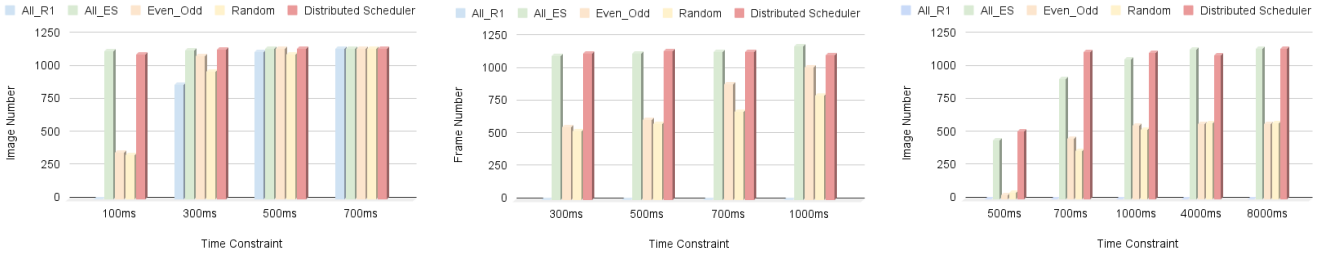
TABLE III: Profile of Warm Container on the Raspberry Pi

# of containers	1	2	3	4	5	6
Average Time (ms)	597	613	651	860	1071	1290
Total Time (ms)	29934	15399	11072	11042	11043	11074

We examine container performance across four scenarios involving N containers, distinguishing between warm and cold statuses: 1) Total processing time for N cold containers; 2) Impact on warm containers when adding a new cold container; 3) Total processing time for N warm containers; 4) Effects on existing warm containers when adding a new warm container.

B. Profile Evaluation Results

For evaluation, we chose face detection using the Viola-Jones algorithm [21]. Performance tests were conducted on a laptop and a Raspberry Pi, with details provided in Table I. For space limit, we show partial results here. Tables II and III



(a) Experiment Results with 240p Video (b) Experiment Results with 480p Video (c) Experiment Results with 240p Video

Fig. 2: Experiment Results with Variable Video Resolution

display warm container test outcomes total processing time for 50 images (29KB).

Based on the above results, we know that first of all, with more containers, average processing times increase due to the limited computing resources. Then for the total processing time for all tasks as shown in Table II, increasing containers from 1 to 2 cuts the processing time for 50 images nearly in half, from 11193ms to 6930ms, utilizing dual-core CPU more efficiently. However, beyond 3 containers, CPU usage nears capacity, with idle CPU dropping close to 0%, leading to minimal time improvements. Consequently, performance plateaus around 6000ms as additional containers no longer enhance efficiency due to CPU load saturation.

The results affirm the approach of aligning the number of active containers with system workloads, establishing a strong basis for dynamic distributed scheduling strategies.

V. Evaluation and Results Analysis

A. An Augmented Reality Application

Augmented Reality (AR) integrates computer-generated content with live video, enhancing real-world scenes via device cameras. Commonly used in security, video conferencing, and healthcare, AR is particularly effective for face detection in streaming videos. The technology functions in three phases: extracting frames, detecting faces, and reassembling frames into their original sequence [23], [24].

Aimed at real-time processing within edge networks, tasks are allocated based on resource intensity, with frame extraction at the end device connected to the camera and face detection distributed across nodes for efficiency. This setup requires a robust scheduling system to ensure seamless application performance by effectively managing computing resources across the network, particularly focusing on the scheduling of face detection tasks within the application. All device profile evaluations have been completed. Due to space limitations, we have included only the results for the 1080p video in Table IV.

TABLE IV: Devices Profile with 1080p Video

# of containers	1	2	3	4
Edge Server (ms)	681	1171	1837	2527
Rasp (ms)	3055	5898	8744	×

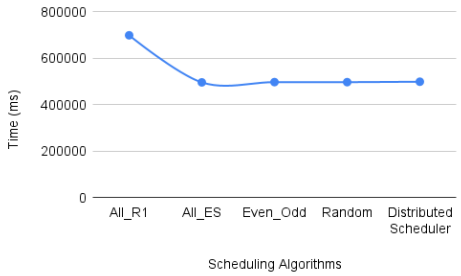


Fig. 3: AR Application End-to-End Time

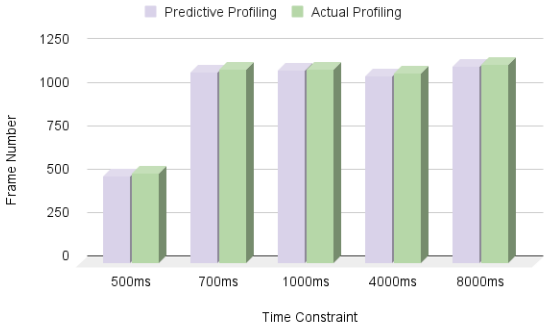
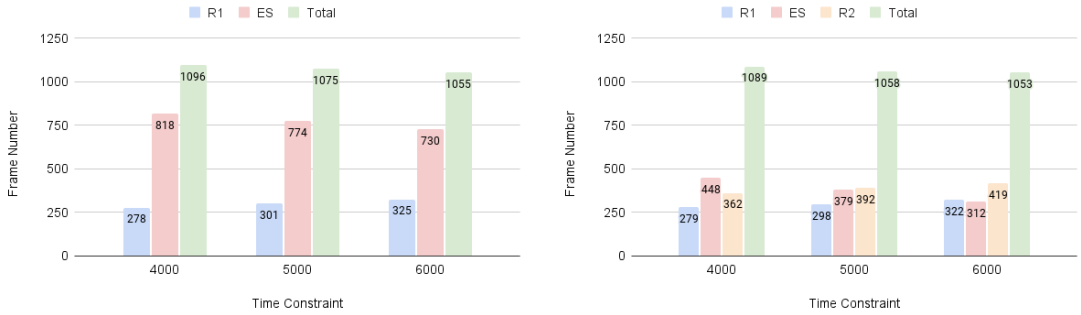


Fig. 4: Predictive profiling vs Actual Profiling

B. Experiment Results

This section details experimental outcomes across different video resolutions, illustrated in Figures 2a, 2b, and 2c for 240p, 480p, and 1080p videos, respectively. We evaluated our scheduler against four static methods: 1) All_R1, where all object detection is done on Rasp 1; 2) All_ES, with all detections on the edge server; 3) Even_Odd, alternating detections between Rasp 1 and the edge server based on frame parity; and 4) Random, where frame processing is randomly assigned between Rasp 1 and the edge server. Figure 2a displays the performance of a 240p video with 1145 frames under four time constraints (100ms, 300ms, 500ms, 700ms).

- As expected, for higher values of time constraints, all algorithms successfully process a higher number of frames.
- Our dynamic distributed scheduler significantly outperforms the single-node algorithms All_R1 and All_ES, especially excelling over All_R1. As shown in Table IV, the edge server's higher processing capabilities boost frame processing success rates. Although All_ES closely matches our scheduler's performance, its heavy workload



(a) Dynamic Distributed Scheduler

(b) Extend Dynamic Distributed Scheduler

Fig. 5: Extend the Dynamic Distributed Scheduler

emphasizes the benefits of our more effective and efficient dynamic distributed scheduling strategy.

- Our dynamic distributed scheduler significantly outperforms the Even_Odd and Random algorithms, especially under tight constraints.

Figures 2b and 2c display results for 480p and 1080p videos with 1145 frames, underscoring the dynamic distributed scheduler's enhanced performance with the growing processing time gap between the Raspberry Pi and the edge server as resolution increases.

Figure 3 illustrates the end-to-end processing times of the AR application for a 480p video, highlighting the efficiency of different scheduling strategies. The All_R1 algorithm takes 697 seconds, significantly longer than the average 495 seconds for other algorithms, indicating a 41% increase in overhead when processing exclusively on the Raspberry Pi. This suggests that offloading tasks to the edge server reduce overhead more effectively than local processing on the Raspberry Pi.

VI. Optimization

We optimize the dynamic distributed scheduler for the following two aspects: first, we propose a predictive profiling solution to avoid completed profile evaluation. Second, we extend the dynamic distributed scheduler to accommodate more computing nodes and further optimize the algorithm.

A. Predictive Profiling and Dynamic Update

With guidance in Section III predictive profiling, leveraging 240p profiling data, we predict 1080p face detection times under various conditions. Table V summarizes these predictions compared with actual evaluation time.

TABLE V: Predictive Profile

# of containers	0	1	2
Predictive Time (ms)	3055	6032	8112
Actual Evaluation Time (ms)	3055	5898	8744

Table V shows that the predicted times align well with actual times. To improve precision, we do runtime profiling as well. In this approach, we start with the predicted profile and then monitor the execution times during runtime to update the profile dynamically. For example, initiating a new container updates the profiling for M active containers with recent data. Figure 4 compares the results with actual and predictive profiling for a 1080p video under various time constraints, revealing similar success rates between both methods.

B. Balance Between the Load on the Edge Server and Success Rate

To evaluate the performance with multiple end devices, Figure 5a demonstrates the processing of a 1080p video under constraints ranging from 4000ms to 6000ms, highlighting individual component performance and Figure 5b shows the system's enhanced performance with the addition of Raspberry Pi 2 (R2).

Upon comparing figures, we observe that under higher time constraints, more frames are processed by end devices, reducing the load on the edge server. While R2 reduces the edge server's workload, its effect on overall performance is slight. Increasing the time constraint from tight to medium improves the frame success rate. However, further loosening constraints diminishes returns. This analysis not only expands the scale of the Dynamic Distributed Scheduler but also offers a strategy to optimize load distribution and success rates across the network.

VII. Conclusion

This study introduces a distributed system designed to facilitate AI applications at the edge through parallel computing. To address the challenges posed by device diversity and the fluctuating computational and network conditions inherent to IoT environments, we introduce a dynamic distributed scheduler that adjusts task allocation in real-time based on current operational states. This scheduler, grounded in comprehensive device profiling, is further refined through two strategies: 1) predictive profiling with dynamic updates during task allocation, and 2) fine-tuning device profiles for enhanced performance. Our experimental findings demonstrate the broad applicability and effectiveness of this dynamic distributed scheduler. For future work, we plan to delve deeper into optimal-level scheduling and integrate it with current research to further enhance scheduling efficiency.

REFERENCES

[1] W. Li, and M. Liewig, "A survey of AI accelerators for edge environment," In Trends and Innovations in Information Systems and Technologies: Volume 2 8 (pp. 35-44). Springer International Publishing, 2020.

[2] D. Liu, H. Kong, X. Luo, W. Liu and R. Subramaniam, "Bringing AI to edge: From deep learning's perspective," Neurocomputing, 485, pp.297-320, 2022.

[3] W. Shi, J. Cao, Q. Zhang, Y. Li and L. Xu, "Edge computing: Vision and challenges," IEEE Internet of Things Journal, 3(5), pp.637-646, 2016.

[4] K. Cao, Y. Liu, G. Meng, and Q. Sun, "An overview on edge computing research," IEEE Access, 8, pp.85714-85728, 2020.

[5] Y. Mao, C. You, J. Zhang, K. Huang, and K. B. Letaief, "A survey on mobile edge computing: The communication perspective," IEEE communications surveys & tutorials 19, no. 4 (2017): 2322-2358.

[6] T. Nepusz, and T. Vicsek, "Controlling edge dynamics in complex networks," Nature Physics, 8(7), pp.568-573, 2012.

[7] R. Han, S Wen, C. Liu, Y. Yuan, G. Wang, and L. Chen. "EdgeTuner: Fast scheduling algorithm tuning for dynamic edge-cloud workloads and resources." IEEE Conference on Computer Communications, pp. 880-889. 2022.

[8] J. Pang, Z. Han, R. Zhou, H. Tan, and Y. Cao. "Online scheduling algorithms for unbiased distributed learning over wireless edge networks." Journal of Systems Architecture 131 (2022): 102673.

[9] Q. Luo, S. Hu, C. Li, G. Li, and W. Shi, "Resource Scheduling in Edge Computing: A Survey," IEEE Communications Surveys & Tutorials 23, no. 4 (2021): 2131-2165.

[10] Y. Deng, Z. Chen, X. Yao, S. Hassan, and A. MA Ibrahim, "Parallel Offloading in Green and Sustainable Mobile Edge Computing for Delay-Constrained IoT System," IEEE Transactions on Vehicular Technology 68, no. 12 (2019): 12202-12214.

[11] H. Badri, T. Bahreini, D. Grosu, and K. Yang, "Energy-Aware Application Placement in Mobile Edge Computing: A Stochastic Optimization Approach." IEEE Transactions on Parallel and Distributed Systems 31, no. 4 (2019): 909-922.

[12] W. Zhang, Z. Zhang, S. Zeadally, H. Chao, and V. C. Leung. "Energy-Efficient Workload Allocation and Computation Resource Configuration in Distributed Cloud/Edge Computing Systems with Stochastic Workloads." IEEE Journal on Selected Areas in Communications 38, no. 6 (2020): 1118-1132.

[13] T. Huang, W. Lin, Y. Li, L. He, and S. Peng. "A Latency-Aware Multiple Data Replicas Placement Strategy for Fog Computing," Journal of Signal Processing Systems 91 (2019): 1191-1204.

[14] Z. Ning, P. Dong, X. Wang, J. J. Rodrigues, and F. Xia, "Deep Reinforcement Learning for Vehicular Edge Computing: An Intelligent Offloading System." ACM Transactions on Intelligent Systems and Technology (TIST) 10, no. 6 (2019): 1-24.

[15] H. Lu, C. Gu, F. Luo, W. Ding, and X. Liu. "Optimization of Lightweight Task Offloading Strategy for Mobile Edge Computing Based on Deep Reinforcement Learning." Future Generation Computer Systems 102 (2020): 847-861.

[16] S. Ranadheera, S. Maghsudi, and E. Hossain, "Computation Offloading and Activation of Mobile Edge Computing Servers: A Minority Game," IEEE Wireless Communications Letters 7, no. 5 (2018): 688-691.

[17] B. Gu, Z. Zhou, S. Mumtaz, V. Frascolla, and A. K. Bashir, "Context-Aware Task Offloading for Multi-Access Edge Computing: Matching with Externalities," IEEE global communications conference (GLOBECOM), pp. 1-6. IEEE, 2018.

[18] D. Zhang, L. Tan, J. Ren, M. K. Awad, S. Zhang, Y. Zhang, and P. Wan, "Near-Optimal and Truthful Online Auction for Computation Offloading in Green Edge-Computing Systems." IEEE Transactions on Mobile Computing 19, no. 4 (2019): 880-893.

[19] J. Konečný, B. McMahan, and D. Ramage, "Federated Optimization: Distributed Optimization Beyond the Datacenter." arXiv:1511.03575 (2015).

[20] Y. Jiao, P. Wang, D. Niyato, and Z. Xiong, "Social Welfare Maximization Auction in Edge computing Resource Allocation for Mobile Blockchain," IEEE International Conference on Communications (ICC), pp. 1-6. IEEE, 2018.

[21] Y.Q. Wang, "An analysis of the Viola-Jones face detection algorithm," Image Processing On Line 4 (2014): 128-148.

[22] R. Morabito, "Virtualization on Internet of Things Edge Devices with Container Technologies: A Performance Evaluation," IEEE Access 5 (2017): 8835-8850.

[23] M. Mekni, and A. Lemieux, "Augmented reality: Applications, challenges and future trends," Applied computational science 20 (2014): 205-214.

[24] F. Hu, K. Mehta, S. Mishra, and M. AlMutawa. "Distributed Edge AI Systems." In Proceedings of the IEEE/ACM 16th International Conference on Utility and Cloud Computing, pp. 1-6. 2023.