Mapping IoT Applications on the Edge to Cloud Continuum with a Filter Stream Model

Shuangsheng Lou
Computer Science and Engineering
The Ohio State University
lou.125@osu.edu

Gagan Agrawal
Computer and Cyber Sciences
Augusta University
gagrawal@augusta.edu

***Abstract*—In recent years, there has been considerable interest in developing streaming applications for IoT (or Edge Computing) environments. In this context, several studies have (manually) deployed application components on different nodes in the path from the *extreme edge* to the *cloud*. It is clearly desirable to automate this mapping process. However, when considering this problem in the context of heterogeneous multi-layer wireless networks, we see challenges like limited computing and battery power at the extreme edge, modest transmission bandwidth, and different processing powers for different nodes. Automatic deployment or partitioning for streaming applications considering these challenges has not been addressed in the previous work.**

In this paper, a framework for automated deployment is presented with an emphasis on optimizing latency in the presence of resource constraints. A dynamic programming based deployment algorithm is developed to make deployment decisions. With battery power being a key constraint, a major component of our work is a power model to help assess the power consumption of the edge devices at the runtime. Using three applications, we show the large reductions in both power consumption and response latency with our framework, as compared to a baseline involving cloud-only execution.

I. Introduction

Internet of Things (IoT) is a recent paradigm that is gaining popularity because of its applicability in everyday scenarios, such as transportation and logistics, healthcare, smart environments, social domain, and many others [21], [2]. As more connected devices are appearing between the edge and the cloud, *multi-layer wireless networks* are becoming a common topology. In deploying applications on such a topology, the computation, data and communication have to be orchestrated along the path from IoT devices to the cloud [20], [23], which involves new research challenges. This is also related to the emerging inter-related paradigms of Edge, Fog, and Mist computing [6], [16].

In application development and execution using any of the above paradigms, a significant goal is reducing latency time by using edge devices [20]. Several application studies and programming frameworks have considered this problem[4], [22], [13]. For example, the work in [9] proposes a method where the user can manually allocate data to be processed on the edge or cloud. However, besides such decision making not being automated, the work only focuses on data and not the application services. An increasing number of layers and heterogeneity can also complicate the work (or data) distribution problem, which has also not been considered. Thus, in this paper, we focus on the problem of automatic (and dynamic) partitioning of the computation stages along the path from the edge to a central (cloud) device.

We now elaborate on some of the challenges in addressing this problem. On one hand, the resource constraints on local devices, including limited computing and battery power, restrict the services that can be performed on them. On the other hand, for many streaming applications that are relevant to such scenarios, an important optimization metric is the application latency. Deployment decisions clearly impact application latency – for example, the time required for the execution of a service on an edge device is typically larger than the time in the cloud due to the difference in computing power. While moving a part of services to (extreme) edge devices, ideally the computation scheduled at edge devices should lead to a reduction in data to be transmitted to the next level. This is because data transmission can be a significant factor in power consumption and because wireless bandwidth can be limited and communication can be unreliable. Because many resources might be shared (multi-tenancy) and both data arrival rate and application requirements can vary over time (e.g. based on an interesting event that may be detected), the decision-making should be dynamic. Finally, a software framework is needed where nodes with different processing capabilities can be organized and managed. This framework should also allow users to decompose their application to a series of computational stages that can be executed at different locations. At the same time, we need to be able to assess resource usage of nodes at a different layer at the runtime, including power consumption at the (extreme) edge and the overall latency.

This paper addresses the above problems and makes the following contributions. We build on the notion of actors [15] for application development, composing a streaming application with units that can be flexibly placed and easily migrated. Within this context, a new framework, Actor Deployment Framework (ADF), is proposed whose components include *service management*, *deployment management* and *variation management* – together they help deploy the computations automatically and dynamically. Next, we model the deployment problem as a Dynamic Programming problem and a cost-based deployment algorithm is developed that considers different resource constraints. In addition, a power model for edge devices is built to support our deployment algorithm. Four streaming applications, including one with dynamically changing data characteristics, arising in the context of Edge/Fog

Computing are implemented using our framework. Evaluation of the placement decisions shows the effectiveness of our approach. As compared to a baseline cloud-only deployment, reductions of power consumption have been achieved ranging from 52% to 92% and the largest reduction of response latency is up to 95%. At the same time, for the dynamic application, we show that our placements are stable when there are short-term changes in the load, but our framework can also change placements given long-term shift in the load.

II. System Design

In this section we propose an integrated framework called Actor Deployment Framework (ADF) for multi-level wireless networks, which is designed with the following goals: 1) Providing ease of connecting IoT devices with the cloud (or other connected devices), and facilitating the processing of streaming data over such connectivity, 2) Providing an API where streaming applications can be decomposed into a set of *actors*, 3) Supporting models for execution time (latency) and power consumption, 4) Mapping and executing the actors over edge and other devices using the above models, and 5) Dynamically modifying the mapping of actors to devices in view of factors like a change in the workload or network congestion.

Overall, ADF can facilitate actors to be deployed to process data concurrently while reducing the latency of the application and network bandwidth consumption. Since application characteristics and/or processing environment can vary dynamically over time (e.g., an interesting event may be detected, or there may be a change in the workload or network bandwidth), ADF provides an interface called *variation management* to update deployment configuration at the runtime.

A. Actors

In this paper, we follow the idea of an *actor* as used in Calvin, which is a hybrid IoT framework for application development, deployment, and execution [15], [1]. Actors are the building blocks for an application, with the following characteristics: 1) an actor represents a computation or a service that can only communicate with others by passing data messages over ports, 2) actors can only be triggered by streaming blocks of data received from the input port or event notification of external activity (for example, time release, or availability of data), and 3) each actor is self-contained and can hide its internal state from the rest of the application. We depict the idea of an actor through Figure 1.

The features of the actors ensure that communication is the only way to influence the internal state of actors. This, in turn, makes it easier to migrate an actor from one device to another device. If an actor needs to be migrated, we need to serialize the internal state and send it to the destination runtime. A new instance of an actor will be created by restoring the state from the serialized state data. The cost of migration (that means processing time needed) depends mostly on the size of the internal state, along with the bandwidth available for the transport.

To decompose an application to actors, a developer describes each actor by defining their input and output, the inner actions that connect the input and output, and its internal state. ADF specifically targets streaming applications that can be decomposed into a pipeline of actors. We believe that most

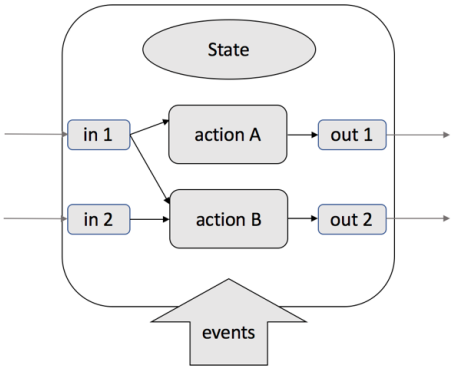

Fig. 1: An Actor (as defined in Calvin project [15])

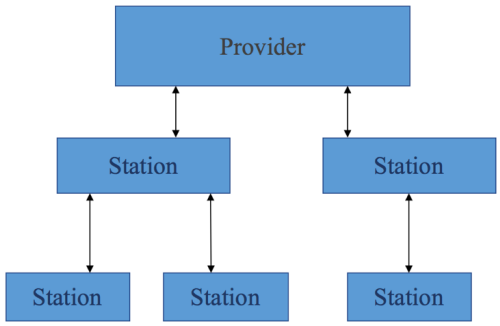

Fig. 2: Architecture Targeted by ADF

streaming applications of interest do fall within this category – besides the applications we have experimented within this paper, other traffic situation analysis and real-time gesture detection applications are also suitable.

B. ADF Architecture

From the view-point of ADF, the system is comprised of two types of components, the *stations* and the *provider(s)*. A station is any device that produces the data, such as sensors and mobile devices, or that processes the data by executing actors, such as IoT gateways (e.g. a Raspberry Pi or Arduino). A provider is a powerful device that serves as a backend. A station may be connected to one or more stations, and a subset of stations are connected to the provider. An example of the architecture targeted by ADF is shown in Figure 2. Every station has its unique identity and stores its connection information, including connected station identifiers and the port(s) for communications. An ADF provider receives and stores the entire topology information at the beginning by receiving bottom-up connection information from all stations. There are two kinds of modes in our system: *Deploy Mode* and *Running Mode*. Deploy Mode is triggered either by variation management module in the Running Mode or external events. When the deploy mode is triggered, ADF tries to find optimal placement for all actors. After the Deploy Mode, all stations and the provider are assigned different actors and the system goes into the Running Mode.

Deploy Mode is consisted of service management and deployment management. The goal of service management is

to profile and manage actors on stations and the provider. On each station or the provider, one or more actors are deployed. Due to the characteristics of actors discussed above, the output data of the station is just the output of the last actor in the pipeline deployed and the state of the station is the combination of states of actors deployed locally. All stations and the provider keep the profile of the application in their local *Service Profile*. This includes the code of all actors and their order in the pipeline. This profile helps each station and the provider easily execute their assigned actors by picking the actors and updating their internal states. Another important component within service management is *Performance Profile*. It aims to analyze different local cost for each actor in the current station or provider and logs the performance analysis into performance profile locally. Specifically, one important performance factor is the response latency of the application, which is decided by both the execution time of each actor and the transmission time between station-to-station or station-to-provider. Because each actor is self-contained without shared state, its local execution time, which influences the overall latency, is a function of the local environment (e.g. CPU, memory and network bandwidth) and the workload. Here we aim at the actors whose pattern of workload is regular so that its local execution time can be measured and predicted. So in order to measure the effect of local environment on actors, each station and the provider execute all actors locally and log the execution time analysis in local *performance profile*. Besides, the network bandwidth impacts the transmission time between station-to-station or station-to-provider, which is reflected by the predicted transmission time of each actor. Thus, in summary, within the service management module, the impact of different local environments on all actors is reflected by their local execution and transmission time, and this information is logged as the performance profile. Also, another important performance factor logged is the power consumption in the (extreme) edge device, which is described in detail in Section 4.

When each station finishes its service management tasks, deployment management starts to work to obtain the optimal decision on the deployment of all actors and dispatch this decision to all stations. At first a local job is created for profile integration. It first waits to receive the profiles through the input port from all connected stations in the previous layers, and then sends these with its local performance profile to the next station. All stations will wait for the decisions from the provider. When the provider collects all performance profiles from its connected stations, a cost-optimized deployment algorithm that considers the power consumption of edge devices and constrained resources of each station will be triggered. The goal here is to yield the optimal decision on the deployment of all actors. After the deployment algorithm is executed, the resulting decision will be dispatched immediately to every station through the same path that every station used to send their performance profiles. The details of our deployment algorithm will be discussed in Section 3.

Through deployment management, every station receives a decision about its assigned actors, which is updated in its performance profile. Receiving this decision triggers the Running Mode of the application. In the Running Mode, all stations and the provider execute their assigned actors simultaneously for the streaming data. In order to support dynamic decision making, a variation management module is designed in our system to manage dynamic situations, such as a change of workload or network congestion. The change of workload for an actor will result in a change in its processing time. Similarly, network congestion can reduce network bandwidth, which will increase the transmission time. In both situations, the latency time of certain actors is changed and the deployment decision made earlier may not be optimal under the new situation. To detect these changes, a local job is created by the detection mechanism to periodically measure the performance of all actors like the way we did in the service management. These measurements are compared against those stored in the performance profile. Any variation not within the specified threshold will trigger the deployment management to make a new decision.

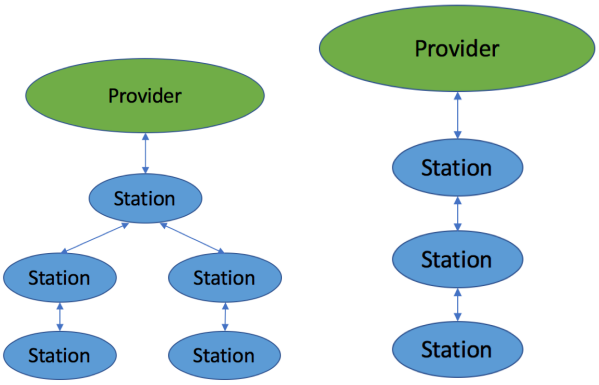

Fig. 3: Examples of Network Topology

Notation	Description
$Energy$	The energy consumption of the stations
$Latency$	End-to-end delay for every batch of input data
P	Parameter used to scale the $Latency$
$W1$	The weight of $Energy$ cost
$W2$	The weight of $Latency$ cost
W	The window size of measurement
$LocalTime_j$	Total local time at the jth station
$T_{i,j}$	Processing time of the ith actor at jth station
V_i	Predicted output size of the ith actor
$Bandwidth_j$	Forwarding bandwidth from jth station
$Tran_{i,j}$	Transmission time of sending V_i data at jth station
$Cont_j$	constant transmission loss at jth station
P_i	Power consumption rate of ith actor on edge devices
P'	Power consumption rate of uploading

TABLE I: Notation

III. Deployment Algorithm

Since the application is decomposed into a series of independent actors, the goal of the deployment algorithm is to specify where each actor should execute. Clearly, given a pipeline, a station can only be assigned one or several contiguous actors. No matter what the network topology is (e.g. see Figure 3), the goal of deployment algorithm reduces to finding a fixed number of partitions in a given pipeline, subject to maximizing a certain objective. Here, the number of partitions is equal to the number of hops from the edge device to the cloud. For example, the number of partitions is four in both network topology shown in Figure 3.

A. Cost Function

Table I gives the set of notations used to describe the cost function. As stated above, the partitions to be placed are subject to maximizing (or minimizing) an objective function. We now develop a cost function we use in this work. This cost function has two components. Most IoT applications are time-sensitive and thus response latency is our first factor. The power consumption in the (extreme) edge devices is also an important factor, especially for those edge devices with limited battery power. Thus, our goal becomes optimizing an objective function that considers both the latency of the application and the power consumption in the (extreme) edge devices. Formally,

$$Cost = \frac{\sum_{d=1}^{W}(Energy_d \times W1 + Latency_d \times P \times W2)}{W} \tag{1}$$

Here $Energy$ refers to the energy consumption of edge devices and $Latency$ is the overall end-to-end delay (across all services) for every batch of input data. $W1$ and $W2$ are the weights given to energy and latency factors ($W1 + W2 = 1$). P is a parameter used to scale the latency term so that the impact of latency and energy consumption can be combined as a whole, and W is the size of the window over which measurements are taken and the decision is made.

A *non-overlapping* window is applied here to make the optimization decisions both relatively stable and but yet sensitive to a long term change in the characteristics of the input data. In many practical situations, the characteristics of data can vary over a short duration, while the long term distribution stays the same. Because we average over a window, we can still get a stable decision as long as variations are short term and within the threshold.

1) Latency : The $Latency$ in Eq. 1 is determined by two important factors: the processing time on the stations and the provider, and the transmission time between one station to either a station or a provider. Both factors are impacted by the partitioning we perform. To better explain and compute the overall latency time $Latency$, an important concept called $LocalTime$ is proposed. Each station or provider j has its own $LocalTime_j$, which is the sum of the time spent on processing the actors deployed on it and the transmission time to the next layer. In certain cases the processing time is 0 if there is no actor deployed in this device. Also, the transmission time can be 0, such as for the provider.

Let the number of layers in our network topology be n and the number of actors needed to be deployed be m. There should be $n-1$ partitioning points that are expressed by w_1, w_2, ..., w_{n-1} – these are the last actors placed on their respective deployment place when deploying all actors into n deployment places. Now, for the jth station, $T_{i,j}$ is the processing time of the ith actor if executed on the jth station, and $Tran_{i,j}$ is the predicted transmission time if the ith actor is the last actor on this station. Then we have

$$LocalTime_j = \begin{cases} \sum_{i=1}^{w_1} T_{i,1} + Tran_{w_1,1}, & \text{if } j = 1 \\ \sum_{i=w_{n-1}+1}^{m} T_{i,n} & \text{if } j = n \\ \sum_{i=w_{j-1}+1}^{w_j} T_{i,j} + Tran_{w_j,j}, & \text{if } 1 < j < n \end{cases} \tag{2}$$

The above expression for $LocalTime_j$ considers three cases, which correspond to three possible deployment places. When j is equal to 1, it corresponds to stations in the first layer. If j is equal to n, it represents the provider. And the third case represents all other cases, i.e., the middle stations.

To compute the point-to-point transmission time, if V_i represents the predicted output size of ith actor and $Bandwidth_j$ denotes the uploading bandwidth at jth-layer station,

$$Tran_{i,j} = \frac{V_i}{Bandwidth_j} + Cont_j \tag{3}$$

The overall latency time $Latency$ of the streaming application is determined by the maximum $LocalTime$ among all stations and the provider since the processing is done in a pipelined fashion. Thus,

$$Latency = \max_{1 \leqslant j \leqslant n} LocalTime_j \tag{4}$$

2) Energy Consumption: The $Energy$ in Eq. 1 is determined by two parts: power consumption rate and the time over which the processing and data transmission occurs. Here we only focus on the energy consumption of the stations in the first layer, which are usually geographically distributed and not connected to a power supply. The method can easily extended to other cases by summing up the energy consumption of all layers. Assume our deployment plan is (w_1, w_2, ..., w_{n-1}), which denote $n-1$ partitioning points. So the energy consumption of executing actors from 1 to w_1 and the energy consumption while transmitting data after the actor w_1 are what needs to be factored in. Thus, we can calculate:

$$Energy = \sum_{i=1}^{w_1} P_i \times T_{i,1} + P' \times Tran_{w_1,1} \tag{5}$$

where P_i and P' represent power consumption rate of ith actor on the edge device and the transmission cost of uploading using the available wireless network, respectively. We will further discuss the power model we use in the next section.

B. Algorithm

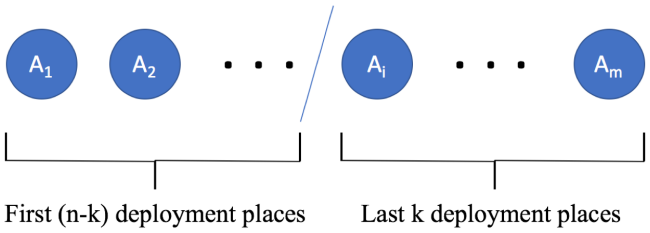

Fig. 4: Deployment Plan

Based on our proposed cost function, the deployment problem is an optimization problem that aims to find a sequence of (w_1, w_2,..., w_{n-1}) so that the minimum cost can be achieved. This optimization problem can be solved by *forward dynamic programming*.

Let $Cost(i,k)$ denote the minimum cost of deploying the actors $(A_i, A_{i+1}, \ldots, A_m)$ in the last k deployment places, including $k-1$ stations and one provider, as shown in Figure

4. Let $Tcost(i,j,k)$ denote the $LocalTime$ cost in the kth deployment place when deploying the actors $(A_i, A_{i+1}, \ldots, A_j)$ on. Let $Ecost(j)$ denote the energy consumption from actors $(A_1, A_2, \ldots, A_j)$ that are deployed into the extreme device (the only device on which power consumption is considered). Then, we have,

$$Tcost(i,j,k) = \begin{cases} (\sum_{i=i}^{j} T_{i,k} + Tran_{j,k}) \times P \times W2, & \text{if } i \leqslant j \\ Tran_{i-1,k} \times P \times W2, & \text{if } i > j \end{cases} \tag{6}$$

$$Ecost(j) = (\sum_{i=1}^{j} P_i \times T_{i,1} + P' \times Tran_{j,1}) \times W1 \tag{7}$$

Eq. 6 and Eq. 7 show the calculation of $Tcost(i,j,k)$ and $Ecost(j)$. For $Tcost(i,j,k)$, if i is larger than j, which means no actor is assigned to the kth deployment place, the kth-layer station will directly send the received output from previous layer to the next station and its $LocalTime$ cost only includes the transmission cost. Otherwise it includes both the processing and transmission cost.

Overall, this optimization problem can be decomposed into the sub-problem of deploying the rest of actors in the last $k-1$ deployment places, after determining the placement of actors in the $(n-k+1)$th place. Now, an important observation is that we only consider the power consumption of extreme edge devices, which are located in the first layer of our network topology. Thus, for $1 < k < n$, the cost is determined by the maximum $LocalTime$ value among all possible deployment places. When $k = n$, the extra energy consumption in the extreme devices should be added together with the $LocalTime$ cost when reducing to the $(k-1)$ problem. If k is equal to 1, which means all remaining actors are assigned to the provider, the cost is equal to $LocalTime$ cost in the provider. Based on this discussion, the term $Cost(i,k)$ can be described by Eq. 8 when $1 < k < n$, by Eq. 9 when k is equal to n, and finally, by Eq. 10 when k is equal to 1.

(1) Case 1 $(1 < k < n)$:

$$Cost(i,k) = \min_{i \leqslant j \leqslant m+1} \max \begin{cases} Cost(j, k-1) \\ Tcost(i, j-1, n-k+1) \end{cases} \tag{8}$$

(2) Case 2 $(k = n)$:

$$Cost(i,k) = \min_{i \leqslant j \leqslant m+1} \max \begin{cases} Cost(j, k-1) + Ecost(j-1) \\ Tcost(i, j-1, 1) + Ecost(j-1) \end{cases} \tag{9}$$

(3) Case 3 $(k = 1)$:

$$Cost(i,k) = Tcost(i,m,n) \tag{10}$$

Our final goal is to compute the value of $Cost(1,n)$ that denotes the minimum cost of deploying all actors in n deployment places.

IV. Modeling Power

To accurately make actors placement decisions, an estimate of power consumption rate of each actor on the first layer or edge devices is required (as was previously shown in Eq. 5). Because measuring the power consumption for edge devices in real time is not practical, accurate power models are necessary. With the hypothesis that an accurate estimation of the Raspberry Pi's power consumption rate can be obtained based on CPU and network utilization only (previously confirmed by Kaup *et al.*'s experiments [8]), we have developed a model that we will describe now.

To help build a power model, the power consumption is measured offline by interrupting the power lines of the USB connection and inserting a measurement *shunt*, which is a resistor in the 5 V power supply line. With a series connection between the resistor and Raspberry Pi, the currents flowing through them are the same. This leads to the measurement of two voltages U_1 and U_2 by an analog-to-digital converter (ADC), which are the voltage of the resistor and the voltage of the Raspberry Pi, respectively.

Now, if R_1 represents the resistance of the resistor, the power consumption of Raspberry Pi is easily calculated as $Power_{pi} = \dfrac{U_1 \times U_2}{R_1}$.

To build the power model as a function of CPU utilization, a configurable CPU load is needed. For this purpose, a load generator and a CPU utilization limiter are used. The desired load is generated by running a parallel infinite loop using *OpenMP*. A tool called *cpulimit* is used to limit the CPU utilization – it is assumed that other processes except the application process have little influence on the CPU utilization.

In our implementation, to avoid the influence of connected USB devices, all connected USB devices of Raspberry Pi including the mouse and the keyboard were disconnected during the power measurement. In addition, to consider the influence brought by the WLAN connection in the applications, the power measurement was conducted in a WLAN environment by connecting to the router. A 100 mΩ resistor R_1 is used and *ADS1115* is used as the ADC to measure the voltages of Raspberry Pi and resistor through separate Inter-Integrated Circuit (I2C) addresses. The default sampling rate of 128 samples per second (SPS) of *ADS1115* was chosen in the measurement. Each test for different operating points of the CPU utilization runs for 10 seconds, which generates a total of 1280 samples. We run 40 tests varying the load in the range from 2.5% to 100% (steps of 2.5%), and the relationship between CPU load and power consumption is established. By a standard linear regression step, the following power model is generated (u is the CPU utilization in the range 0 to 1 and the measurements are in mW).

$$Power_{CPU} = 1142 + 2526 \times u \tag{11}$$

V. Experiments and Results

This section reports results from experiments conducted to evaluate the effectiveness of our framework. We first applied our deployment algorithm to three applications and the results of best placement produced by our algorithm are compared against the performance of a *cloud-execution* baseline, where the edge device directly pushes input data to a cloud node where all processing takes place. This comparison used both application latency and power consumption as metrics.

A. Experimental Settings

In our experimental settings, a three-layer network is built for our applications. Two Raspberry Pis play the role of station and are connected by WLAN (the upload bandwidth is 32Mbps). A desktop computer (MacBook Pro) plays the role of a provider in our system and is connected to one station by

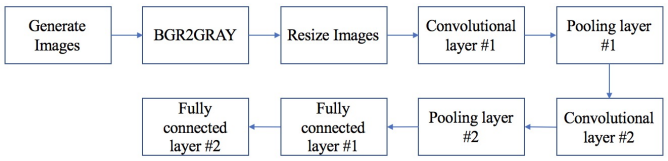

Fig. 5: CNN actors

LAN with 94 Mbps upload bandwidth. As a simple baseline, we have a setup where the edge device pushes all input data to the provider directly and the provider performs all processing tasks. The streaming images are generated and captured in real time from the camera of Raspberry Pi in the first layer for our CNN and FLD applications with a frame rate of 30 FPS and resolution of $640 \times 480 \times 3$. All measurements were conducted in a fixed environment, thus we assume that the network bandwidth stays constant and the power consumption rate during uploading (P' in Eq. 5) is also stable at 1860 mW on average. We set both $W1$ and $W2$ to 0.5 and set P in Eq. 1 to the idle power consumption rate of Raspberry Pi, which is 1294 mW.

B. CNN Application

Convolution Neural Network (CNN) has been applied in many scenarios related to object recognition, such as ImageNet classification and face recognition [10] [11], as it exhibits good classification performance. In our work a simple CNN model is built and trained in advance to classify the gender of the person in the image – this training is done using the public *IMDB-WIKI* dataset [18]. Here, we will discuss what the specific actors are in our implementation and report performance with the output deployment against the baseline.

The CNN architecture that is used for our experiments contains two convolution layers followed by pooling layers and two fully-connected layers. As shown in Figure 5, nine actors were implemented to support these operations.

Specifically, OpenMP is utilized in each CNN layer to make full use of Raspberry Pi cores and speed up the application. The window size is set to 3 and results are reported with each data chunk comprising of 100 images. that desktop computer speeds up the execution of each actor, ranging from $1.5X$ to $9X$.

The best placement calculated and obtained from our deployment algorithm is as follows (Figure 6). The first 3 actors are deployed to the first layer station and others are deployed to the provider in the third layer, implying that the middle layer station is only responsible for transferring the output message of the third actor from previous station to the provider. The main reason for this deployment solution is that the fourth actor (the first convolution layer) has both high execution time and the transmission time. So, placing the fourth actor at the second level device does not help reduce these costs.

Figure 11 summarizes the power consumption in the edge device using our Actor Deployment Framework (ADF) compared to the baseline solution. The entire power consumption including transmission and computation power of the baseline solution is normalized to 100%. ADF generated solution only has 7.12% power consumption of baseline solution. The main reason is that in the baseline solution the entire video data is transmitted to the cloud node, requiring a large amount of power for the transmission. However, our solution leverages edge nodes to perform a subset of the actors, resulting in a large reduction of the amount of data required to be transmitted. Figure 12 shows the relationship of response latency between our solution and baseline. Here the latency of baseline is normalized to 100% and our deployment solution only has 5% of baseline's latency time, which means 95% latency reduction was achieved.

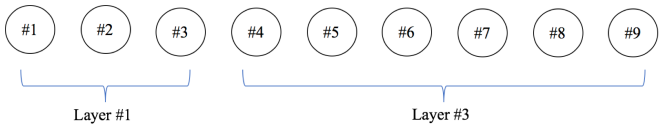

Fig. 6: Best Partitioning of CNN

C. FLD Application

The second application we implemented is for detecting facial landmarks (FLD) for the images captured in real time, which is often used in the scenarios like face swapping, face averaging, and others [3]. An overview of the design and implementation of this application is shown in Figure 7. In this application, the first actor captures the images from the camera whereas in the second actor, these RGB images are converted to gray images. Next the third actor will equalize the histogram of previous grayscale images in order to stretch out the intensity range (implemented by using the OpenCV function $equalizeHist$). The equalized images of the same size are sent to the fourth actor. After receiving the equalized images, the fourth actor detects the face using the function $get\_frontal\_face\_detector$ from the dlib library. Both equalized images and additional data about the face positions are sent to the last actor. The last actor detects the landscape based on the face detected in the previous actor and the face landmarks model obtained from $shape\_predictor\_68\_face\_landmarks.dat$ in the dlib library.

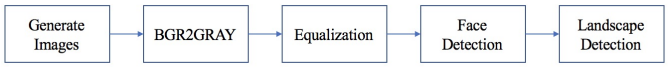

Fig. 7: FLD actors

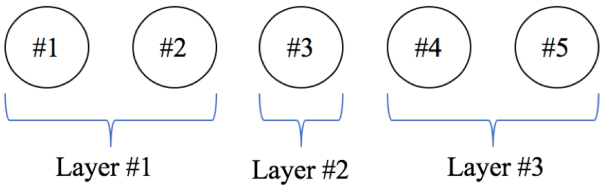

Fig. 8: Best Partitioning of FLD

Through our deployment algorithm, the best placement computed is shown in Figure 8. The output of the deployment algorithm suggests that the first two actors be deployed to the first station, the third actor be deployed to the middle station and the last two actors be deployed on the provider. In

this application, the reduction of power consumption against the baseline is close to 52%, as shown in Figure 11. Even though the computation power consumption is increased by moving a part of actors to the first station, a large reduction in transmission power consumption (around 67%) helps to reduce the total power consumption significantly. However, our response latency is around the same as the baseline's response latency. The main reason is that the fourth actor (face detection) takes more than 99% of total computation time, and thus placing other actors on the edge does not reduce the latency over cloud execution.

D. TSM Application

In recent years, traffic congestion has become a serious issue especially in modern cities, which requires sufficient solutions for traffic management and guidance. So here we want to explore this common IoT application – traffic situation management (TSM) – in our framework.

Our implementation of TSM application contains eight actors [14]. The application first loads frames from the video (size of $480 \times 800 \times 3$). These RGB frames are then converted into Grayscale images. In the third actor, background subtraction is performed for detecting foreground objects, and then GaussianBlur (kernel size 5×5) function from the OpenCV library is applied, and lastly, a threshold function is applied to ensure that the pixel's value greater than 40. In our fourth actor, two very common morphological operators (dilation and erosion) are applied. Next the function $findContours$ from the OpenCV library is used to retrieve contours from the images. Next these contours are converted into blobs and several filters are applied to filter these blobs. In the seventh actor, the current blobs are matched against previous blobs. The last actor will output the car number in a specified region based on the blobs.

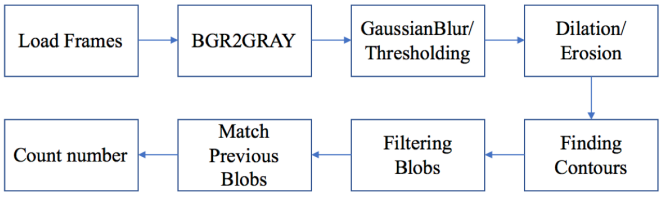

Fig. 9: TSM actors

Fig. 10: Best Partitioning of TSM

In our implementation, the best partitioning can be obtained by deploying first two actors to the first station and other actors to the provider. The power consumption of this application with this deployment, as shown in Figure 11, is about 42% of baseline's consumption. Though computation power consumption increased around 6% more from the baseline, the transmission power consumption reduced more than 60%. Figure 12 shows the relationship of the response latency between our solution and baseline, which is labeled TSM in the X-axis. Our deployment solution only takes up 39% of the baseline's latency time.

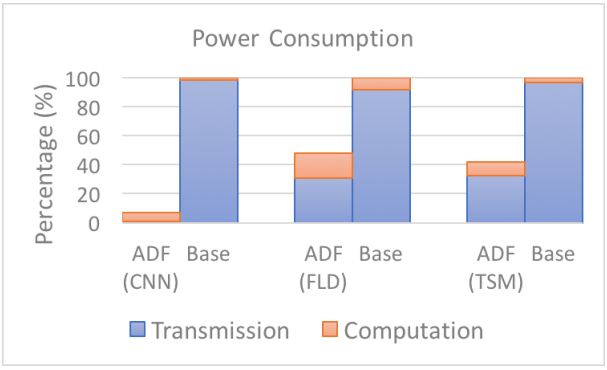

Fig. 11: Power Consumption against Baseline

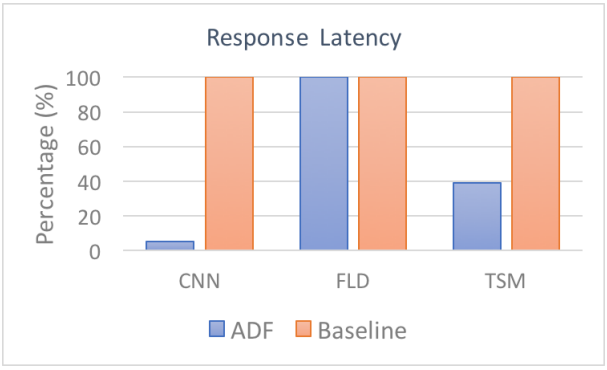

Fig. 12: Response Latency against Baseline

VI. Related Work

There has only been limited and relatively preliminary work on service development and management for IoT applications. Authors in [17] develop a distributed framework named MediaBroker for live stream management and data transformations. The main focus of their work is the type-aware data transport and the system for describing types of streaming data – in other words, resource constraints or deployment is not considered. Hong *et al.* [7] propose a high-level programming model called Mobile Fog that targets a large number of distributed heterogeneous devices. Each Mobile Fog process in their model handles the workload from a certain geo-spatial region and it supports a dynamic distribution of heavy workload within the same network hierarchy. However, they did not consider workload optimization between different levels of network hierarchy. Satyanarayanan *et al.* propose a decentralized cloud computing architecture called GigaSight [19] to support edge video analytics using virtual machine-based cloudlets. Their goals are to avoid sending huge amounts of video data to the cloud and enforcement of privacy preferences. Although a multi-step pipeline is considered for video analytics, automated deployment of those steps is not their focus. Bittencourt *et al.* discuss the quality of service issues by introducing three scheduling policies in Fog Computing [4]. These policies include Concurrent, First

Come-First Served, and Delay-priority policies. A decision of where a new application should be deployed – whether in the cloudlet or the cloud – is made using these strategies. However, for any given application, all application modules are deployed either in the cloud or in the cloudlet, i.e., an application is not decomposed. Zhang *et al.* have presented a framework, Firework, to facilitate the sharing of IoT applications through a virtual shared data view and service composition [22]. Their approach involves manually testing every possible deployment choice – this is not scalable and feasible with dynamically changing workloads. The work from Mehta *et al.* [13] is similar to our work, in the sense that they also propose a distributed algorithm for the automatic development of IoT applications in heterogeneous environments. However, their domain is not streaming applications for which cost computation is more complex. They also do not consider power consumption in edge devices as a factor in decision making. Cardellini*et al.*[5] also provided a general formulation of placement for stream processing applications. However, the important constraint – power consumption – is not considered as well in their work.

Building on Function as a Service (FAAS), Calvin framework [15] divided IoT applications into four well-defined aspects - *describe*, *connect*, *deploy* and *manage*. Their main concept is the use of *actors* to provide a *data flow* model of application components. Mehta *et al.* extended the Calvin framework to cover resource-constrained devices [12]. However, neither of these works provided any deployment strategy to specify where each actor should execute.

VII. Conclusions

This paper has presented the design, implementation, and evaluation of a framework for automating the placement of actors on the devices in multi-layer wireless networks. The placement is performed with the goal of optimizing a cost function that combines power consumption at the extreme edge device and application latency. Several challenges have been addressed in the process: a nuanced framework design, a dynamic programming-based deployment algorithm, a cost-model that is used as the objective function, and a power model to estimate power usage. Evaluations with three streaming applications compared to a baseline solution, where the edge device directly pushes input data to a cloud node, show that the reductions in power consumption in our approach achieve from 52% to 92% and the largest latency reduction is close to 95%. Experiments with the fourth application demonstrate our framework's ability to make stable placement decisions when there are short-term changes in the distribution of data, while also adjusting placements when the change is long-term.

References

[1] Ola Angelsmark and Per Persson. Requirement-based deployment of applications in calvin. In Ivana Podnar Žarko, Arne Broering, Sergios Soursos, and Martin Serrano, editors, *Interoperability and Open-Source Solutions for the Internet of Things*, pages 72–87, Cham, 2017. Springer International Publishing.

[2] Luigi Atzori, Antonio Iera, and Giacomo Morabito. The internet of things: A survey. *Comput. Netw.*, 54(15):2787–2805, October 2010.

[3] Dmitri Bitouk, Neeraj Kumar, Samreen Dhillon, Peter Belhumeur, and Shree K. Nayar. Face swapping: Automatically replacing faces in photographs. In *ACM SIGGRAPH 2008 Papers*, SIGGRAPH '08, pages 39:1–39:8, New York, NY, USA, 2008. ACM.

[4] L. F. Bittencourt, J. Diaz-Montes, R. Buyya, O. F. Rana, and M. Parashar. Mobility-aware application scheduling in fog computing. *IEEE Cloud Computing*, 4(2):26–35, March 2017.

[5] Valeria Cardellini, Vincenzo Grassi, Francesco Lo Presti, and Matteo Nardelli. Optimal operator placement for distributed stream processing applications. In *Proceedings of the 10th ACM International Conference on Distributed and Event-Based Systems*, DEBS '16, page 69–80, New York, NY, USA, 2016. Association for Computing Machinery.

[6] M. Chiang, S. Ha, C. I, F. Risso, and T. Zhang. Clarifying fog computing and networking: 10 questions and answers. *IEEE Communications Magazine*, 55(4):18–20, April 2017.

[7] Kirak Hong, David Lillethun, Umakishore Ramachandran, Beate Ottenwälder, and Boris Koldehofe. Mobile fog: A programming model for large-scale applications on the internet of things. In *Proceedings of the Second ACM SIGCOMM Workshop on Mobile Cloud Computing*, MCC '13, pages 15–20, New York, NY, USA, 2013. ACM.

[8] F. Kaup, P. Gottschling, and D. Hausheer. Powerpi: Measuring and modeling the power consumption of the raspberry pi. In *39th Annual IEEE Conference on Local Computer Networks*, pages 236–243, 2014.

[9] Y. N. Krishnan, C. N. Bhagwat, and A. P. Utpat. Fog computing - network based cloud computing. In *2015 2nd International Conference on Electronics and Communication Systems (ICECS)*, pages 250–251, Feb 2015.

[10] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E Hinton. Imagenet classification with deep convolutional neural networks. In F. Pereira, C. J. C. Burges, L. Bottou, and K. Q. Weinberger, editors, *Advances in Neural Information Processing Systems 25*, pages 1097–1105. Curran Associates, Inc., 2012.

[11] S. Lawrence, C. L. Giles, Ah Chung Tsoi, and A. D. Back. Face recognition: A convolutional neural-network approach. *Trans. Neur. Netw.*, 8(1):98–113, January 1997.

[12] A. Mehta, R. Baddour, F. Svensson, H. Gustafsson, and E. Elmroth. Calvin constrained - a framework for iot applications in heterogeneous environments. In *2017 IEEE 37th International Conference on Distributed Computing Systems (ICDCS)*, pages 1063–1073, June 2017.

[13] A. Mehta and E. Elmroth. Distributed cost-optimized placement for latency-critical applications in heterogeneous environments. In *2018 IEEE International Conference on Autonomic Computing (ICAC)*, pages 121–130, 2018.

[14] Ahmet özlü. Vehicle detection, tracking and counting. https://github.com/ahmetozlu/vehicle_counting, 2017.

[15] Per Persson and Ola Angelsmark. Calvin - merging cloud and iot. In *ANT/SEIT*, 2015.

[16] J. S. Preden, K. Tammemäe, A. Jantsch, M. Leier, A. Riid, and E. Calis. The benefits of self-awareness and attention in fog and mist computing. *Computer*, 48(7):37–45, July 2015.

[17] Umakishore Ramachandran, Martin Modahl, Ilya Bagrak, Matthew Wolenetz, David J. Lillethun, Bin Liu, James Kim, Phillip W. Hutto, and Ramesh Jain. Mediabroker: A pervasive computing infrastructure for adaptive transformation and sharing of stream data. *Pervasive and Mobile Computing*, 1(2):257–276, 2005.

[18] Rasmus Rothe, Radu Timofte, and Luc Van Gool. Deep expectation of real and apparent age from a single image without facial landmarks. *International Journal of Computer Vision (IJCV)*, July 2016.

[19] M. Satyanarayanan, P. Simoens, Y. Xiao, P. Pillai, Z. Chen, K. Ha, W. Hu, and B. Amos. Edge analytics in the internet of things. *IEEE Pervasive Computing*, 14(2):24–31, Apr 2015.

[20] Ashkan Yousefpour, Caleb Fung, Tam Nguyen, Krishna Kadiyala, Fatemeh Jalali, Amirreza Niakanlahiji, Jian Kong, and Jason P. Jue. All one needs to know about fog computing and related edge computing paradigms: A complete survey. *CoRR*, abs/1808.05283, 2018.

[21] A. Zanella, N. Bui, A. Castellani, L. Vangelista, and M. Zorzi. Internet of things for smart cities. *IEEE Internet of Things Journal*, 1(1):22–32, Feb 2014.

[22] Q. Zhang, Q. Zhang, W. Shi, and H. Zhong. Firework: Data processing and sharing for hybrid cloud-edge analytics. *IEEE Transactions on Parallel and Distributed Systems*, 29(9):2004–2017, Sep. 2018.

[23] Qi Zhang, Lu Cheng, and Raouf Boutaba. Cloud computing: state-of-the-art and research challenges. *Journal of Internet Services and Applications*, 1(1):7–18, May 2010.