DAG-based Task Orchestration for Edge Computing

Xiang Li[(a)], Mustafa Abdallah[(a)], Shikhar Suryavansh[(b)], Mung Chiang[(a)], Kwang Taik Kim[(a)], Saurabh Bagchi[(a)]
(a): Purdue University *(b):* Cisco Systems

***Abstract*—As we increase the number of personal computing devices that we carry (mobile devices, tablets, e-readers, and laptops) and these come equipped with increasing resources, there is a vast potential computation power that can be utilized from those devices. Edge computing promises to exploit these underlying computation resources closer to users to help run latency-sensitive applications such as augmented reality and video analytics. However, one key missing piece has been how to incorporate personally owned, unmanaged devices into a usable edge computing system. The primary challenges arise due to the heterogeneity, lack of interference management, and unpredictable availability of such devices. In this paper we propose an orchestration framework IBDASH, which orchestrates application tasks on an edge system that comprises a mix of commercial and personal edge devices. IBDASH targets reducing both end-to-end latency of execution and probability of failure for applications that have dependency among tasks, captured by directed acyclic graphs (DAGs). IBDASH takes memory constraints of each edge device and network bandwidth into consideration. To assess the effectiveness of IBDASH, we run real application tasks on real edge devices with widely varying capabilities. We feed these measurements into a simulator that runs IBDASH at scale. Compared to three state-of-the-art edge orchestration schemes and two intuitive baselines, IBDASH reduces the end-to-end latency and probability of failure, by 14% and 41% on average respectively. The main takeaway from our work is that it is feasible to combine personal and commercial devices into a usable edge computing platform, one that delivers low and predictable latency and high availability.**

***Index Terms*—Edge Computing, Directed Acyclic Graphs, Task Orchestration, Service Time.**

I. Introduction

There has been a surge of latency-sensitive applications running on user-generated streaming data, such as augmented reality and video analytics. Such a surge has driven the wide popularity of edge computing since it offers low latency by performing the computation near the source of the data and offers scalability by distributing the workload among multiple edge devices. After some notable innovations in academic publications over the last few years, we have started seeing the growth of small, edge-located data centers managed by infrastructure providers such as Amazon [1], Microsoft [2] and Google [3]. We refer to such devices as "**Commercial Edge Devices (CEDs)**". Being commercially managed, CEDs are expected to be available over extended periods and achieve reasonably low latencies. However, they have the drawbacks of incurring $ cost and still not being widely available. Almost all existing literature on edge computing systems implicitly deals with CEDs as they assume the above desirable properties.

Edge computing systems could also comprise personal devices such as laptops, tablets, and mobile phones. This trend is increasing as such devices are becoming more ubiquitous and are increasing in their compute power. By the end of 2020, it is estimated 6.06B smartphones were in use globally, which is three times the number of PCs, and it is expected to keep growing at a 4% rate and hit 7.69B by 2026 [4]. Moreover, smart devices now have significant storage and processing power and this trend is continuing. For example, a 2010 Samsung Galaxy S only had just 512MB of RAM and 8GB of storage with a single core at 1 GHz, but the 2021 Galaxy Z Fold3 comes with 12GB of RAM and 256GB of storage with 8 cores clocking at a maximum of 2.84 GHz. We call such devices that may be pulled into an edge computing system as "**Personal Edge Devices (PEDs)**". Running latency-sensitive applications on PEDs is appealing as they are often closer to the user than CEDs, with the same user often carrying multiple PEDs. Furthermore, with the right kind of incentive schemes, the usage of PEDs can be at zero cost. On the other hand, such devices are expected to have sporadic availability and have little to no way to manage contention that can arise due to multiple co-located applications. Therefore, one has to carefully manage such PEDs in an edge computing system to achieve reliable and low latency executions.

In this paper, we present the design of an edge computing task orchestration scheme that we call IBDASH[1] that combines PEDs and CEDs into one system to leverage the benefits of both types of devices. Ours is the first appoach to combine PEDs and CEDs into a single edge computing system, while supporting reliable and low-latency execution. In particular, IBDASH introduces a method to schedule complex latency-sensitive applications, whose tasks (stages) have dependencies and can be represented by directed acyclic graphs (DAGs). The scheduling happens among available PEDs and CEDs to reduce the end-to-end execution latency of each application and the probability of application failure, while accounting for the dependencies among different stages of the application. For example, a video analytic application may do scene change detection and pass onto a second task that does object detection only if there is a scene change. IBDASH also takes into account the interference on a particular device from multiple co-located applications. This is particularly important because in our target class of devices, there do not exist good hardware mechanisms for avoiding contention, unlike in server-class devices [5].

Table I compares the features of our proposed solution IBDASH with prior related works. In particular, prior works such as LAVEA [6] and Petrel [7] propose their orchestration schemes that target client-edge offloading. LAVEA [6] seeks to provide low-latency video analytics while Petrel [7] provides randomized load balancing by leveraging the "power of two" choice [8] to randomly choose two edge devices and allocate the task to the one which gives better performance. Moreover, LaTS [9] allocates the task to the device that has the shortest estimated latency based on a pre-profiled latency-CPU usage

[1]IBDASH stands for **I**nterference **B**ased **D**AG **A**pplication **Sc**H**e**duler and is pronounced as [ˈīb-dash] and its code is available at the anonymous repository: https://github.com/SRDS-2022/ibdash

TABLE I: A comparison between the prior related works and our system in terms of the available features. IBDASH offers DAG support, failure reduction and memory consideration.

Framework	Failure reduction	Supporting DAG	Heterogeneous edge devices	Memory Consideration	Low latency	Orchestration overhead reduction
LaTS [9]	✗	✓	✓	✗	✓	✗
Petrel [7]	✗	✗	✓	✗	✓	✓
LAVEA [6]	✗	✓	✓	✗	✓	✗
Edge OD [11]	✗	✗	✗	✗	✓	✗
JCAB [10]	✗	✗	✗	✗	✓	✗
IBOT [12]	✓	✗	✓	✗	✓	✓
IBDASH (This work)	✓	✓	✓	✓	✓	✗

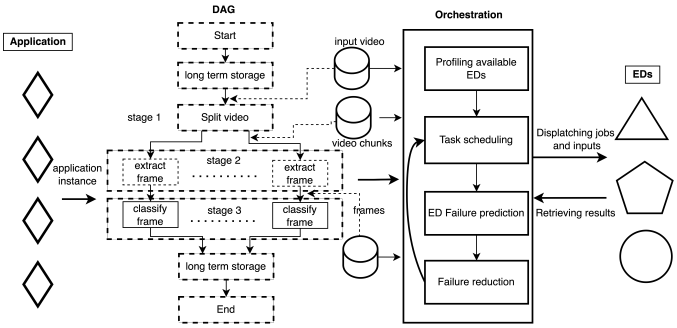


Fig. 1: A system overview of IBDASH orchestration scheme for DAG based application. All edge devices in the network need to be profiled before task allocation. IBDASH do pre-processing the tasks in the application and puts them into different stages, then the scheduler orchestrates the allocation for each task's execution to reduce execution latency and probability of failure.

model. On the other hand, JCAB [10] effectively balances the accuracy and energy consumption while keeping low system latency by jointly optimizing configuration adaption and bandwidth allocation. Liu et al. [11] propose a system that employs low latency offloading techniques jointly with pipeline decoupling methods and fast object tracking methods to enable accurate object detection. However, among these frameworks, some do not consider the heterogeneity of edge devices at all [10, 11] and among those that consider the heterogeneity of edge devices, Yi et al. [6] considers the heterogeneity of CPU architectures, Zhang et al. [9] considers the heterogeneity of CPU and GPU mix, and Lin et al. [7] considers the mix of devices as a cloudlet entity. *None of these works considers the mix of PEDs and CEDS.* Moreover, most of them (except LaTS [9]) do not address the interference of co-located tasks on the same edge device. The previous work IBOT [12] proposed an orchestration scheme that takes the interference among tasks on each edge device into account and orchestrates an optimal execution strategy that jointly optimizes both execution latency and probability of failure. However, IBOT treats each task separately and does not take into account dependencies among tasks within an application, it cannot handle application DAGs. Also, it fails to address the memory constraints of each edge device since some tasks may require loading models into memory to successfully carry out the task execution.

Figure 1 shows an overview of our proposed solution IBDASH. Each application instance from the end-user consists of one or more tasks, which may have dependencies among them. For example, in Figure 1, we show a DAG example of a video analytics application that shows control and data dependencies among stages, where the dashed arrow shows the input of each stage of the DAG, and the solid arrow indicates the data flow. The application is explained in detail in Section V-B.

In our evaluation, we compare IBDASH with two intuitive baselines, Random allocation and Round Robin, and three state-of-the-art solutions, LAVEA [6], Petrel [7], and LaTS [9]. To test our framework, we use four applications that span various DAG structures from different application domains, with some tasks requiring models and some that do not. For example, if the task performs object recognition, a pre-trained model is needed on the designated edge device before the task can start running. Compared with existing schemes, IBDASH reduces the average service time of applications by 14% and the average probability of failure for the application by 41% compared to the best baseline scheme.

In summary, this paper makes the following contributions:

1) We propose an orchestration framework IBDASH, an interference-based dynamic task orchestration scheme that executes DAG-based user applications in a heterogeneous edge computing environment that comprises both commercial and personal edge devices (called CEDs and PEDs by us).
2) To increase the reliability of edge computing, our solution strategically replicates the tasks that are allocated to devices with a high probability of failure.
3) We propose a device availability prediction model and validate it through the data collected from a real-world mobile crowdsourcing dataset [13].
4) We validate our model via extensive simulation of application instances that arrive randomly within a time period and shows its superiority in reducing average application service time and probability of failure.

II. Problem Statement

The combination of CEDs and PEDs, task dependencies within applications, and sporadic availability of PEDs pose unique challenges that have not been addressed in the edge computing literature. We discuss the four primary challenges, whose solutions bring out the novelty in IBDASH.

Substantial heterogeneity in computational capacity: PEDs such as laptops, tablets and mobile devices can have a substantial variation in their compute power, memory, etc. For example, in the current smartphone market, Samsung, Apple, and Xiaomi contribute 20%, 14%, and 13% respectively to the market share and others contribute the remaining 53%. Within each brand, there is a wide range of devices with different capabilities that target different customers. The penetration of different brands in different markets and economies varies widely leading to a natural heterogeneity in the PEDs.

Heterogeneity in task interference pattern: Different tasks, when running on the same edge device, interfere with each other affecting their execution time. There is heterogeneity in the interference experienced by different tasks on an edge device. For example, suppose that we have three tasks where the first task (t_0) loads a set of images, the second task (t_1) performs convolution on the pre-loaded images, and the third task (t_2) rotates the processed images. For such a scenario, Figure 2 shows the different interference patterns for different

task types and the different CPU usages for different task types. In Figure 2a, we see that the interference pattern for $t_1 - t_1$ is different from the interference pattern for $t_2 - t_1$. Figure 2b shows that the CPU usage for each task under three different scenarios also varies. The interference patterns and CPU usage differences can be due to multiple reasons, such as resource contention [14], priority scheduling [15], etc. The main insight to note is that the different interference patterns among tasks result in different service times.

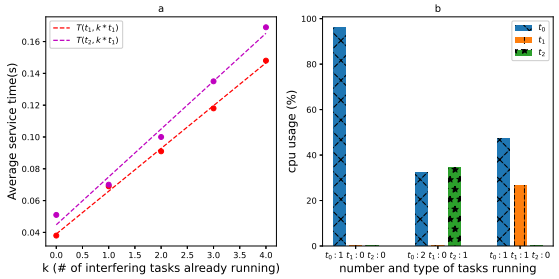


Fig. 2: (a) The heterogeneity in interference among different tasks. (b) The CPU usage of each task varies when different tasks are running in the background.

Sporadic availability of PEDs: Due to the unmanaged nature of the PEDs, their availability in the network can be hard to predict. For example, in a more predictable scenario such as a classroom setting, when students leave the classroom, their laptops and mobile devices will not be available anymore so any tasks that are scheduled close to the end of class will experience a high probability of failure. However, in other scenarios where people come and go less predictably (e.g., a university library), it would be hard to predict the availability. For instance, Zhang et al. [13] did an experiment to track the availability of the mobile devices of students on a university campus. The results show that the probability of failure for mobile devices (mobile devices disconnect from the crowdsensing framework) increases with the length of time that elapses since they connected to the framework.

DAG-based application orchestrations: The dependency among tasks within the same application adds one more layer of complexity to the orchestration problem as the execution of tasks need to follow a certain order. Some can be executed in parallel and some cannot. Prior work [16] shows that several partition algorithms [17, 18] are developed to achieve different optimization goals such as saving energy [19, 20], and reducing execution latency [20]. Our proposed framework IBDASH utilizes the structure within DAG-based application where it explicitly characterizes the task flow and data flow. Then, we orchestrate the task allocation to reduce the overall end-to-end latency and probability of failure. We emphasize that the significant prior work on DAG scheduling in the cloud is less relevant in our context as our EDs are more heterogeneous and less predictable in their availability.

III. Preliminaries and Notations

A. Feasibility of PEDs: A survey

We conducted a user study with 110 participants from USA and India that are engaged in diverse fields such as educators, software professionals, students, engineering professionals, etc. to understand their willingness to share their computing devices (e.g., laptops, desktops, tablets, etc.) as edge devices. Figure 3a shows that 86.4% of the participants are willing to share their devices under one of four proposed incentive schemes. Only 13.6% of the participants were not interested in sharing their devices at all, primarily due to privacy and security concerns. Moreover, Figure 3b indicates that the majority of the participants are willing to share 0-40% of their CPU resources. The amount of CPU resources people are willing to share varies as well depending on the device type. As shown in Figure 3c, we obtained a double Gaussian device usage pattern with peaks at 90% and 30% of usage indicating that most people either use their devices very heavily (video editing, running sophisticated software, etc.) or use them only for computationally light applications such as browsing, reading, etc. The average usage across all users was 50.9%, thereby supporting our claim that a lot of devices are not utilized to their capacity.

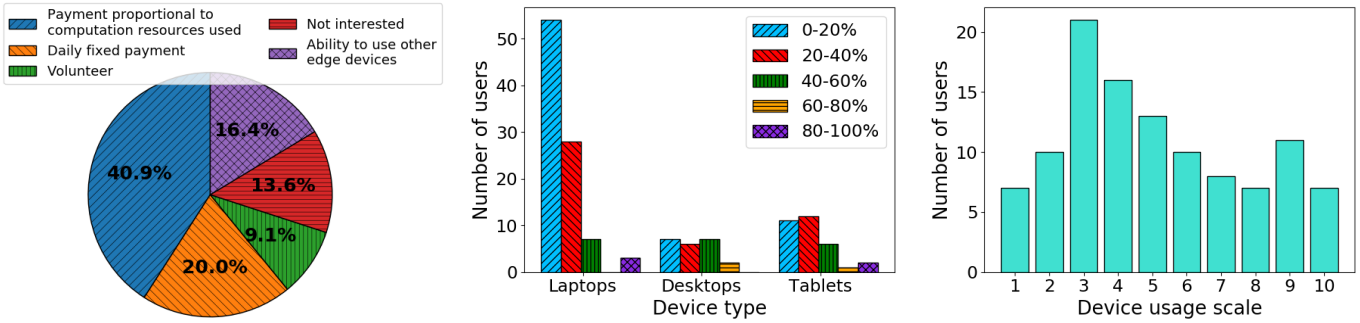
Now, we introduce the notations and terms used in our framework IBDASH and show them in Table II.

TABLE II: The list of symbols and definitions used in our work.

Symbol	Definition
$T = \{T_1, T_2, \ldots, T_N\}$	Types of tasks for a given application
$ED = \{ED_1, \ldots, ED_N\}$	Set of all edge devices
$S = \{S_1, \ldots, S_N\}$	Number of stages in DAG
$G = (V_i, E_j),\ V_i \in T$	DAG representation of the application
$L(T_i)_{ED_j}$	Execution latency of T_i on ED j
$L(M(T_i))_{ED_j}$	Model download latency for T_i to ED j
$H(ED_j)$	Memory available on edge device j
$H(T_i)$	Memory required for T_i
$L(T_i)_d$	Data transfer latency for Task i input
$L(T_i)$	End-to-end latency for task T_i
$L(G)$	End to end latency for the application
$T(i)_d$	Input data for task T_i
$L(S_i)$	End-to-end latency of stage i
$P(T_i)$	Placement of task T_i
$D(T_i)$	Dependency of task T_i in terms of other tasks
$P(G)$	Placement of each task in graph G
$P_f(G)$	Probability of failure of application, given by graph G
T_{rep}	Tracker for the number of replications
β	Probability of failure threshold
γ	Threshold on the Replication degree
$F(T_i)$	Probability that T_i fail
B	Network bandwidth
$M(T_i)$	Model required for T_i
Ed_{info}	Total and free space on each ED
M_{info}	Available models on each edge device
$Task_{info}$	# executing tasks and types on each ED
$WeightS$	Weight score of joint optimization
$WeightSnew$	Weighted score after PF reduction

B. Directed Acyclic Graphs

Each application is represented as a directed acyclic graph (DAG) $G = (V, E)$. The set of nodes V represents the individual tasks in an application instance, while the set of edges E characterizes the dependency among those tasks. The dependency can mean both execution order dependency and data dependency. In particular, an edge from task v_i to task v_j indicates that v_i needs to be finished before the start of v_j.



(a) Preferred incentive model

(b) Percentage of CPU resources willing to share

(c) Device usage (1: Low usage, 10: Very high)

Fig. 3: User survey results (N = 110)

C. End-to-end Latency and Probability of Failure

Throughout the paper, we use the terms end-to-end latency and probability of failure to describe the goal of joint optimization. End-to-end latency is defined as the time from when the first task in the DAG-based application starts executing till the last task finishes its execution. In this paper, we use the average end-to-end latency of hundreds of application instances to evaluate the performance of our orchestration strategy. The term probability of failure is defined as the probability that the application instance did not successfully finish its execution. An application instance may not complete its execution because one of the edge devices becomes unavailable in the middle of task execution, or the owner of the edge device decided to perform some heavy-duty task, which results in the edge device being less responsive.

Having explained the problem statement and the main notations used in our work, we now turn our attention to the design of IBDASH.

IV. Our Proposed Solution: IBDASH

To target the problems we listed in Section II, we propose the framework IBDASH, which is an interference-based orchestration scheme for DAG-based applications that aims at jointly optimizing the end-to-end latency and probability of failure for application instances. The rest of this section covers the task interference in our framework, the components of the framework, and our proposed orchestration algorithm.

A. Interference service time plots

We define interference as a linear service time plot $T_i = m_j * k + c_j$ that characterizes the execution time of a new task of type T_i on ED_p, given that k tasks of type T_j are already running on that edge device. For example, on a given edge device, for a new task T_i, we can plot N interference plots for every other type of task including T_i itself. Therefore, there are overall N^2 such plots and N^2 pairs of m and c values to characterize all interference plots for that edge device. The expected service time of the new incoming task T_i on ED_p, which has α_1, α_2, ..., α_N running tasks is given by:

$$f_{i,(1,2,\ldots,N)}(T_i, (\alpha_1 T_1, \ldots, \alpha_i T_i, \ldots, \alpha_N T_N)) = \sum_{j=1}^{N} f_{ij}(T_i, \alpha_j T_j) \quad (1)$$

The above equation assumes that the interference patterns are independent and additive, which we verify experimentally (Section V Figure 4). A lower interference coefficient (m, c values) of an application for a device means the a shorter estimated execution latency for running that application on the device. A pairwise interference coefficient matrix has been defined as ED_{mc}, in which each row contains N^2 pairs of m, c values for that particular edge device. The element $< m_{ij}, c_{ij} >_p$ means that if we want to schedule a new task of type T_i while k instances of T_j are already running on edge device p, the estimated service time for T_i can be calculated as $k * m_{ij} + c_{ij}$. We use the matrix $Task_{info}$ to record the allocation of each task and the estimated time it will be on that edge device, then we calculate the number of running tasks on each device at a certain time through simple summation.

B. Design Components

The proposed framework contains three main functions, DAG transformation, minimum service time scheduling, and failure likelihood reduction. In IBDASH, we adopt a partially peer-to-peer architecture where every CED in the network (but not the PEDs) can take up the role of orchestrator as well as perform its core function of an edge device. When a new application instance arrives, the framework first transforms the application's DAG and divides the execution into stages. The advantage of dividing the DAG into stages is that the dependency of the tasks is embedded within the stages and all tasks within the same stage can be executed in parallel. Figure 1 shows an example of the DAG for the video analytics application.

Determining the stage of a node in the DAG is performed through modified Breadth-First Search where the stage of a node is the length of the longest path from the start node. After staging the DAG, the orchestrator uses the profiling data, detailed in Appendix B, (saved in matrix ED_{mc}) to retrieve the interference coefficients ((m, c) value pairs) and using $Task_{info}$ matrix to retrieve the number of running tasks of each task type on the edge device of interest. The execution latency of the task is estimated using Equation 1 and denoted as $L(T_i)_{ED_p}$. Besides the execution latency, some tasks may require models to successfully execute the task. Therefore, we introduce the term $M(T_i)$ to denote the models required for task T_i (this may be null in cases) and $L(M(T_i))_{ED_p}$ to

express the corresponding model downloading latency, which depends on both model size and network bandwidth B.

Another important latency factor is the data transfer latency of the input data $T(i)_d$ for task T_i if it is from other EDs, and this is expressed as $L(T_i)_d$. The device p that gives the minimum latency of the task T_i, which is the summation of the minimum execution latency, model downloading latency and data transmission latency for T_i. The choice of the edge device minimizes this combined latency and is given by

$$\begin{aligned} &\arg\min_{p} L(T_i) \\ &where\ L(T_i) = L(T_i)_{ED_p} + L(M(T_i))_{ED_p} + L(T_i)_d \\ &s.t.\ B \leq B_{max},\ H(T_i) \leq H(ED_p),\ ED_p \in ED \end{aligned}$$

Here, B is the current network bandwidth, B_{max} is the maximum available network bandwidth, $H(T_i)$ is the memory required for T_i's execution, including memory to store data and model, and $H(ED_p)$ is the available memory on ED_p.

Now, let us define $L(S_i) = \max_{T_i \in S_i} L(T_i)$ as the stage i latency. Therefore, the end-to-end latency of the entire application is the sum of the longest latency task in each stage.

$$L(G) = \Sigma_{i=1}^{i=S} L(S_i) \tag{2}$$

In the end, due to the sporadic availability of PEDs, IBDASH adds redundancy to replicate tasks that are assigned to edge devices with a high probability of failure to other edges devices. The goal of this redundant replication is to reduce the average probability of failure of the application instance below a certain threshold or to the minimum probability of failure within the replication degree constraints. For each edge device that was chosen to execute task T_i, we predict the probability of failure of that edge device during the estimated service time of task T_i and this is the probability of failure of task T_i, which is denoted by $F(T_i)$. If $F(T_i)$ is above a certain threshold β, IBDASH replicates T_i on another edge device which gives the next optimal minimum service time. We keep repeating the replication until $F(T_i)$ is reduced below the probability of failure threshold or the number of replications for T_i reaches the maximum replication degree γ. Every task within the application instance needs to be successfully executed so that the entire application can be counted as successful. Therefore, our framework seeks to minimize the probability of failure of every single task. We use $1 - F(T_i)$ to denote the probability of success for T_i, then use the intersection of the probability of success for all tasks to denote the probability of success for the entire application instance. Such intersection notation captures the dependence among tasks (which is captured by conditional probabilities). In our setup, we used the data to estimate such conditional probabilities of task successes.

$$P_f(G) = 1 - \cap_{i=1}^{N}(1 - F(T_i)) \tag{3}$$

The final optimization problem is given by

$$\min \alpha L(G) + (1 - \alpha) P_f(G), \tag{4}$$

where α is the joint optimization parameter that is controlled by the user to give proper weight for end-to-end latency and probability of failure. This kind of joint optimization problem formulation by attributing different weights to different metrics, which are linearly related, has been widely used in literature [20, 21, 22, 23].

C. Orchestration algorithm description

The orchestration algorithm is shown in Algorithm 1. It greedily examines each task on available edge devices, while concurrently considering the allocation of its prerequisite tasks to minimize application latency and the likelihood of failure globally. The algorithm outputs the placement choice for each task in the application on an edge device, PED or CED. The orchestration time complexity can be expressed as $O(|T| * N)$, where $|T|$ is the total number of tasks in the application and N is the number of edge devices in the network. Therefore, the orchestration overhead has a linear relationship with the number of edge devices in the network and the number of tasks in the application. Our results show that the orchestration overhead is around 10% of the average service time of an application instance when more than 50 devices are present, and around 20% when more than 500 tasks need to be orchestrated in one application instance. The detailed experimental results are shown in Figures 16-18 in Appendix C.

Lines 4-17 check each task in each stage against all available edge devices for estimated latency and consider the extra data transmission and model downloading latency globally when tasks with dependency are assigned to different devices. By the end the process, we have the expected latency of T_i on each edge device. Lines 19-27 remove infrequently used model to free space and downloads the required models to the targeted edge device and updates the M_{info} structure which is used to keep track of the model availability on each device and ED_{info} structure, which is used to keep track of the memory available on each device. Line 28 calculates the probability of failure of the task on the device based on its dependency then a weighted optimization score is calculated in line 29.

Now, the algorithm checks the probability of failure of the current allocation against the preset threshold β. If the probability of failure is greater than the threshold and the number of replications for the task is less than max replication degree γ, line 31 dequeues the next item and lines 32-38 recalculate the weighted optimization score with the new probability of failure and the execution latency. If the new weighted optimization score is less than the previous weighted score, we replicate the task. Then repeating this process until the probability of failure is below the threshold β, or replication degree is reached, or the queue is exhausted. Line 40 updates the $Task_{info}$ on the selected device and the finishing time of those tasks already running on the device is adjusted correspondingly. Line 41 records the task allocation and line 44 records the longest latency in the current stage and makes sure that tasks from the next stage will not start until the previous stage is finished due to dependency. Finally, line 46 keeps track of the end-to-end latency of the entire application.

V. Experiments

We seek to answer the following research questions:

- RQ1: What is the interference pattern among tasks?

Algorithm 1: Orchestration Algorithm

Input : DAG for application instance G
Output : Task placement $P(T_i) \forall T_i \in T_1, T_2, \cdots, T_N$, application end-to-end latency $L(G)$
Initialization: ED_{mc}, ED_{info}, $Task_{info}$

$P.init()$ # init placement structure
$S = app\_stage(G)$ # stagerize the DAG
for $S_i \in S$ **do**
  for $T_i \in S_i$ **do**
    for $ED_p \in ED$ **do**
      $L(T_i)_{ED_p} = GetEstimatedTime(T_i, ED_p)$
      $L(M(T_i))_{ED_p} = 0$
      if $M(T_i)$ *not on* ED_p **then**
        $L(M(T_i))_{ED_p} = GetMdUpTime(size(M(T_i)), B)$
      end
      $L(T_i)_d = 0$
      if $T(i)_d$ *not on* ED_p **then**
        $L(T_i)_d = GetDTrTime(size(T(i)_d), B, D(T_i))$
      end
      $L(T_i) = L(T_i)_{ED_p} + L(M(T_i))_{ED_p} + L(T_i)_d$
      $PQueue.enqueue([ED_p, L(T_i)])$
    end
    $ED_p,\ L(T_i) = PQueue.dequeue()$
    if $M(T_i)$ *not on* ED_p **then**
      while $H(ED_p) \leq H(T_i)$ **do**
        $M_{info}[ED_p].removeEnd()$
      end
      $M_{info}[ED_p].add(M(T_i))$
      $ED_{info}.Update()$
    else
      $M_{info}[ED_p].moveFront(M(T_i))$
    end
    $F(T_i) = GetPf(T_i, ED_p, L(T_i)_{ED_p}, D(T_i))$
    $WeightS = \alpha L(T_i) + (1-\alpha)F(T_i)$
    while $F(T_i) \geq \beta$ *and* $T_{rep} < \gamma$ **do**
      $ED_p,\ L(T_i) = PQueue.dequeue()$
      $F(T_i) = GetPf(T_i, ED_p, L(T_i))$
      $WeightSnew = \alpha L(T_i) + (1-\alpha)F(T_i)$
      if $WeightSnew \leq WeightS$ **then**
        $P(T_i).add(ED_p)$
        $WeightS = WeightSnew$
        $T_{rep}++$
      end
    end
    $Task_{info}.Update(ED_p)$
    $P(T_i) = min(\alpha L(T_i) + (1-\alpha)F(T_i))$
  end
  $L(S_i) = max_{T_i \in S}(L(T_i))$
end
$L(G) = \sum_{i=1}^{i=S} max(L(S_i))$
return : $P(T_i) \forall T_i \in \{T_1, T_2, \cdots, T_N\}$, $L(G)$

- RQ2: How does IBDASH's performance compare to other baseline schemes with respect to end-to-end latency and failure likelihood?
- RQ3: How to predict the availability of edge devices?
- RQ4: How do the parameters α, γ affect IBDASH?

A. Interference pattern verification

In Section IV-A, we assume that the interference patterns among tasks are independent and additive, and we find that this assumption holds across all different architectures, tasks, and OSes. We validate this assumption through multiple experiments with a wide variety of types of tasks and devices. In particular, three generic types of tasks that cover three major bottlenecks of applications are used — computation-intensive tasks, network/IO tasks, and memory-intensive tasks. The tests are performed on multiple devices with different architectures (ARM and x86), different OSes (macOS, Android, and Linux) and different platforms (mobile phones and PCs). The comprehensive validation results are shown in Appendix A (see Figures 12-14). From these we conclude that our assumption of linear and additive interference holds across a wide variety of real-world scenarios. We consider cases where the assumption does not hold in Section VII.

One experimental result is shown below where Figure 4 is verified on a PC and a mobile device with two computationally intensive tasks. In particular, task t_1 represents matrix multiplication of randomly generated floating point entries 100 times and task t_2 denotes inversion of matrices with the above randomly generated floating entries. The two platforms used in this experiment are a Macbook Pro (3.1GHz dual-core Intel Core i5 and 8GB 2133 MHz LPDDR3 memory) and a Huawei Nexus 6P (Qualcomm Snapdragon 810 with 3GB RAM), as these reflect two possible PEDs with widely varying capabilities. The matrix size used in tasks t_1 and t_2 are 1000×1000 and 100×100 for MacBook Pro and Nexus 6P, respectively. From Figures 4a and 4b, we see that the average execution latency has a linear relationship with the number of (same type) tasks running on the edge device for both platforms by looking at $T(t_1, k*t_1)$ and $T(t_2, k*t_2)$. Moreover, we also see that the lines representing $T(t_2, j*t_1 + k*t_2)$ and $T(t_2, j*t_1) + T(t_2, k*t_2)$ have almost complete overlap. The same pattern is also observed for $T(t_1, j*t_1 + k*t_2)$ and $T(t_1, j*t_1) + T(t_1, k*t_2)$, which verifies our assumption that the interference pattern is additive.

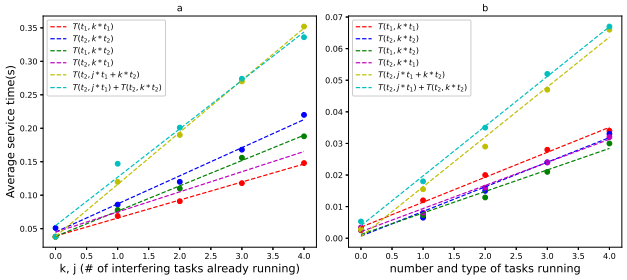
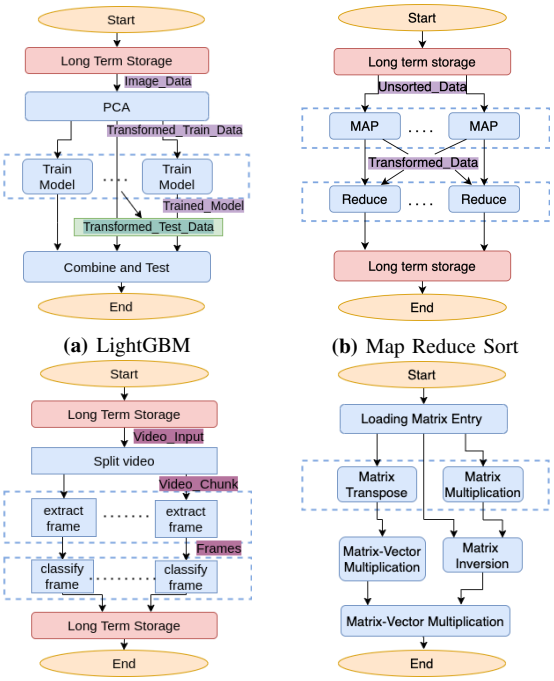


Fig. 4: Interference pattern verification, j,k are the number of tasks of t_1 and t_2 running on the ED. (a) Interference pattern on MacBook Pro. (b) Interference pattern on Huawei Nexus 6P.

B. Testing Applications

The simulator for our proposed framework IBDASH is built in Python and four applications (Figure 5) from different application domains such as machine learning (LightGBM), data analytics (Mapreduce sort), mathematics (Matrix computation) and video analytics, that span a variety of dependency levels among tasks are used for testing the generality of IBDASH.



(a) LightGBM **(b)** Map Reduce Sort

(c) Video Analytic **(d)** Matrix Computation

Fig. 5: DAG applications under test

(1) LightGBM: (Figure 5a) It trains decision trees and combines them to form a random forest predictor. It first reads the training examples and performs dimensionality reduction (PCA). Then, a user-specified number of functions train the decision trees in parallel (every function randomly selects 90% for training, 10% for validation). In the end, all trained models are collected and combined to get tested on held-out test data. The inputs are the handwritten image databases: NIST (800K images) and MNIST (60K images).

(2) MapReduce Sort: (Figure 5b) In the first stage, the parallel mappers (MAPs) fetch input data and generate intermediate files. In the next stage, the reducers sort the intermediate file and write the result back to the storage.

(3) Video Analytics: (Figure 5c) In the first stage, the input video is split into multiple chunks, then a significant frame is extracted from each chunk. Eventually, the significant frame will be used for classification.

(4) Matrix Computation: (Figure 5d) This has tasks that are heavy matrix computations. In particular, the tasks used in this application are matrix inversion, matrix-matrix multiplication and matrix-vector multiplication.

C. Baseline Systems

We compare ourselves with the following baselines:

LAVEA [6]: LAVEA is a system built to offload computation between clients and edge nodes to provide low-latency video analytics. We compare our scheme with their best performing scheme, Shortest Queue Length First (SQLF), which tries to balance the number of tasks running on each edge device.

Petrel [7]: Petrel is a randomized load balancing framework that utilizes the strategy of 'the power of two choices' [8]. The framework randomly selects two edge devices and offloads the task to the one that has the lowest expected service time.

LaTS [9]: LaTS is a latency-aware task scheduling heuristic that distributes tasks to edge devices based on the predicted execution latency of each task through a model which characterizes the relationship between execution latency and CPU usage of the edge device node.

Round Robin: In this scheme, the tasks in each application instance are distributed to edge devices present in network in round-robin manner.

Random: In this scheme, the task will be distributed to edge devices present in network randomly.

D. Performance Metrics

Service Time: We define the service time for an application instance scheduled by the orchestrator as the end-to-end latency, starting from the execution of the first task until the last task finishes its execution. In our simulation, application instances may arrive in a clustered manner, which can cause tasks to accumulate on edge devices and result in longer end-to-end latency for some instances. Therefore, the average service time for a single application instance across all application instances of all applications throughout the entire simulation period is used in our measurement and is measured in seconds.

Probability of Failure (PF): The probability of failure for an application instance is defined as $1 - P_{(all\_success)}$, where $P_{(all\_success)}$ denotes the probability that all tasks composing the application instance (not counting the replicas) are executed successfully. Tasks can fail due to sporadic availability of edge devices or tasks taking abnormally long to execute (e.g., a person leaves the room with his laptop during the middle of the task execution).

E. Evaluation of Heterogeneity

Recall that heterogeneity across edge devices is captured in our work through various computing powers (Section V-B) of different edge devices and a mix of PEDs and CEDs. To show the impact of heterogeneity, we used the exponential function $P(ED_i) = e^{-\lambda t}$ to simulate the sporadic availability of different edge devices with different failing rates (different λs) for different devices. The exponential is chosen as the average probability of failure increases as time passes.

This experiment is meant to validate that the exponential model can be used as a good prediction of the probability of availability of edge devices and the set of λ values used in our simulation are realistic. We used the mobility trace from [13] to validate our assumption. The mobility data is collected over one month (Feb 7th - Mar 7th, 2018) with 50 users on a university campus. Each user was performing their daily routine while their smartphones were running tasks for

collecting sensor data, such as geolocation. The missing data points in the data sets indicate students turned off their devices, have no network access or quit the data collection program for their reasons. By analyzing the mobility data, we show the corresponding results in Figure 6a, which shows the change in probability of availability since it first becomes available.

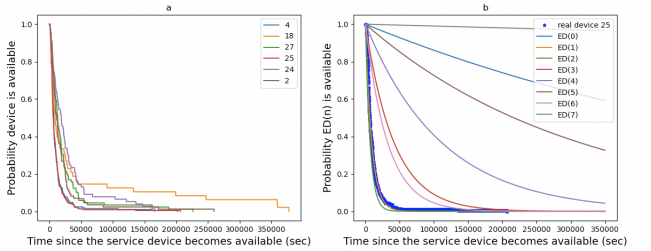


Fig. 6: Availability of edge device throughout the simulation time. (a) Probability of 5 random devices' availability in mobility data collected from a university campus [13] (b) Probability of device availability used in simulations.

TABLE III: Failure rates λ used in simulation. λ_1 = Mix, λ_2 = CED, λ_3 = PED.

ED devices	ED0	ED1	ED2	ED3	ED4	ED5	ED6	ED7
λ_1	$1.5*10^{-6}$	$1.1*10^{-4}$	$1.5*10^{-4}$	$2.4*10^{-5}$	$9*10^{-6}$	$3.2*10^{-6}$	$3.1*10^{-5}$	$1*10^{-7}$
λ_2	$1.5*10^{-5}$	$1.1*10^{-5}$	$1.5*10^{-5}$	$1.1*10^{-5}$	$1.8*10^{-5}$	$1.2*10^{-5}$	$1.0*10^{-5}$	$2.0*10^{-5}$
λ_3	$1.5*10^{-4}$	$1.1*10^{-4}$	$1.5*10^{-4}$	$2.4*10^{-4}$	$9*10^{-4}$	$3.2*10^{-5}$	$1.0*10^{-4}$	$9.0*10^{-4}$

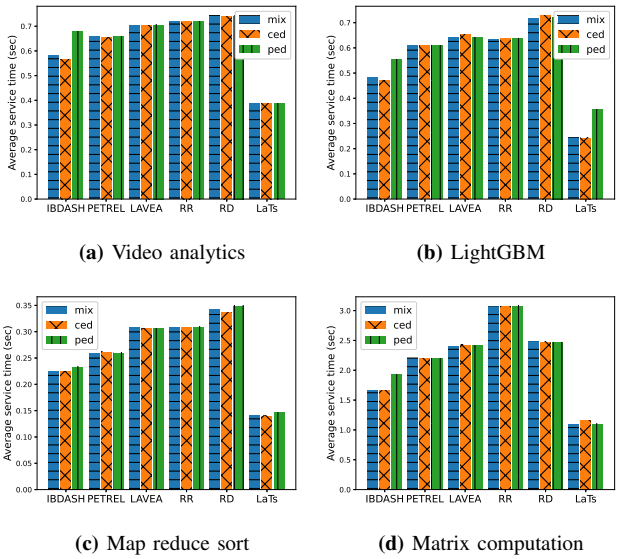
Table III shows different sets of λ values that have been used in our simulations. λ_1 simulates the scenario of a mixture of PEDs and CEDs. λ_2 represents the scenario when there are only CEDs present and λ_3 represents when there are only PEDs available. We specifically plot the set of λ_1 (mix of CEDS and PEDs) and the real-world data collected from the real-world participants in Figure 6b, and we see that the model for ED_6 fits the real-world data well which verifies our assumption that exponential function can be used as a good prediction for device availability with careful selection of failure rates λ.

F. Evaluation of End-to-End Latency

In our experiment, we repeated a 15s simulation cycle 20 times giving a total simulation time of 5 minutes. In each cycle, 1000 application instances arrive randomly clustered within the initial 1.5s and there are 100 edge devices uniformly distributed among the 8 device classes listed in appendix B Table V. Their corresponding interference coefficients are collected from real-world experiments. It can be observed from Figure 7 that the average service time of IBDASH outperforms other orchestration schemes except for LaTS under all three scenarios (CEDs, PEDs, the mix of CEDs and PEDs (50%:50%)) for all applications due to its awareness of the co-located task interference. The reason that LaTS outperforms IBDASH in execution latency comparison is that LaTS allocates the majority of tasks to a single powerful device. However, if that device were to become unavailable, then the performance of LaTS will suffer drastically.

G. Evaluation of Probability of Failure

Figure 8 demonstrates the average probability of failure of application instances under six different orchestration schemes for three different scenarios. We see that IBDASH outperforms other baselines under all three scenarios,

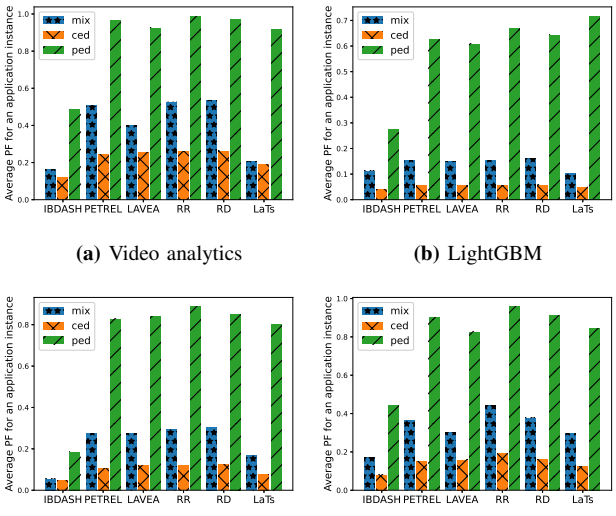


(a) Video analytics **(b)** LightGBM

(c) Map reduce sort **(d)** Matrix computation

Fig. 7: The average service time for all 4 testing applications under 6 different orchestration schemes. IBDASH **outperforms other schemes under all tests (except LaTS).**

especially in the scenario where edge devices are a combination of PEDS and CEDs or all are PEDs as IBDASH offers the redundant replication to reduce the probability of failure. IBDASH is better than LaTS by 29.7% for mix, 58.5% for PEDs, and 34% CEDs on average across four applications. We emphasize that LaTS outperforms IBDASH in rare cases since the majority of tasks are allocated to a single device. If that single device being allocated had a low probability of failure, the overall probability of failure value is low. However, as discussed earlier, this can lead to a catastrophic failure of all application instances.



(a) Video analytics **(b)** LightGBM

(c) Map reduce sort **(d)** Matrix computation

Fig. 8: Average probability of failure for 4 testing applications under 6 different orchestration schemes. IBDASH **outperforms other schemes especially when edge devices are PEDs or PEDs and CEDs combination.**

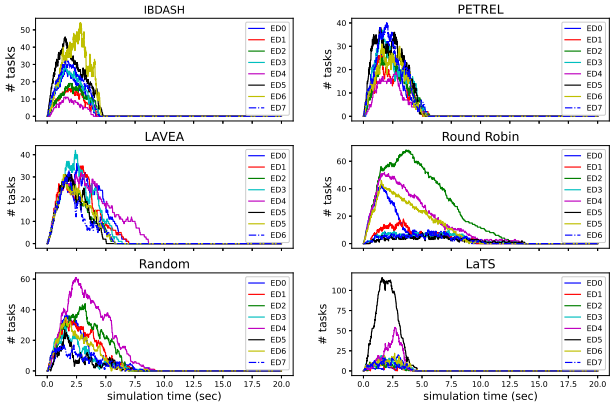


Fig. 9: Plot of load on each edge device for different orchestration schemes under mixed scenario (λ_1 in Table III). IBDASH provides an even load.

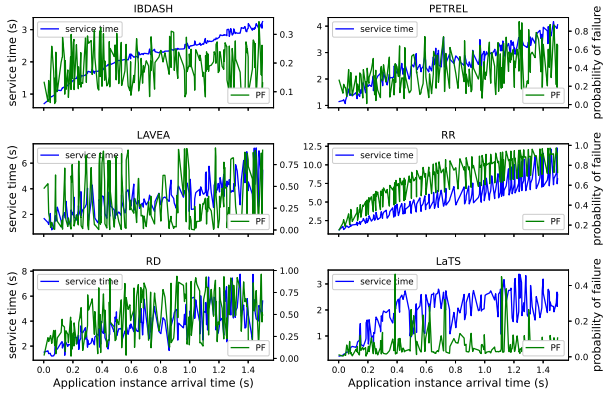


Fig. 10: Plot of service time and probability of failure of each individual instance for 200 application instances that arrive randomly in 1.5 second under mixed PED:CED scenario (λ_1 in Table III) for six different orchestration schemes.

H. Microscopic View

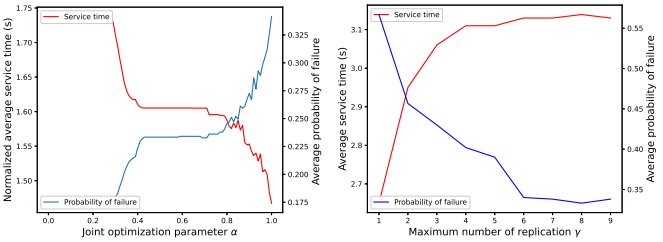
To show the advantage of using IBDASH in detail, we performed separate experiments (shown in Figure 9 and Figure 10) with 8 edge devices (one from each class) so that we can plot the loads, which is the number of tasks on each device and examine the load distribution. We zoom into one of the 15s simulation cycles for this experiment. We see that IBDASH tends to allocate more tasks on edge devices with low interference coefficients to reduce the overall service time, in this case, devices ED5 and ED6. On the other hand, LAVEA [6] chooses to allocate the task to the edge device with the least number of running tasks, which results in the load being fairly balanced on each edge device. Petrel [7] chooses two edge devices randomly and allocates the task to the one which has a lower expected service time. It results in a fairly balanced load distribution as well except for those edge devices with significantly larger interference coefficients compared to others. LaTS [9] allocates the majority of the tasks to ED5 due to its significant superiority of performance compared to others. Even though it does produce low execution latency, it results in a highly imbalanced allocation, which has negative consequences.

Both Round Robin and Random allocation result in task accumulation on edge devices with high interference coefficients due to their fixed task distribution scheme, which leads to a long service time.

From Figure 10, we see that when the probability of failure of edge devices increases toward the end of the simulation, our orchestrator IBDASH starts to replicate the tasks to reduce the probability of failure, which results in increasing the overall number of tasks on some edge devices (Figure 9) and correspondingly the average service time for the application instance goes up due to the redundant tasks. For Petrel [7], LAVEA [6], the average probability of failure is higher as they do not have the extra redundancy to reduce the probability of failure. As for LaTS [9], it assigned most tasks to a single device, which happens to have a low probability of failure, so it shows a fairly low probability of failure.

I. Evaluation of Joint Optimization of End-to-end Latency and Probability of Failure

To evaluate the joint optimization of our orchestration scheme, we performed a sweep of the replication threshold γ and (separately) of the joint optimization parameter α. The result is shown in Figure 11.



(a) Optimization parameter α sweep **(b)** Maximum replication degree γ

Fig. 11: a. Sweep of joint optimization parameter α from 0 to 1 in step of 0.01 @ $\beta = 0.1, \gamma = 3$ under λ_1 Table III. b. Sweep of replication degree γ @ $\beta = 0.1, \alpha = 0.5$ under λ_3 Table III

In Figure 11a, initially, when the probability of failure is assigned much more weight than the service time, the algorithm tends to optimize the probability of failure as much as it can until it meets the probability of failure threshold β or replication degree γ. At around $\alpha = 0.3$, IBDASH starts to prioritize optimizing service time as more weight is given to it and shorter service time gives a better joint optimization score. Throughout the sweep, we see the general trend is that as the normalized service time decreases, the average probability of failure increases. However, there are some fluctuations in the sweep. For example, at $\alpha \approx 0.8$, we see that there is a temporary drop in the probability of failure and an increase in the average service time. The reason for this is that as the α value changes in each sweep, the task allocation changes as well. Therefore, this change in task allocation can result in fluctuations in both probability of failure and service time, but the general trend is not affected as shown in Figure 11a.

From Figure 11b, we see that as we increase the replication degree, the average service time increases while the average probability of failure decreases, then it stays fairly stable after around 6 replications as IBDASH is able to determine that further replications will not result in better joint optimization, therefore it stops replicating.

VI. Related Work

Scheduling on the edge: One of the most important goals in edge computing is reducing the end-to-end latency to enable latency-sensitive applications for users. Several prior works such as LaTS [9], LAVEA [6], and Petrel [7] propose scheduling strategies that aim at minimizing the service time in a multi-edge collaborative environment. We have shown that IBDASH outperforms these schemes in terms of average service time and probability of failure in a heterogeneous edge computing setting. Other works consider joint optimization of low latency and other goals under some constraints [20, 24]. In particular, Ran et al. [20] focus on the latency and accuracy optimization for video analytics under the battery, network, and cost constraints. There is a growing body of work on low-latency scheduling on the edge [25, 26] and a subset considers DAG-based applications [27]. However, none of these works considers the interference among co-located tasks or intermittent availability of a subset of devices.

Interference-based scheduling: A few efforts study the availability of heterogeneous edge devices and interference among tasks on the same edge device [9, 28, 29]. LaTS [9] proposed to use a Latency-CPU usage model to address the interference among co-located tasks. It constantly monitors CPU usage on each edge device to schedule tasks to get the minimum predicted latency. Moreover, Aral et al. [29] proposed a score-based edge service scheduling algorithm that evaluates the network, compute, and reliability capabilities of edge nodes, but the drawback of such an algorithm is that it requires sharing of monitoring information across all devices which is infeasible in edge computing. Comparatively, IBDASH considers heterogeneity in devices and requires much less information sharing among the edge devices.

Edge device reliability and failure prevention: There is a significant amount of work on investigating the reliability of edge devices and failure prevention [30, 31, 32]. Inaltekin et al. [32] conducted a small-scale experiment to show the trade-offs between reliability and latency for edge nodes and server-less computing functions. Liu and Zhang [30] proposed three different algorithms for offloading that are based on a heuristic search to reduce the failure probability and latency, but it fails to address the interference of task co-location on the same edge device and the heterogeneity of edge devices. Park et al. [31] proposed a fault detection model based on the long short-term memory (LSTM) recurrent neural networks that are used in industrial robot manipulators. None of those frameworks addresses how to predict edge device availability. On the other hand, our work offers a model for edge device availability prediction and consequently guides the extra redundancy needed for application success.

VII. Discussion and Future Work

In this section, we present extensions of IBDASH that would need to be implemented to handle some use cases. *First*, the current algorithm checks each incoming task against *all* available edge devices. This procedure can result in high orchestration overhead for simple tasks when many edge devices are available. This problem can be addressed through edge device clustering based on (static) capabilities and (dynamic) load on each device. Then the computation overhead reduces from the number of devices to the number of clusters. Any of several existing techniques for edge device clustering can be used, such as [33]. Moreover, we expect that only those devices that are close to the user would be candidates for executing the user's tasks. Therefore, the total number of devices available for scheduling should fall into a reasonable range. *Second*, we hope that the execution latency of tasks within the same stage are fairly balanced. A long latency task in a stage can delay the execution of later stage tasks. The task execution latency balancing can be achieved through further task partitioning [34] and input load balancing [35]. *Third*, the linearity in the task interference plots may not hold if the number of tasks running on an edge device is large enough to cause a discontinuous change such as cache spillage. In that case, a higher-order characterization (say, quadratic or piece-wise linear) of the interference plots is needed to accurately predict the execution latency. Finally, we use exponential functions to predict the sporadic availability of edge devices. Even though we validated this assumption using real-world mobility data, this may not hold in certain scenarios (say a student class schedule changes from one module to the next). This can be improved by using the history of availability of each edge device and semi-Markov process to predict availability.

VIII. Conclusion and Takeaways

In this paper, we proposed a novel orchestration framework, IBDASH, that enables multi-stage applications to be executed on edge computing systems. Crucially IBDASH can incorporate personal edge devices (PEDs) along with commercial edge devices (CEDs) in executing the tasks that constitute the application. To support this, we make three novel contributions. First, IBDASH determines the dependency among different tasks within an application represented using a DAG. Second, IBDASH leverages PEDs while accounting for the possibility of resource contention and low and unpredictable availability of such devices. Third, IBDASH jointly minimizes the average application execution latency (via dynamic scheduling) and application failure likelihood (via task replication). We evaluated IBDASH with four applications that span various DAG structures, with unit measurements of real application tasks on real devices. We compared IBDASH with three state-of-the-art edge scheduling solutions, LAVEA, Petrel, and LaTS. We observe that IBDASH yields an average reduction of 14% on the service time of applications and reduces the average probability of failure for the applications by 41%.

There are three takeaways from our work that are of general applicability to edge computing systems. *First*, it is possible to leverage highly heterogeneous devices to compose a usable, i.e., low-latency and reliable, edge computing platform. *Second*, it is feasible to use unmanaged edge devices (called PEDs here) to create the usable edge computing platform, *if* these are combined with commercially managed devices. *Third*, a usable edge computing platform, unlike a cloud computing platform, must manage anticipated failures by proactively replicating tasks as these are far more likely than in the cloud computing world.

References

[1] Amazon: Lambda@edge. https://aws.amazon.com/lambda/edge/, 2020. Accessed: 2021-12-21.

[2] Microsoft: The future of computing: intelligent cloud and intelligent edge. https://azure.microsoft.com/en-us/overview/future-of-cloud/, 2020. Accessed: 2022-01-11.

[3] Google: Edge network. https://peering.google.com/, 2020. Accessed: 2021-12-21.

[4] Your phone is now more powerful than your pc. https://bit.ly/344UsiR, 2021. Accessed: 2022-01-11.

[5] Liang Wang, Mengyuan Li, Yinqian Zhang, Thomas Ristenpart, and Michael Swift. Peeking behind the curtains of serverless platforms. In *USENIX ATC*, pages 133–146, 2018.

[6] Shanhe Yi, Zijiang Hao, Qingyang Zhang, Quan Zhang, Weisong Shi, and Qun Li. Lavea: Latency-aware video analytics on edge computing platform. In *2nd ACM/IEEE Symposium on Edge Computing*, pages 1–13, 2017.

[7] L. Lin, P. Li, J. Xiong, and M. Lin. Distributed and application-aware task scheduling in edge-clouds. In *2018 14th International Conference on Mobile Ad-Hoc and Sensor Networks (MSN)*, pages 165–170, 2018.

[8] M. Mitzenmacher. The power of two choices in randomized load balancing. *IEEE Transactions on Parallel and Distributed Systems*, 12(10):1094–1104, 2001.

[9] Wuyang Zhang, Sugang Li, Luyang Liu, Zhenhua Jia, Yanyong Zhang, and Dipankar Raychaudhuri. Hetero-edge: Orchestration of real-time vision applications on heterogeneous edge clouds. In *IEEE INFOCOM 2019 - IEEE Conference on Computer Communications*, pages 1270–1278, 2019.

[10] Can Wang, Sheng Zhang, Yu Chen, Zhuzhong Qian, Jie Wu, and Mingjun Xiao. Joint configuration adaptation and bandwidth allocation for edge-based real-time video analytics. In *IEEE INFOCOM 2020 - IEEE Conference on Computer Communications*, pages 257–266, 2020.

[11] Luyang Liu, Hongyu Li, and Marco Gruteser. Edge assisted real-time object detection for mobile augmented reality. In *The 25th Annual International Conference on Mobile Computing and Networking*, 2019.

[12] Shikhar Suryavansh, Chandan Bothra, Kwang Taik Kim, Mung Chiang, Chunyi Peng, and Saurabh Bagchi. I-bot: Interference-based orchestration of tasks for dynamic unmanaged edge computing. *arXiv preprint arXiv:2011.05925*, 2020.

[13] Heng Zhang, Michael A Roth, Rajesh K. Panta, He Wang, and Saurabh Bagchi. Crowdbind: Fairness enhanced late binding task scheduling in mobile crowdsensing. *2020 International Conference on Embedded Wireless Systems and Networks*, page 61–72, 2020.

[14] Robert Hood, Haoqiang Jin, Piyush Mehrotra, Johnny Chang, Jahed Djomehri, Sharad Gavali, Dennis Jespersen, Kenichi Taylor, and Rupak Biswas. Performance impact of resource contention in multicore systems. In *2010 IEEE International Symposium on Parallel Distributed Processing (IPDPS)*, pages 1–12, 2010.

[15] AR. Arunarani, D. Manjula, and Vijayan Sugumaran. Task scheduling techniques in cloud computing: A literature survey. *Future Generation Computer Systems*, 91:407–415, 2019.

[16] Li Lin, Xiaofei Liao, Hai Jin, and Peng Li. Computation offloading toward edge computing. *Proceedings of IEEE*, 107(8):1584–1607, 2019.

[17] Byung-Gon Chun, Sunghwan Ihm, Petros Maniatis, Mayur Naik, and Ashwin Patti. Clonecloud: Elastic execution between mobile device and cloud. In *6th European Conference on Computer Systems (Eurosys)*, page 301–314, 2011.

[18] Eduardo Cuervo, Aruna Balasubramanian, Dae-ki Cho, Alec Wolman, Stefan Saroiu, Ranveer Chandra, and Paramvir Bahl. Maui: Making smartphones last longer with code offload. In *Proceedings of the 8th International Conference on Mobile Systems, Applications, and Services*, page 49–62, 2010.

[19] Zhiyuan Li, Cheng Wang, and Rong Xu. Computation offloading to save energy on handheld devices: A partition scheme. In *CASES*, page 238–246, 2001.

[20] Xukan Ran, Haolianz Chen, Xiaodan Zhu, Zhenming Liu, and Jiasi Chen. Deepdecision: A mobile deep learning framework for edge video analytics. *IEEE Conference on Computer Communications (INFOCOM)*, pages 1421–1429, 2018.

[21] Elie El Haber, Tri Minh Nguyen, and Chadi Assi. Joint optimization of computational cost and devices energy for task offloading in multi-tier edge-clouds. *IEEE Transactions on Communications*, 67(5):3407–3421, 2019.

[22] Chao Zhu, Giancarlo Pastor, Yu Xiao, Yong Li, and Antti Ylae-Jaeaeski. Fog following me: Latency and quality balanced task allocation in vehicular fog computing. In *2018 15th Annual IEEE International Conference on Sensing, Communication, and Networking (SECON)*, pages 1–9, 2018.

[23] Meng Qin, Nan Cheng, Zewei Jing, Tingting Yang, Wenchao Xu, Qinghai Yang, and Ramesh R. Rao. Service-oriented energy-latency tradeoff for iot task partial offloading in mec-enhanced multi-rat networks. *IEEE Internet of Things Journal*, 8(3):1896–1907, 2021.

[24] Yuvraj Sahni, Jiannong Cao, and Lei Yang. Data-aware task allocation for achieving low latency in collaborative edge computing. *IEEE Internet of Things Journal*, 6(2): 3512–3524, 2019.

[25] Daniel Zhang, Nathan Vance, Yang Zhang, Md Tahmid Rashid, and Dong Wang. Edgebatch: Towards ai-empowered optimal task batching in intelligent edge systems. In *Real-Time Systems Symposium (RTSS)*, pages 366–379, 2019.

[26] Ting He, Hana Khamfroush, Shiqiang Wang, Tom La Porta, and Sebastian Stein. It's hard to share: Joint service placement and request scheduling in edge clouds with sharable and non-sharable resources. In *2018 IEEE 38th International Conference on Distributed Computing*

Systems (ICDCS), pages 365–375. IEEE, 2018.

[27] Hanlong Liao, Xinyi Li, Deke Guo, Wenjie Kang, and Jiangfan Li. Dependency-aware application assigning and scheduling in edge computing. *IEEE Internet of Things Journal*, 2021.

[28] Ron C. Chiang and H. Howie Huang. Tracon: Interference-aware scheduling for data-intensive applications in virtualized environments. In *Proceedings of 2011 International Conference for High Performance Computing, Networking, Storage and Analysis*, 2011.

[29] Atakan Aral, Ivona Brandic, Rafael Brundo Uriarte, Rocco De Nicola, and Vincenzo Scoca. Addressing application latency requirements through edge scheduling. *Journal of Grid Computing*, 17, 12 2019.

[30] Jianhui Liu and Qi Zhang. Offloading schemes in mobile edge computing for ultra-reliable low latency communications. *IEEE Access*, 6:12825–12837, 2018.

[31] Donghyun Park, Seulgi Kim, Yelin An, and Jae-Yoon Jung. Lired: A light-weight real-time fault detection system for edge computing using lstm recurrent neural networks. *Sensors*, 18(7), 2018.

[32] Hazer Inaltekin, Maria Gorlatova, and Mung Chiang. Virtualized control over fog: Interplay between reliability and latency. *IEEE Internet of Things Journal*, 5(6): 5030–5045, 2018.

[33] A. Asensio, X. Masip-Bruin, R.J. Durán, I. de Miguel, G. Ren, S. Daijavad, and A. Jukan. Designing an efficient clustering strategy for combined fog-to-cloud scenarios. *Future Generation Computer Systems*, page 392–406, 2020.

[34] Jiawen Hu, Miao Jiang, Qi Zhang, Quanzhong Li, and Jiayin Qin. Joint optimization of uav position, time slot allocation, and computation task partition in multiuser aerial mobile-edge computing systems. *IEEE Transactions on Vehicular Technology*, 68(7):7231–7235, 2019.

[35] Yanfang Le, Jiangchuan Liu, Funda Ergün, and Dan Wang. Online load balancing for mapreduce with skewed data input. In *IEEE INFOCOM 2014 - IEEE Conference on Computer Communications*, pages 2004–2012, 2014.

APPENDIX

A. Independent and Additive verification

t_1: computational intensive task
t_2: memory intensive task
t_3: Network/IO tasks

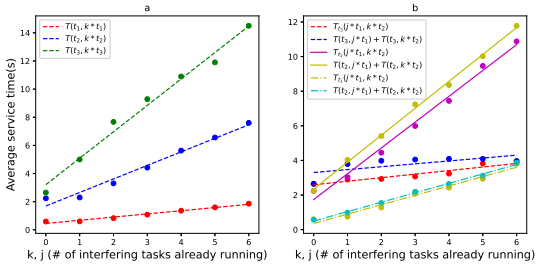


Fig. 12: Interference pattern verification on ARM-based architecture with Linux OS, j,k are the number of types of tasks running on the ED.

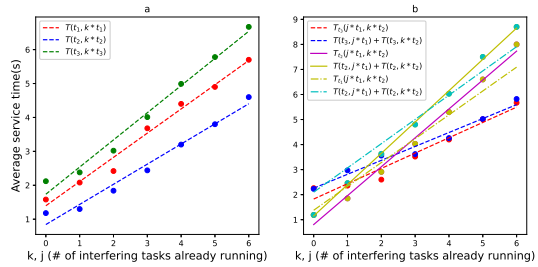


Fig. 13: Interference pattern verification on x86 architecture with macOS, j,k are the number of tasks of t_1 and t_2 running on the ED.

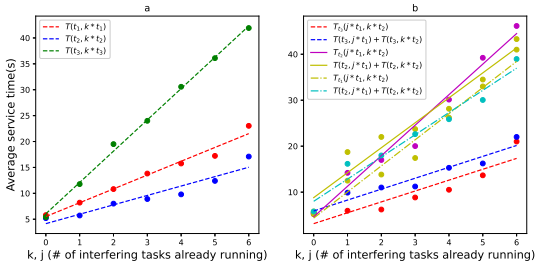


Fig. 14: Interference pattern verification on ARM-based architecture with AndroidOS, j,k are the number of tasks of t_1 and t_2 running on the ED.

TABLE IV: Verification assessment of valuation of the independent and additive interference pattern.

	x86-macOS	ARM-AndroidOS	ARM-Linux
com (r^2)	0.983	0.974	0.975
mem (r^2)	0.976	0.909	0.978
net (r^2)	0.983	0.997	0.985
com-mem-net (error %)	9.867	22.840	19.257
com-mem-mem (error %)	16.653	18.596	14.654
com-mem-com (error %)	21.261	16.335	14.522

com: computation intensive task
mem: memory intensive task
net: network/IO request intensive task
x-y-z: x, y are types of tasks already running on the device, z is the new incoming task

Table IV shows the verification of the independent and additive patterns. The first three rows show that for the same type of tasks, the execution time and the number of tasks are linearly related and the corresponding R-square values show that such a regression model is fairly accurate. The last three rows show that the execution time of a task under the interference of other types of tasks which are calculated as a sum of the execution time of that task under the interference of each individual task within an acceptable error range.

B. Edge device profiling

To test IBDASH's joint optimization of end-to-end latency and probability of failure under edge devices with various capabilities, we profiled four applications that span different application domains against seven EC2 instances and a MacBook Pro. Since the profiling only needs to be performed once for each task on each device, the time used for device profiling is not a concern for IBDASH.

TABLE V: Edge device configuration

ED id	Instance type	(v)CPUs	Memory(GB)	Frequency(GHz)
ED0	Macbook Pro 2017	2	8	3.1
ED1	t2.xlarge	4	16	2.3
ED2	t2.2xlarge	8	32	2.3
ED3	t3.xlarge	4	16	2.5
ED4	t3a.xlarge	4	16	2.2
ED5	c5.2xlarge	8	16	3.4
ED6	c5.4xlarge	16	32	3.4
ED7	t3.2xlarge	8	32	2.5

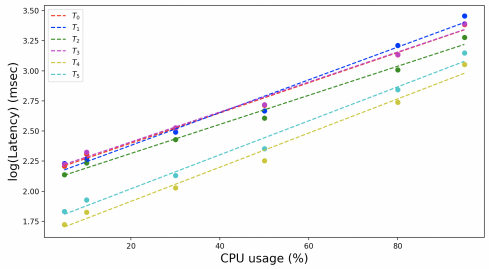


Fig. 15: Profiling data on ED1 (t2.xlarge) shows the linear relationship between log(latency) and the cpu usage.

Table V shows the 7 different AWS EC2 instances and the Macbook Pro we profiled against the four applications used in our simulation. Note that different configurations emulate different compute power, memory, and number of CPUs, which result in different interference patterns for each device.

Every pair of m, c values (task interference parameter) between every two tasks are profiled on each of the eight edge devices. In addition, LaTS [9] makes an assumption that there is a parametric model that captures the relationship between latency and CPU usage. For our baseline experiments with LaTS, we have to determine this parametric relationship in our experimental setting. Therefore, we also profiled the relationship between the CPU usage and task execution latency and fitted a linear regression model to capture such a relationship. Figure 15 shows the profiling data collected on the t2.xlarge EC2 instance at 5 different CPU usage levels for different task types. From this result, we conclude that there is a linear relationship between the log(latency) and the CPU usage. We use this linear relationship for latency estimation for our experiments with the baseline LaTS.

C. Scalability Verification

Figures 16-18 show the scalability of IBDASH.

For the result in Figure 16, we performed orchestration for 30 lightgbm application instances. As we include more edge devices in the network, we see that the average service time for each application instance drops drastically initially and gradually becomes stable. On the other hand, the orchestration time slowly increases as more devices need to be considered for task allocation.

For the result in Figure 17, we performed orchestration for application instances that contain up to 500 tasks on 8 simulated devices. As more tasks need to be orchestrated for each application instance, we see that the average orchestration time for each application instance gradually increases, following a linear trend. However, since the number of devices is fixed, more tasks will be allocated to each edge device and the service time for each application instance increases due to the interference.

To further verify our orchestration complexity, we performed task orchestration, task distribution, and result gathering on 8 real devices, and the corresponding result is shown in Figure 18.

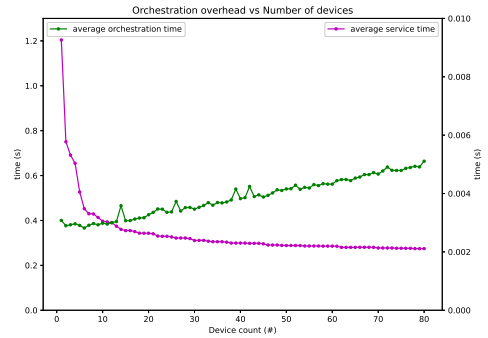


Fig. 16: Average orchestration overhead and average service time for lightgbm. The average service time for an application instance drops drastically initially when more edge devices are included in the network, then it slowly decreases and tends to be flat eventually. The orchestration overhead linearly increases as the number of devices included in the network increases.

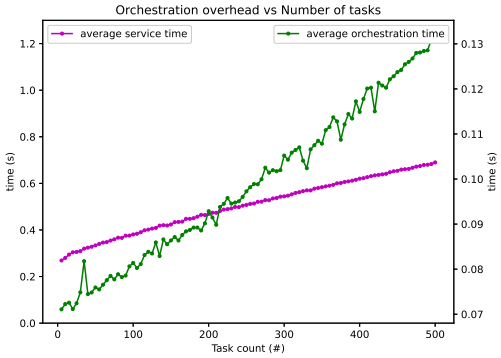


Fig. 17: Average orchestration overhead and average service time for lightgbm application with 8 simulated devices. The average service time for an application instance gradually increases as the number of tasks running on each device is increasing. The orchestration overhead linearly increases as the number of tasks to orchestrate increases.

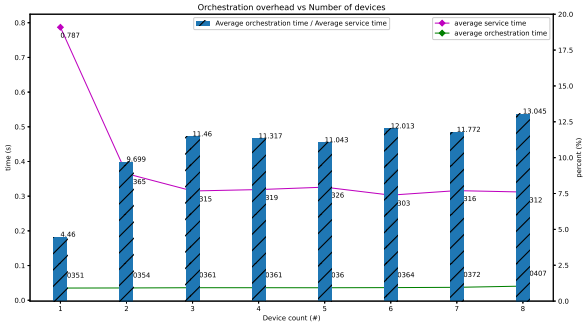


Fig. 18: Average orchestration overhead and average service time for lightgbm application on 8 real devices. The orchestration overhead is around 10% of the average service time of an application instance.