2018 Third ACM/IEEE Symposium on Edge Computing

F-MStorm: Feedback-based Online Distributed Mobile Stream Processing

Mengyuan Chao, Chen Yang, Yukun Zeng, Radu Stoleru
Texas A&M University, USA
Email: {chaomengyuan, yangchen08, yzeng}@tamu.edu; stoleru@cse.tamu.edu

***Abstract*—A distributed mobile stream processing system allows mobile devices to process stream data that exceeds any single device's computation capability without the help of infrastructure. It is paramount to have such a system in many critical application scenarios, such as military operations and disaster response, yet an *efficient online* mobile stream processing system is still missing. In this paper, we make the key observation that *the unique characteristics of mobile stream processing* call for a *feedback-based system design*, which is in sharp contrast with the static configuration and scheduling of the current mobile stream processing system, "MStorm" [1]. At first, we demonstrate the inefficiencies of MStorm through several real-world experiments. Then, we propose F-MStorm, a feedback-based online distributed mobile stream processing system, which adopts the feedback-based approach in the *configuration*, *scheduling* and *execution* levels of system design. We implement F-MStorm on Android phones and evaluate its performance through benchmark applications. We show that it achieves up to 3x lower response time, 10% higher throughput and consumes 23% less communication energy than the state-of-the-art systems.**

***Index Terms*—stream processing; edge computing; scheduling**

I. Introduction

Emerging computation-intensive mobile applications [2]–[7] that process stream data collected by various mobile sensors often require more computation resources than a single device can provide. Therefore, many existing systems [8]–[10] offload the computation-intensive tasks to the cloud or nearby high performance computing (HPC) servers to achieve low latency. For example, MCDNN [8] offloads deep neural network based video processing tasks to the cloud, Odessa [9] chooses nearby HPCs as additional computing resources to recognize objects from real-time videos and LEO [10] utilizes on-chip DSP co-processors, GPUs together with the cloud to run inference algorithms. Although such systems achieve low latency and high throughput, resources in the cloud or nearby HPC servers are not always accessible in some infrastructure-less scenarios. For example, imagine the following scenario:

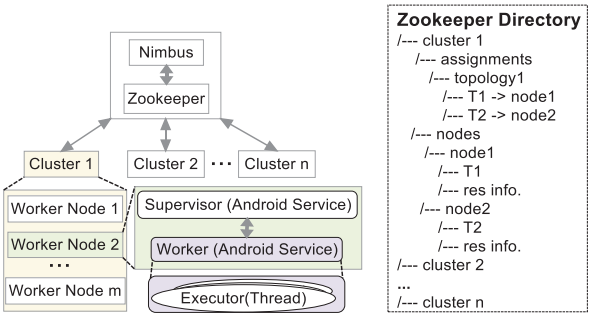
"A group of first responders, equipped with mobile devices, is assigned to a post-earthquake area to discover dangerous zones (e.g., leaking chemical pipes or unstable buildings) to avoid. The teams collect a large amount of data via different sensors (e.g., on body video cameras) and analyze the data in real time via analytical software. Usually, such data analysis requires significant computational resources and, thus, it is pushed to the cloud for analysis. However, the communication infrastructure was destroyed during the earthquake, which makes offloading to the cloud impossible. In such case, offloading computation to the nearby mobile devices at the edge becomes a promising option."

In this paper, we focus on **a distributed stream processing system deployed on a cluster of mobile devices without an Internet access (stream processing at the edge)**. Different from most stream processing systems that run on a cluster of wire-connected servers in the cloud (such as Storm [11], Spark [12] and Flink [13], etc.), stream processing on a cluster of mobile devices is much more complicated, because mobile devices have very limited computation resources and batteries, and the wireless links between different devices are unstable. Some existing works [14], [15] are designed for the same environment as ours. However, they focus on processing bounded batch jobs instead of unbounded stream data.

MStorm [1] makes an important first step towards mobile stream processing at the edge by implementing a lightweight system on mobile phones. It provides some basic functionality, such as *parallelism configuration*, *task scheduling* and *stream grouping*. However, since its current implementation ignores some specific characteristics of stream processing at the edge, it is inefficient as we demonstrate through some simple experiments. First, MStorm configures the number of executors (threads that execute tasks) at each device simply based on CPU cores while not taking into account the current CPU utilization. As the computation resources of a mobile device is also shared by other applications, this static configuration can easily lead to a bottleneck that negatively impacts the system performance (response time and throughput). Second, the task assignment, when MStorm assigns computation tasks to devices, is based on a naive round robin strategy. This may incur unnecessary inter-device traffic and consequently higher delay and energy consumption. Third, MStorm adopts a shuffle stream grouping mechanism, where upstream tasks distribute the output to downstream tasks uniformly at random. However, as downstream tasks may run on highly occupied devices, the shuffle grouping mechanism might cause congestion there and lead to high response time and low throughput.

To solve these problems, one potential solution is to carve out some static resources dedicated for stream processing [16]. However, unlike servers in the cloud, the resources of mobile devices at the edge are very limited and need to be shared with some other resource-intensive applications. It is unreasonable to allow MStorm to take up some resources even when there is no stream processing tasks. Another approach is to apply a pull model [17], [18] like most modern cloud computing systems, where the machines ask for tasks when they have free slots. However, this model is not enough for stream processing at the edge as it does not consider other factors like device-to-device delays and remaining batteries of devices. Our insight is that, *instead of adopting an open-loop task scheduling which assumes a static environment, a feedback-based approach that*

978-1-5386-9445-9/18/$31.00 ©2018 IEEE
DOI 10.1109/SEC.2018.00027



(a) MStorm Architecture (b) Zookeeper directory

Fig. 1: MStorm system architecture

makes decisions based on the changing system state should be utilized to deal with the dynamic environment at the edge.

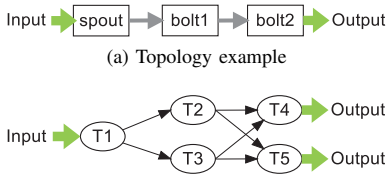
To accomplish this goal, we propose F-MStorm, a feedback-based online distributed mobile stream processing system. F-MStorm adopts a feedback-based approach at many different levels of system design so that the system can adapt quickly to the changing environment to achieve high performance. At the configuration level, F-MStorm configures the number of executors on each mobile device based on the free CPU resources of mobile devices and CPU usage of tasks. At the scheduling level, F-MStorm assigns tasks to mobile devices based on the task-to-task traffic and device-to-device communication delay and energy consumption. At the execution level, the upstream tasks distribute the output data to the downstream tasks based on the latter's stream arrival/processing rate and waiting queue length. We implement a prototype of F-MStorm on Android phones and evaluate its performance through a customizable benchmark application. We also compare F-MStorm with two scheduling algorithms proposed for Storm (i.e., T-Storm [19] and R-Storm [20]). The experimental results show that, by using the feedback information, F-MStorm achieves up to 3x lower response time, 10% higher throughput and 23% less communication energy than the state-of-the-art systems.

Our main contributions are summarized as follows:

- Through some real world experiments, we demonstrate that, without an accurate estimation of the current system state and appropriate adjustment of the initial configuration, task scheduling and stream grouping, MStorm suffers up to three order of magnitude increase in response time and 60% reduction in throughput (Section II), which calls for the feedback-based system design.
- We propose F-MStorm (Section III), which consists of a feedback-based configuration (FBC) method, a feedback-based task assignment (FBA) algorithm and a feedback-based stream grouping (FBG) strategy.
- We implement F-MStorm on Android phones and conduct real world experiments to evaluate its performance (Section IV), which demonstrate the superiority of F-MStorm over MStorm and two other state-of-the-art systems.

II. Background and Motivation

In this section, we at first briefly introduce the background of MStorm and its architecture. Then, we show the inefficiency of MStorm via three experiments. Finally, we motivate our F-MStorm by explaining why existing solutions do not work.



(a) Topology example

(b) Extended topology example

Fig. 2: Example of an MStorm application

A. MStorm

MStorm [1] is the first online distributed stream processing system running on mobile devices with Android OS. It is designed for critical scenarios such as military operations and disaster response, where no Internet access is available, whereas the mobile devices of the team members are connected as a cluster through a manpack Wi-Fi access point. Instead of porting popular stream processing systems (such as Storm [11], Spark [12] or Flink [13]) running on the cloud, MStorm is designed and implemented from scratch with a lightweight infrastructure. This is paramount for mobile stream processing at the edge, which only has limited resources.

MStorm adopts some technical designs from Apache Storm. Its architecture is shown in Fig. 1a. An MStorm master node contains one **Nimbus** and one **Zookeeper** service. Nimbus schedules task execution while Zookeeper coordinates between Nimbus and mobile devices and maintains the cluster metadata in a directory-like structure shown in Fig. 1b. Every mobile device in the MStorm cluster runs a **supervisor** process and a **worker** process, both as Android services. The supervisor receives tasks from Nimbus and assigns tasks to the worker, while the worker manages multiple **executors** (threads) which are used to execute tasks. MStorm guarantees an at-most-once processing semantics.

An application in MStorm is modeled as a directed graph called **topology**. A topology contains two types of nodes, i.e., **spout** and **bolt**. A spout partitions the input stream into tuples and sends these tuples to downstream bolts. A bolt processes tuples from spout or upstream bolts, and sends the processed tuples to downstream bolts for further processing. We refer to a spout or a bolt as **application component** (or simply component) in the rest of the paper. A directed edge between two nodes in a topology indicates that traffic flows from one to the other. Each component can spawn multiple parallel tasks which are executed by devices' executors. If we expand the component in a topology with multiple nodes, each of which represents an individual task, we get another directed graph, i.e., the **extended topology**. The extended topology graph shows the actual data flow between individual tasks. Fig. 2 shows the topology and the extended topology of a sample MStorm application that contains one spout and two bolts, i.e., bolt1, bolt2. Each bolt further contains two parallel tasks, i.e., T2, T3 for bolt1, and T4, T5 for bolt2, respectively.

The MStorm application developer needs to provide the application topology as well as configure the *parallel tasks* for each application component. MStorm then decides *the number of executors* for each device and assigns tasks to devices for execution. The output tuples of each task may need to be sent to the downstream tasks. The mechanism for distributing tuples is called **stream grouping**. MStorm currently adopts a

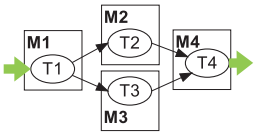
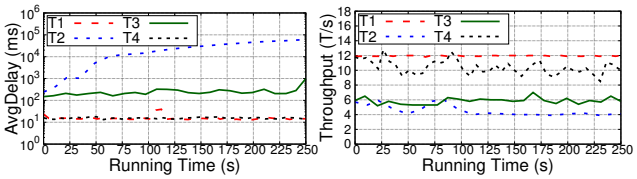


Fig. 3: Sample application that demonstrates the inefficiency of resource-unaware configuration and stream grouping



(a) Delay (b) Throughput

Fig. 4: The results of the sample application that demonstrates poor performance of resource-unaware configuration

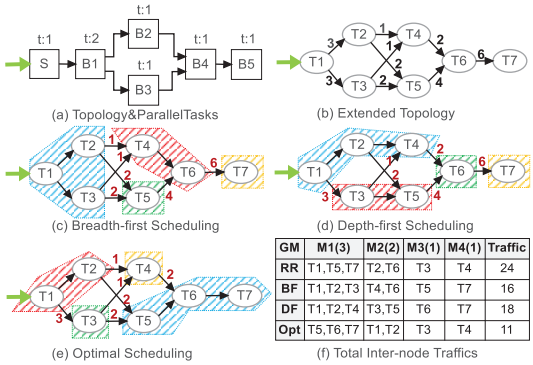
shuffle grouping strategy, i.e., tuples are randomly distributed to downstream tasks such that all tasks have identical expected workload in the long run.

B. Limitation of MStorm

Although MStorm makes an important first step towards a successful design of a mobile distributed stream processing system, *its unawareness of system resource utilization and mobile network's characteristics* lead to the suboptimal behaviors that prevent it from achieving a good performance. In the following sections, we present three experiments that show the inefficiency of the current MStorm system.

1) Resource-unaware configuration (RUC): We refer to the configuration as *the users' preference of parallelism for each component* and *the system's initial configuration of the number of executors at each device*. In MStorm, the system configures the number of executors at each device based on its CPU cores. However, since users may run other applications on the mobile device, the available CPU resources depend not only on the CPU cores but also on the current utilization. Configuring the number of executors without an accurate estimation of current resource utilization may lead to performance bottleneck, where the heavily-used nodes may be assigned identical number of executors compared to the idle ones.

We demonstrate this inefficiency through a sample application shown in Fig. 3, which consists of 4 tasks (T1 - T4). Four Google Nexus 5 mobile devices M1 - M4 form a cluster and MStorm configures identical number of executors per node. Due to *round robin* task assignment, which we discuss later, as it is also inefficient, each mobile device needs to execute one task. Consider the case when M2 is actively used by other user's application, MStorm fails to adapt to this situation and results in poor performance as shown in Fig. 4. We generate a stream of data at 12 tuples per second (T/s) and measure the delay and throughput of each task. As we can see, the delay for T2 increases from 100ms to 100s because of queuing after the system runs for 250s. This is unacceptable for any real-time mobile application. Besides, the overall throughput (10T/s) is smaller than the input rate (12T/s) due to the bottleneck at T2. This leads to increasing queues in the system.



GM	M1(3)	M2(2)	M3(1)	M4(1)	Traffic
RR	T1,T5,T7	T2,T6	T3	T4	24
BF	T1,T2,T3	T4,T6	T5	T7	16
DF	T1,T2,T4	T3,T5	T6	T7	18
Opt	T5,T6,T7	T1,T2	T3	T4	11

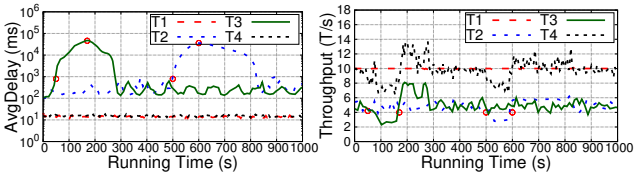
(f) Total Inter-node Traffics

Fig. 5: The inter-device traffic incurred by different scheduling

2) Traffic-unaware task assignment: As mentioned earlier, MStorm uses a *round robin* task assignment strategy, i.e., it sequentially assigns tasks to each mobile device until all tasks are assigned. Although it is easy to implement, it does not minimize the inter-device traffic, which leads to a higher delay and energy consumption. To reduce the inter-device traffic, an intuitive idea is to put as many tasks as possible on the same node. This leads to our two preliminary attempts for a more efficient task scheduling algorithm, namely breadth and depth-first scheduling, respectively. The breadth (depth) first scheduling works as follows: at first, sort the mobile device based on the number of configured executors; then assign tasks to the same mobile device in a breadth (depth) first order when traversing the extended topology, until the assigned tasks reach its capacity or there is no more task to assign. However, we reveal by the following example that, even if the breadth (depth) first scheduling reduces some inter-device traffic, their performance are still much worse than the optimal schedule.

Fig. 5 (a) and (b) represent the topology and extended topology of an application. The edge weights in the extended topology represent the total traffic from task to task during a period ΔT. All tasks are assigned to mobile devices M1, M2, M3, M4 with 3, 2, 1, 1 executors by different scheduling algorithms. Based on the breadth/depth-first scheduling, tasks are assigned to mobile devices as shown in Fig. 5 (c) and (d). Figure (e) represents the optimal scheduling that minimizes the inter-device traffic. Fig. 5 (f) summarizes the scheduling result and corresponding inter-device traffic for each scheduling algorithm, which shows that the round robin scheduling generates 118% more inter-device traffic than the optimal scheduling, whereas the breadth-first and depth-first scheduling also generates 45%, 64% more inter-device traffic than the optimal. This is because they fail to further distinguish different inter-task traffic and don't assign tasks with large inter-task traffic to the same node.

Moreover, even if we distinguish different inter-task traffic, it is still coarse grained, considering the diversity of wireless links. For example, given two tasks with fixed inter-task traffic, if they are assigned to two nodes with a lower inter-device delay, the total delay will be lower. If they are assigned to two nodes with lower communication power, the total energy consumption for traffic transmission will be less. With round robin or breadth (depth) first scheduling, all these potential chances of improving system performance will be missed.



(a) Delay (b) Throughput

Fig. 6: The results of the sample application that demonstrates poor performance of resource-unaware shuffle grouping

Furthermore, except for minimizing delay and energy consumption, sometimes soldiers or first responders require the system to last longer. To achieve this goal, the tasks need to be assigned based on the remaining battery of each device. With round robin or breadth (depth) first scheduling, the batteries of some devices might be depleted very soon.

3) Resource-unaware stream grouping: Recall that MStorm adopts a *shuffle grouping* strategy to distribute output tuples to downstream tasks. Although shuffle grouping achieves fairness among tasks in terms of overall workload, we demonstrate that it cannot adapt to the resource fluctuation resulted from users' own application usage.

We use the same application as shown in Fig. 3, where all nodes are idle in the beginning. We set the input rate at 10T/s. We run resource-intensive applications on M2 and M3 at 500s and 50s, and close them at 600s and 170s, respectively. Since shuffle grouping assigns output tuples from T1 to T2 and T3 uniformly at random, the arrival rate at T2 and T3 are 5T/s on average, even if M2 and M3 are busy with other applications. As a result, we can see a significant increase in response time in Fig. 6a (from roughly 10^2ms to 10^5ms) and a decrease in throughput in Fig. 6b (from roughly 5T/s to 2T/s), when the resource-intensive application is running.

A key observation from Fig. 3 is that the throughput of T2 and T3 increases to 6T/s and 8T/s after the resource-intensive application terminates, which indicates a higher processing capability than the average input rate of 5T/s. Therefore, it is possible to improve the overall throughput by assigning more stream to other more capable nodes, given that we have an accurate online estimation of resource utilization at each node.

C. Motivation of F-MStorm

One simple approach to solve the problems above is to carve out some static resources on each mobile device dedicated for stream processing [16]. However, unlike servers in the cloud that are uniformly managed by a cluster manger to undertake specific jobs, the limited resources of mobile devices at the edge are shared by both MStorm and other user applications. Those applications might also be resource-intensive and even have higher priorities. It is unreasonable to allow MStorm to take up the valuable resources even when there is no stream processing tasks. Another approach is to apply a pull model [17], [18] like most modern cloud computing systems, where the machines ask for tasks when they have free slots. However, this model is not enough for stream processing at the edge as it does not consider other factors like device-to-device delays, energy consumption of inter-device communication and remaining batteries of devices. We argue that, instead of adopting an open-loop task scheduling algorithm which assumes a rather static environment, a feedback-based approach which makes decisions based on the current system state (such as CPU utilization, delays and energy consumption of inter-device communication, remaining batteries, etc.) should be utilized to deal with the dynamic environment at the edge. Two ameliorations proposed for Apache Storm, namely T-Storm [19] and R-Storm [20], have similar insights with F-MStorm. They also adopt a feedback-based approach to improve the system performance. However, neither of them concerns the energy consumption and balance of energy usage, as they are proposed for systems running in the cloud. Nevertheless, for a mobile stream processing system running at the edge, the above two factors directly decide how long a system can last for. It is paramount for F-MStorm to take all these factors into account.

III. Design and Implementation of F-MStorm

In this section, we present the design and implementation of F-MStorm. To overcome the inefficiencies presented in the previous section, F-MStorm sets configuration, task scheduling and stream grouping all based on the feedback information. To better present our idea, we use a scenario with m devices and an application with N components. The mathematical model of the problem involves many notations. For readers' convenience, we summarize them in Table I.

A. Overview

Similar with MStorm, F-MStorm configures the parallelism of each application component based on the user's experience and assigns tasks to the mobile devices through round robin at the beginning. Then, after a "warm-up" period, each mobile device periodically reports the feedback information, including task execution, device resources and network condition, etc., to Nimbus. Based on this feedback, F-MStorm reconfigures tasks for each component, resets available executors for each device and recalculates the best schedule for the whole application. Because of the dynamic processing workload and changing environment, the best task schedule might change from time to time. However, the new best schedule sometimes only achieve a small performance improvement than the previous one. In such case, it is not beneficial to switch to the new schedule, considering the rescheduling overhead and system stability. To deal with this issue, we propose several reschedule conditions. If none of these conditions are met, the system just keeps the origin schedule; otherwise, the reschedule takes place. Except for reporting to Nimbus, each downstream task needs to report the execution information to the upstream tasks periodically. The upstream tasks then distribute the output to the downstream tasks based on the feedback information.

B. Status Report

In F-MStorm, each mobile device reports the following information to Nimbus periodically (every ΔT):

- w_j: the CPU usage (in MHz) of task j obtained from */proc/stat* and */proc/stat/pid/task/tid/stat* [19].
- l_j: the queue length at task j, i.e., the number of stream tuples that are waiting to be processed.
- λ_j and μ_j: the average input and processing rate (in T/s) of task j during ΔT.
- $t_{jj'}$: the output rate (in T/s) from task j to j' during ΔT.

TABLE I: Main notations

Notation	Description
i	Index of component, $i = 1, ..., N$
j	Index of task, $j = 1, ..., n$. n varies with configurations
k	Index of mobile device, $k = 1, ..., m$
$A(i)$	Task set of component i
$c(j)$	Component that task j belongs to
$v(j)$	Device that task j is assigned to
W_i	Expected CPU usage of component i
I_i	Expected input rate of component i
O_i	Expected output rate of component i
w_j	CPU usage (MHz) of task j
l_j	Waiting queue length at task j
λ_j	Input rate (T/s) of task j
μ_j	Processing rate (T/s) at task j
$t_{jj'}$	Output rate (T/s) from task j to j'
$s_{jj'}$	Average tuple size (bit) from task j to j'
f_k	Single core frequency (MHz) of device k
c_k	The number of CPU cores of device k
u_k	Total CPU usage (MHz) of device k
r_k	Available CPU resource (MHz) at device k
$d_{kk'}$	Communication delay (ms) from device k to k'
b_k	Remaining battery at device k
e_k^t	Energy consumption (nJ) per bit for Tx at device k
e_k^r	Energy consumption (nJ) per bit for Rx at device k
$\boldsymbol{P}$	Vector: parallel task number for each component
$\boldsymbol{E}$	Vector: available executors for each mobile device
$\boldsymbol{B}$	Vector: remaining battery for each mobile device
$\boldsymbol{T}$	Matrix: average output rate from task to task
$\boldsymbol{S}$	Matrix: average tuple size from task to task
$\boldsymbol{D}$	Matrix: communication delay from device to device
$\boldsymbol{Q}$	Matrix: energy/bit for transmitting between devices
$\boldsymbol{X}$	Matrix: task assignment to mobile devices
ΔT	Period that mobile devices report to Nimbus
Δt	Period that tasks report to upstream tasks

- $s_{jj'}$: the average tuple size (bit) from task j to j' during ΔT. It is defined as the ratio between the total tuple data size and the total number of tuples from task j to j'.
- r_k: the available CPU resource (in MHz) at device k, defined as $r_k = f_k \cdot c_k - (u_k - \sum_{v(j)=k} w_j)$, where f_k is the single core frequency, c_k is the number of cores, u_k is the current CPU usage at device k, and $\sum_{v(j)=k} w_j$ is the total CPU usage of current F-MStorm tasks at device k.
- $d_{kk'}$: device-to-device communication delay (in ms) from device k to k'.
- b_k: remaining battery (in mAh) at device k.
- e_k^t and e_k^r: energy consumption per bit (nJ/bit) for tuple transmission (Tx) and reception (Rx). They can be estimated based on throughput [21], which we obtained from *Device-BandwidthSampler* [22].

Nimbus maintains a moving average for each status, that is, $V = \delta * V_{old} + (1-\delta) * V_{new}$, where V_{old} is the old value stored at Nimbus, V_{new} is the new feedback value, and $0 \leq \delta \leq 1$ is a factor used to indicate how the status depends on the history.

It deserves to be mentioned that, periodically reporting and updating these system statuses might cause some extra communication and computing overhead. However, compared with the communication traffic size and processing workload of stream data, the overhead is negligible.

C. Feedback Based Configuration (FBC)

Based on the feedback, Nimbus calculates the following vectors to reconfigure the system: $\boldsymbol{P} = [P_i]_{i=1}^N$, $\boldsymbol{E} = [E_k]_{k=1}^m$. P_i represents the number of parallel tasks of component i and E_k represents the available executors of each mobile device k. They are calculated by equation $P_i = \lceil \frac{W_i}{\mathcal{R}} \rceil$ and $E_k = \lfloor \frac{r_k}{\mathcal{R}} \rfloor$, where W_i is the expected CPU usage of component i, r_k is the available CPU resource at device k and $\mathcal{R}$ represents the computing resource of each executor. The ceiling and floor functions are used to leave some margins for device resource fluctuation. $\mathcal{R}$ is calculated by the following equations:

$$\begin{cases} \mathcal{R} &= \min\{\max\{\mathcal{R}_l, \mathcal{R}_e\}, \mathcal{R}_u\} \\ \mathcal{R}_l &= \eta_l * \min_k\{f_k\} \\ \mathcal{R}_e &= \min_{i,k}\{W_i, \frac{r_k}{c_k}\} \\ \mathcal{R}_u &= \eta_u * \min_k\{f_k\} \end{cases} \quad (1)$$

where η_l and η_u ($0 \leq \eta_l \leq \eta_u \leq 1$) are parameters to control the lower and upper bound of an executor's resource. The intuitions are as follows. First, the resource of an executor is mostly determined by the CPU usage of components and the available resource of mobile devices. Based on this, we can calculate a basic version of executor resource, namely $\mathcal{R}_e$. Then, to make full use of the CPU resource, a single executor should not occupy more resource than a CPU core. Therefore, we need to set an upper bound $\mathcal{R}_u$ for the executor resource. On the other hand, the resource of an executor should not be too little, otherwise a mobile device will be configured with too many executors, which incurs a lot of OS scheduling overheads. Therefore, we also need to add a lower bound $\mathcal{R}_l$ for the executor resource.

D. Feedback Based Assignment (FBA)

We formulate the task assignment in F-MStorm as a mixed-integer quadratic programming (MIQP) and solve it by a genetic algorithm. Moreover, to ensure the system stability, we propose 4 reschedule conditions to avoid frequent reschedules.

1) Problem Formulation: Let matrix $\boldsymbol{T} = [T_{jj'}]_{n \times n}$ represent the expected tuple output rate from task to task, with $T_{jj'} = \frac{I_i/P_i}{\sum_{j'} t_{jj'}} * t_{jj'}$, where $i = c(j)$ is the component that task j belongs to, I_i is the expected input rate of component i. I_i/P_i represents the expected tuple input rate of each task. Let matrix $\boldsymbol{S} = [s_{jj'}]_{n \times n}$ represent the measured average tuple size from task to task and let matrix $\boldsymbol{D} = [d_{kk'}]_{m \times m}$ represent the communication delay between mobile devices. Let matrix $\boldsymbol{Q} = [Q_{kk'}]_{m \times m}$ represent the energy per bit for communication from device k to k', where $Q_{kk'} = e_k^t + e_{k'}^r$. We denote the decision variables for the task assignment as a n-by-m 0-1 matrix $\boldsymbol{X}$, with $X_{jk} = 1$ representing that task j is assigned to device k.

Our objective is to minimize the average *end-to-end delay and energy consumption* for each tuple while ensuring the load balance in energy consumption. The end-to-end delay consists of processing delay, queuing delay and communication delay. The energy consumption consists of processing energy and communication energy. Since the system we care is homogeneous in executor's processing speed and energy consumption model, the way we assign tasks will not affect the end-to-end processing delay, queuing delay and processing energy. Therefore, the objective is reduced to minimize the average *end-to-end communication delay and communication energy consumption* while ensuring the load balance in energy consumption. We formulate the problem as:

$$\underset{\boldsymbol{X}}{\text{minimize}} \quad F = \alpha * \frac{g_d}{g_d^{max}} + \beta * \frac{g_q}{g_q^{max}} + \gamma * \frac{g_b}{g_b^{max}} \quad (2)$$

$$\text{s.t.} \quad \begin{aligned} &\forall j, \ \sum_{k=1}^{m} X_{jk} = 1 \\ &\forall k, \ \sum_{j=1}^{n} X_{jk} \leq E_k \\ &\forall j,k, \ X_{jk} \in \{0,1\} \end{aligned} \tag{3}$$

where g_d is the average communication delay, g_q is the average communication energy consumption and g_b is the load balance index. g_d^{max}, g_q^{max}, g_b^{max} represent the maximum g_d, g_q and g_b that are used to unify the units. $\alpha, \beta, \gamma \in [0,1]$ are customized according to the user's preference.

g_d is calculated as follows. Given the task-to-task output rate matrix $\boldsymbol{T}$ and task assignment matrix $\boldsymbol{X}$, we can obtain matrix $\boldsymbol{T'} = \Delta T \cdot \boldsymbol{TX}$, where element T'_{jk} represents the total number of tuples output by j from device $v(j)$ to k during ΔT. Similarly, we can obtain matrix $\boldsymbol{D'} = \boldsymbol{XD}$, where element D'_{jk} represents the communication delay from $v(j)$ to k. With $\boldsymbol{T'}$ and $\boldsymbol{D'}$, we can obtain matrix $\boldsymbol{M^d} = \boldsymbol{T'} \odot \boldsymbol{D'}$, where element M^d_{jk} represents the total communication delay of tuples output by task j from $v(j)$ to k during ΔT, and $\odot$ represents Hadamard product. $\lambda = \sum_{j \in A(1)} \mu_j$ represents the total output rate of the spout. Then, the average communication delay of each tuple is calculated as:

$$\begin{aligned} g_d = \frac{\sum_{j,k} \boldsymbol{M^d_{jk}}}{\lambda \Delta T} &= \frac{\sum_{j,k} \Delta T \cdot (\boldsymbol{TX} \odot \boldsymbol{XD})_{jk}}{\lambda \Delta T} \\ &= \frac{\sum_{j,k} (\boldsymbol{TX} \odot \boldsymbol{XD})_{jk}}{\lambda} \end{aligned} \tag{4}$$

Next, we consider the calculation of g_q. Similar to the communication delay, we utilize matrix $\boldsymbol{T''} = \Delta T \cdot (\boldsymbol{T} \odot \boldsymbol{S})\boldsymbol{X}$ to represent the total traffic data size output by task j from device $v(j)$ to k in ΔT, matrix $\boldsymbol{Q'} = \boldsymbol{XQ}$ to represent the energy consumption per bit for tuple transmission by Wi-Fi from device $v(j)$ to k. Thereby, the total energy consumption for tuple transmission from device $v(j)$ to k in ΔT can be represented as matrix $\boldsymbol{M^q} = \boldsymbol{T''} \odot \boldsymbol{Q'}$. Then, the power of tuple transmission is calculated as:

$$\begin{aligned} g_q = \frac{\sum_{j,k} \boldsymbol{M^q_{jk}}}{\Delta T} &= \frac{\sum_{j,k} \Delta T \cdot ((\boldsymbol{T} \odot \boldsymbol{S})\boldsymbol{X} \odot \boldsymbol{XQ})_{jk}}{\Delta T} \\ &= \sum_{j,k} ((\boldsymbol{T} \odot \boldsymbol{S})\boldsymbol{X} \odot \boldsymbol{XQ})_{jk} \end{aligned} \tag{5}$$

Finally, g_b is calculated as follows:

$$g_b = \sum_{k=1}^{m} (\sum_{j=1}^{n} X_{jk} - \frac{b_k}{\sum_{k=1}^{m} b_k} * n)^2 \tag{6}$$

where b_k is the remaining battery of device k. The effect of this item is intuitive: when two devices posses the same available CPU resources, the computation tasks should be assigned to the one with more remaining battery to prolong the lifetime of the whole system.

2) Genetic Algorithm-based Solution: The aforementioned problem is typically solved by a CPLEX solver [23]. However, based on our experimental results, it takes 77 seconds on average to solve a moderately sized (e.g., 15 nodes and 15 tasks) problem on a desktop, which is impractical for real time applications on mobile platforms. To deal with this issue, we implement an approximation algorithm (Algorithm 1), which returns a near-optimal solution (within 5% of the optimal) in less than 1s for the same problem. Algorithm 1 is based on a "GeneTaskAlloc" procedure that implements the Genetic Algorithm (GA) to solve the optimization problems.

Algorithm 1: ApproxTaskSchedulingAlg()

Input : $\boldsymbol{T}$, $\boldsymbol{D}$, $\boldsymbol{S}$, $\boldsymbol{Q}$, $\boldsymbol{B}$, α ,β, γ, λ
Output: Task schedule matrix $\boldsymbol{X}$

1 **if** $\alpha \neq 0$ **then**
2 $g_d \leftarrow$ Equation 4;
3 $g_d^{max} \leftarrow$ GeneTaskAlloc$(g_d, max).value$;
4 **if** $\beta \neq 0$ **then**
5 $g_q \leftarrow$ Equation 5;
6 $g_q^{max} \leftarrow$ GeneTaskAlloc$(g_q, max).value$;
7 **if** $\gamma \neq 0$ **then**
8 $g_b \leftarrow$ Equation 6;
9 $g_b^{max} \leftarrow$ GeneTaskAlloc$(g_b, max).value$;
10 $F \leftarrow$ Equation 2;
11 $\boldsymbol{X} \leftarrow$ GeneTaskAlloc$(F, min).solution$;
12 return $\boldsymbol{X}$;

Algorithm 1 works as follows. Notice that, F, g_d^{max}, g_q^{max} and g_b^{max} share the same constraints as shown in Equation 3. Therefore, they can be solved by GeneTaskAlloc with different objective functions and goals, i.e. $\max$ or $\min$. We first solve the optimization problem to get g_d^{max}, g_q^{max} and g_b^{max} (line 1 - 9). Then, we use g_d^{max}, g_q^{max} and g_b^{max} to construct the final objective function F (line 10), and call GeneTaskAlloc again to get the final solution (line 11).

The GeneTaskAlloc procedure, which takes a fitness function and an optimization type as input, maintains an iterative process containing the following operations [24]: *SelectParents*, which selects parents from all candidate schedules with the probability proportional to the fitness function value; *GenerateOffspring*, which generates children schedules with parent schedules by uniform crossover; *Mutate*, which chooses a certain number of rows randomly from the schedule matrix according to the mutate rate and changes the position of 1 randomly; *Recombination*, which goes through a schedule matrix row by row and replaces the original schedule with a better one by exchanging adjacent two rows; *FilterOffSpring*, which filters the offspring schedules that do not satisfy the constraints; *SelectPopulations*, which selects a fixed number of populations from the current available schedules according to the fitness function value; *SelectBestSchedules*, which selects the best schedule in terms of the fitness function value. When the iteration times reach the previously set threshold, the procedure will exit with the current best schedule.

3) Reschedule Condition: Sometimes, the new schedules achieve small performance improvement, and switching to them will actually hurt the performance, considering the rescheduling overhead and system stability. To avoid such unnecessary reschedules, we propose the following reschedule conditions:

- It is the first time that the system gets feedback and do reschedule.
- The average end-to-end delay exceeds a threshold τ.
- The input rate of any component i exceeds a threshold times the output, i.e., $\exists i, \sum_{j:c(j)=i} \lambda_j > \sigma * \sum_{j:c(j)=i} \mu_j$.

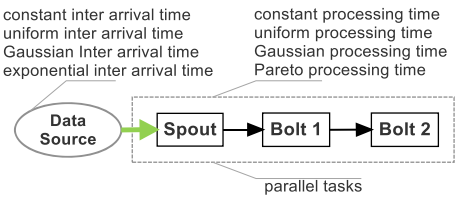


Fig. 7: The benchmark application

- The metric of old schedule exceeds a threshold times the metric of new schedule, i.e., $F(\boldsymbol{X}) > \xi * F(\boldsymbol{X}_{new})$, where F is the objective function in Equation 2.

When any of the above conditions is met, the task reschedule will take place. To guarantee consistent processing, before switching to a new schedule, the spout of the old schedule will stop pulling stream from the data source and the old schedule will continue running for a while until all the remaining tuples in the system are processed.

E. Feedback Based Grouping (FBG)

Except for reporting to Nimbus, each mobile device also reports the execution information (such as task input and output rates, queue lengths, etc.) to the upstream tasks periodically (every Δt). The upstream tasks then direct the output stream tuples to the downstream task with the "Least Expected Waiting Time (LEWT)", which is calculated as follows.

Without loss of generality, we assume task j which belongs to the component i receives the task execution report from task j', which belongs to the downstream component i', at time t_0. Then, at time t which satisfies $t - t_0 < \Delta t$, if task j chooses to send an output stream tuple to j', the "Expected Waiting Time (EWT)" for this tuple at task j' can be calculated by

$$EWT_{j'} = \frac{[(\lambda_{j'} - t_{jj'} - \mu_{j'})(t - t_0) + l_{j'} + \Delta l]^+ + 1}{\mu_{j'}} \quad (7)$$

where $(\lambda_{j'} - t_{jj'})$ is the input rate from other tasks to task j', $\mu_{j'}$ is the processing rate and $l_{j'}$ is the waiting queue length at task j'. Δl is the number of tuples sent to j' from task j in the past $(t - t_0)$ time. Function $[x]^+ = \max(0, x)$. Since we consider the *applications in which tuples have no temporal and spatial relations with each other*, according to our LEWT stream grouping mechanism, an output tuple of j should be sent to task $j' \in A(i')$ that achieves the minimum $EWT_{j'}$.

IV. Performance Evaluation

In this section, we at first briefly introduce the benchmark application for evaluating the system performance. Then, we present the experimental setup and analysis for the evaluation results. Due to the limitation of space, the experiments of real applications are presented in a journal extension in the future.

A. Benchmark Application

In order to thoroughly test the performance of F-MStorm, we developed a benchmark application shown in Fig. 7. The application consists of a data source and three components, i.e., spout, bolt1 and bolt2. To precisely control the input, we let the data source directly generate tuples with the same size and different inter-arrival time (IAT). The IAT can be configured with different distributions, including constant, uniform (UR), Gaussian (GA) and exponential (EP) distributions. The processing time (PT) for each tuple can be configured with different distributions as well, which includes constant, uniform, Gaussian and Pareto (PA) distributions. For the ease of presentation, we denote spout, bolt1, bolt2 by C1, C2, C3 in the rest of the paper.

B. Experimental Setup

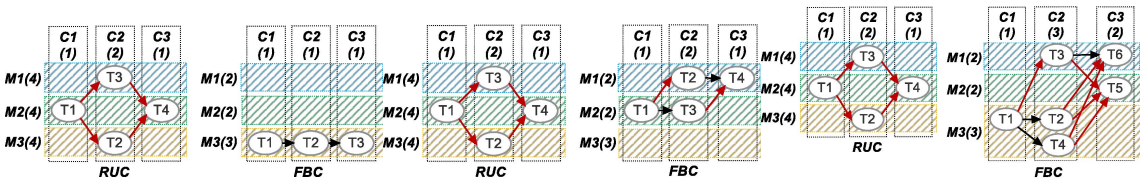
We conduct experiments on three Google Nexus 5 phones running Android 6.0 and a laptop configured as WiFi hotspot. Each Nexus 5 phone has a 4-core CPU and each core is set to run at 1574MHz. All phones are connected to the WiFi hotspot. We run F-MStorm or MStorm on these phones and set system parameters $\Delta T = 15s$, $\Delta t = 5s$, $\delta = 0.4$, $\tau = 2s$, $\sigma = 1.1$ and $\xi = 1.5$. To simulate the resource fluctuation when users invoke other applications during the execution of F-MStorm or MStorm, we developed a resource-intensive disturbance application. We define the light, medium, and heavy disturbance as the scenarios where we run this disturbance application with 10%, 80% and 270% CPU utilization respectively (the total is 400%).

We thoroughly evaluate the system through different types of experiments. First, we run the benchmark application with constant IAT (10T/s) and different constant workloads (defined below) to demonstrate the efficiency of FBC and FBA. We define the light, medium, and heavy constant workload (LCW, MCW, and HCW respectively) as the scenarios where the processing time ratios between C1, C2, and C3 are 1:1:1, 1:15:1, and 1:25:15, respectively. Then, we show the efficiency of FBG by running experiments with constant IAT and MCW, with light, medium, and heavy disturbances respectively. We further evaluate the system's overall performance and compare it with two state-of-the-art solutions (TStorm [19] and RStorm [20] on MStorm) under different constant input speeds, IAT distributions and processing time distributions. Finally, we investigate the overall performance when the CPU frequency of phones decreases due to overheating.

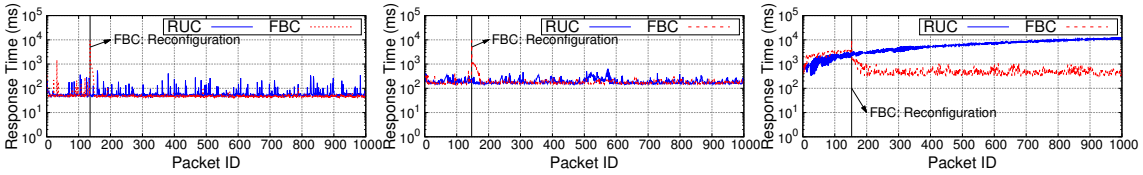
For most experiments, we are interested in the response time (RT, in ms) and throughput (T/s). For FBA experiments specifically, we care about the communication delay (ms) and communication power (mW). In most experiments, we set $\alpha = 0.5$, $\beta = 0.5$, $\gamma = 0$ because delay and energy consumption are more important for us. However, to show that our system also provides configuration for load balance, we run extra FBA experiments with $\alpha = 0.1$, $\beta = 0.1$, $\gamma = 0.8$ and compare its performance with the $\alpha = 0.5$, $\beta = 0.5$, $\gamma = 0$ case.

C. Evaluation Results

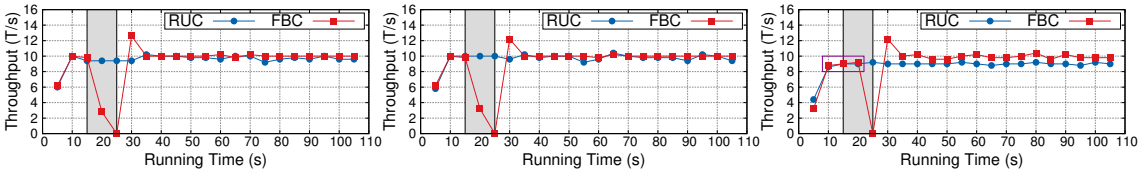
1) The Effects of FBC: Fig. 8 (a)-(c) show the results of both configuration and task assignment when we have the same constant input rate and different workloads. The left side of each figure shows the result of resource-unaware configuration (RUC) and round robin task assignment, while the right side shows the result of FBC and FBA. The number of executors are shown beside each mobile device. With RUC, the number of parallel tasks for C1, C2, C3 are always configured as 1:2:1, regardless of the workload. The number of executors is configured as 4 for all devices, which equals the number of CPU cores. On the other hand, FBC reconfigures the number of parallel tasks for the light workload as 1:1:1 and for the heavy workload as 1:3:2. The number of executors for M1, M2, M3 are reconfigured as 2, 2, 3 based on the feedback.



(a) Config. and task assignment for LCW (b) Config. and task assignment for MCW (c) Config. and task assignment for HCW

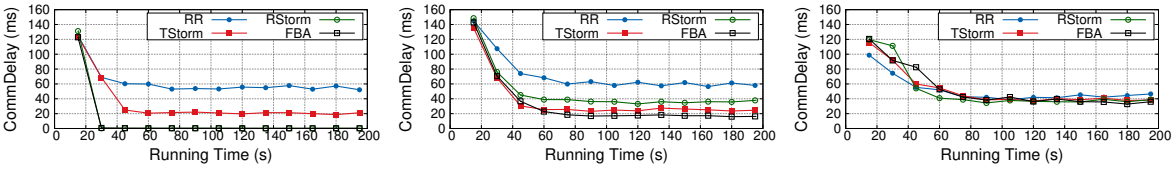


(d) End-to-end response time for LCW (e) End-to-end response time for MCW (f) End-to-end response time for HCW

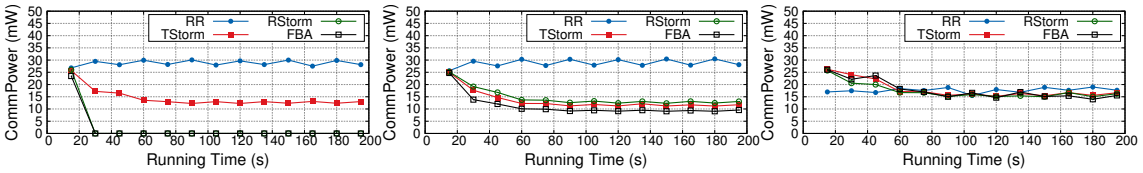


(g) Throughput for LCW (h) Throughput for MCW (i) Throughput for HCW

Fig. 8: Evaluation results for constant inter arrival time and processing time



(a) Communication delay for LCW (b) Communication delay for MCW (c) Communication delay for HCW



(d) Communication power for LCW (e) Communication power for MCW (f) Communication power for HCW

Fig. 9: Evaluation results for round robin and feedback based assignment under different constant workloads

Fig. 8 (d)-(f) show the results of response time. Since FBC reconfigures the number of tasks of LCW as 3, it allows FBA to assign all tasks to a single node M3. This significantly reduces the communication delay and hence the total response time (Fig. 8 (d)). In Fig. 8 (f), FBC increases the number of tasks for C2 and C3 of HCW to 3 and 2 respectively, which eliminates the congestion and reduces the response time.

Fig. 8 (g)-(i) show the results of throughput. For LCW and MCW, both RUC and FBC have throughput equal to the input rate; whereas for HCW, RUC has throughput lower than the input rate because there are not enough parallel tasks for C2 and C3. On the other hand, the throughput of FBC in HCW equals the input speed as the parallel tasks for C2 and C3 are reconfigured. An interesting observation is about MCW, where FBC doesn't change the original parallel tasks, but the FBA algorithm reduces the inter-device traffic by half, just as shown in Fig. 8 (b). However, since the communication delay is much less than the computing delay in MCW, the end-to-end response time only decreases a little bit in Fig. 8 (e).

2) The Effects of FBA: To isolate the effects of task assignment algorithms, when we compare FBA with round robin (RR), TStorm and RStorm, we let RR use the correct parallelism configuration from the beginning, i.e., 1:1:1 for LCW, 1:2:1 for MCW and 1:3:2 for HCW (see Fig. 8 (a) - (c)). In contrast, in FBA, TStorm and RStorm, the parallelism configuration is reconfigured based on the feedback.

Fig. 9 shows the experimental results, where CommDelay is the average end-to-end communication delay and CommPower is the power consumed for transmitting tuples. For LCW, FBA and RStorm achieve much lower communication delay (Fig.

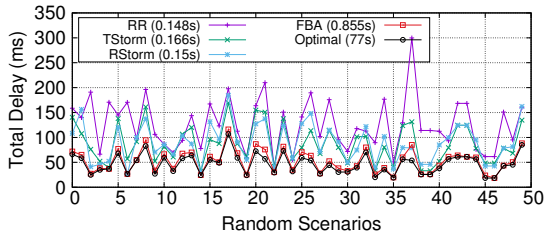


Fig. 10: Comparison of Assignment Algorithms

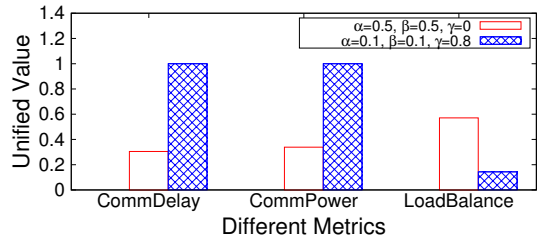


Fig. 11: Comparison of Metrics with different α, β, γ settings

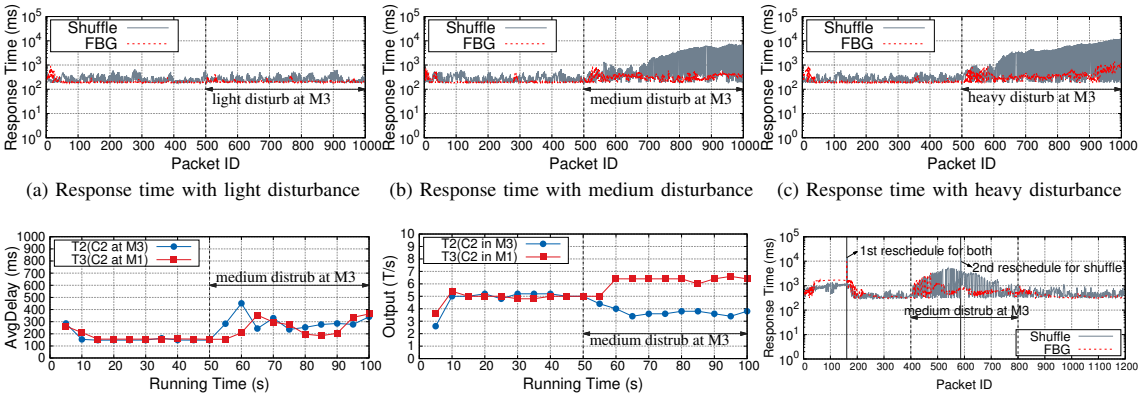
9 (a)) and communication power (Fig. 9 (d)) than RR by assigning all tasks to a single device. Meanwhile, TStorm achieves better performance than RR but worse performance than FBA and RStorm, because its task scheduling algorithm not only tries to reduce the inter-device traffic but also aims at achieving load balance between different nodes. This prevents assigning all tasks to a single node and therefore incurs extra communication delay and energy consumption. For MCW, FBA reduces the communication delay (Fig. 9 (b)) of RR, RStorm and TStorm by 68%, 50% and 26% respectively and the communication power (Fig. 9 (e)) of RR, RStorm and TStorm by 64%, 23% and 17% respectively. This demonstrates the importance of distinguishing different inter-device communication, in terms of traffic size, communication delay and power. As for HCW, FBA achieves similar communication delay and power as TStorm and RStorm while a little bit lower communication delay and power than RR (Fig. 9 (c) and (f)). This is because, for this specific heavy workload, the FBA scheduling algorithm happens to achieve the same schedule as TStorm and RStorm, and a little bit better schedule as RR. It is interesting that, the performance of FBA, TStorm and RStorm is worse than RR until they do rescheduling. This is because, RR utilizes the optimal parallelism at the beginning, while FBA, TStorm and RStorm reschedule to the optimal parallelism based on the feedback by themselves.

In order to throughly compare the performance of FBA, RR, TStorm and RStorm in more general cases, we did 50 extra simulations with different mobile device capacities, application topologies, number of tasks, task-to-task delays and traffic sizes. The scale the experiment is about 15x15, which means 15 tasks are assigned to 15 mobile devices. Fig. 10 shows the experimental results. As we can observe, the performance of FBA is always close to (within 5% of) the optimal schedule, while the performance of TStorm and RStorm is unstable, ranging from near the optimal to more than 3x times worse. In terms of running time, TStorm and RStorm can run as fast as RR (150ms), and our FBA takes about 850ms, while the optimal scheduling takes 77s. Taking both performance and running time into account, our FBA is more practical for a real system deployment.

In order to show that our system provides flexible configuration for load balance, we run extra FBA experiments with MCW and set $\alpha = 0.1$, $\beta = 0.1$, $\gamma = 0.8$. As shown in Fig. 11, a large γ chooses a schedule with better load balance but higher communication delay and power. This is because, if a schedule achieves a good load balance, those tasks are very likely to be assigned to different nodes, which will definitely increase the communication delay and energy consumption.

3) The Effects of FBG: In order to show the effectiveness of FBG, we turn off FBC and FBA. Tasks T1, T2, T3, T4 are assigned to M2, M3, M1, M2 respectively (see Fig. 8 (a) left). We start the disturbance application at 50s on M3. As we can observe from Fig. 12 (a-c): The light disturbance does not impact the system performance. However, the medium and heavy disturbance lead to an increasing response time for Shuffle. This is because T1 completely ignores the disturbance at M3 and still sends the output tuples to T2 and T3 randomly. This causes congestion at T2 and leads to increasing delay. On the other hand, FBG relieves the negative impact of disturbance on the performance. As shown in Fig. 12 (d) and (e), when the disturbance begins, the throughput of T2 begins to decrease and the throughput of T3 begins to increase, while the delay at T2 remains low. This is because T1 sends more output tuples to T3 at M1 after it receives feedback from T2 and T3. We also run experiments with constant IAT and HCW with FBC and FBA turned on. As shown in Fig. 12 (f), when the medium disturbance occurs, the system with Shuffle grouping has to perform rescheduling to achieve low latency while the system with FBG can avoid rescheduling by directing more stream to the tasks running on the under-loaded devices.

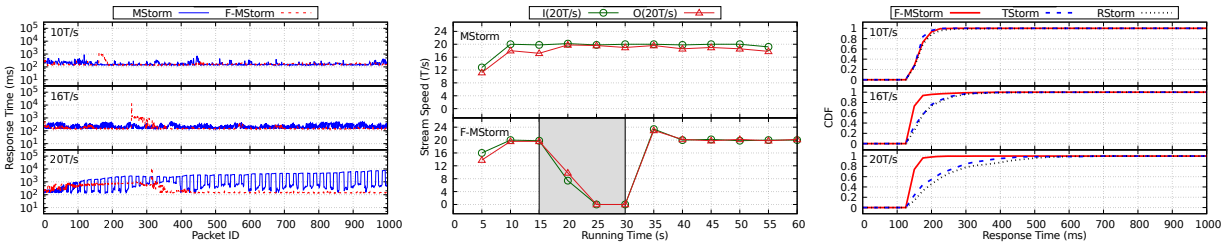
4) Varying Input Speed: Fig. 13 (a)-(c) show the results when we adopt the MCW and use different constant input speeds. Fig. 13 (a) shows the response time. When the input speeds are 10T/s and 16T/s, both MStorm and F-MStorm can achieve a stable low response time. However, when the input speed increases to 20T/s, the response time becomes unstable and keeps increasing. This is because, the input speed exceeds the maximum processing speed and causes congestions at the computation intensive component. In that case, F-MStorm will do reschedule and increase the parallelism for that component, which finally eliminates the congestion and brings the latency back to normal. Fig. 13 (b) shows the results for throughput with constant input at 20T/s. As we can see, MStorm always has a lower output rate than the input rate, while F-MStorm has an output rate equal to the input rate after rescheduling. Fig. 13 (c) shows the CDF of stable response time in F-MStorm, TStorm and RStorm with input speeds equal to 10T/s, 16T/s and 20T/s respectively. When the input speed is 10T/s, the response time of F-MStorm, TStorm and RStorm are similar with each other. This is because they adopt similar task assignment, and have similar inter-devices communication delay. However, when the input speed increases to 16T/s, the latency of TStorm and RStorm can be up to 1.5x latency of F-MStorm. And when the input speed increases to 20T/s, the advantage of F-MStorm becomes more obvious, which can be up to 3x faster than TStorm and RStorm. This is because, as the input speed increases, there are more tasks after rescheduling. The inter-device communication among different



(a) Response time with light disturbance (b) Response time with medium disturbance (c) Response time with heavy disturbance

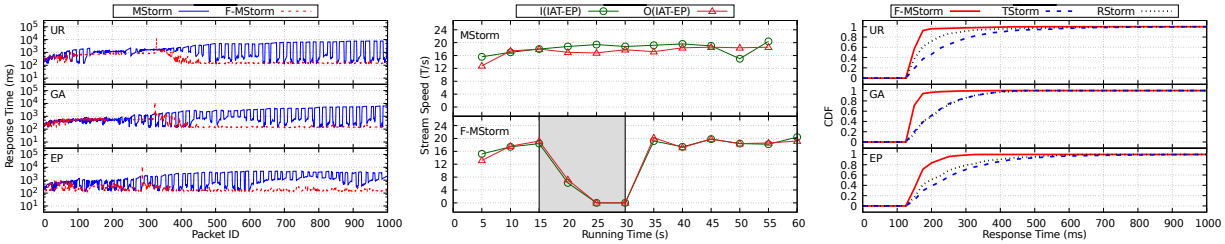
(d) Dealy of C2 with medium disturbance (e) Output of C2 with medium disturbance (f) FBG reduces rescheduling frequency

Fig. 12: Shuffle and feedback based stream grouping for different disturbance



(a) RT for Diff. Input Speeds (b) Input/output for Diff. Input Speeds (c) RT CDF comparison for Diff. Input Speeds

Fig. 13: Evaluation results when we vary the constant input speeds

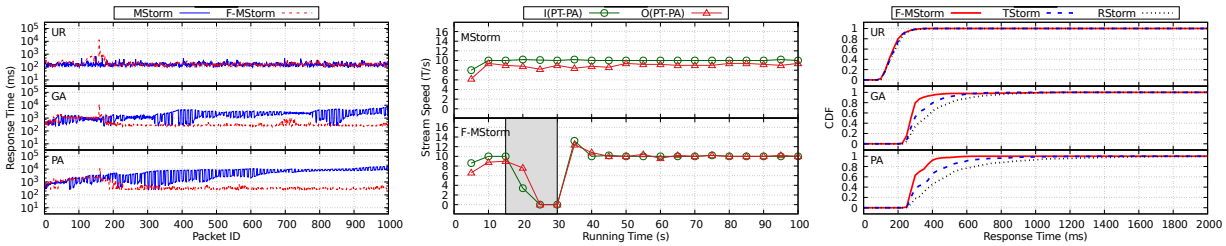


(a) RT for Input with Diff. IAT (b) Input/output for Input with Exp. IAT (c) RT CDF comparison for input with Diff. IAT

Fig. 14: Evaluation results when we vary the IAT of input

tasks will increase and the advantages of F-MStorm, which takes inter-device communication diversity into account, will become more obvious.

5) Varying Inter Arrival Time: Fig. 14 (a)-(c) show the results when we adopt the MCW and vary the IAT distribution. Fig. 14 (a) shows the response time. In MStorm, regardless of the IAT distribution, the response time increases with the running time. However, in F-MStorm, although the response time is increasing at the beginning, it goes back to be low after rescheduling. This is because F-MStorm increases the parallelism of the computing-intensive component to eliminate the congestion. The results in Fig. 14 (b) prove this claim, where the IAT distribution is exponential. MStorm always has a 1T/s lower output rate than the input rate, while F-MStorm has an output rate equal to the input rate after rescheduling. It should be noted that, although the throughputs of MStorm and F-MStorm seem similar, there is a fundamental difference: a congestion happens in MStorm, while no congestion happens after rescheduling in F-MStorm. The delays of MStorm and F-MStorm in Fig. 14 (a) clearly reflects this phenomenon. Due to space limitation, we omit the throughput results for other distributions because they are similar. Fig. 14 (c) shows the CDF of the stable response time in F-MStorm, TStorm and RStorm with uniform, Gaussian and exponential IAT distribution, respectively. With uniform IAT distribution, the response time of RStorm can be up to 1/3 shorter than that of TStorm but still up to 2x that of F-MStorm. With Gaussian IAT distribution, the response time in TStorm and RStorm are similar with each other, but they are both up to 2x that of F-MStorm. With exponential IAT distribution, the response



(a) RT for Input with Diff. PT (b) Input/output for Input with Pareto PT (c) RT CDF comparison for Input with Diff. PT

Fig. 15: Evaluation results when we vary the PT of input

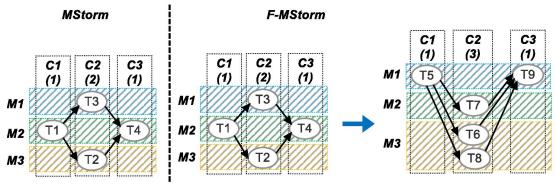


Fig. 16: Sample example for CPU frequency decrease

time of F-MStorm becomes longer. However, it is up to 1/2 shorter than that of TStorm and RStorm. The mechanism behind is that: more tasks and inter-device communication exist in the Gaussian and exponential IAT distribution cases, so the advantages of F-MStorm, which takes inter-device communication diversity into account becomes more obvious.

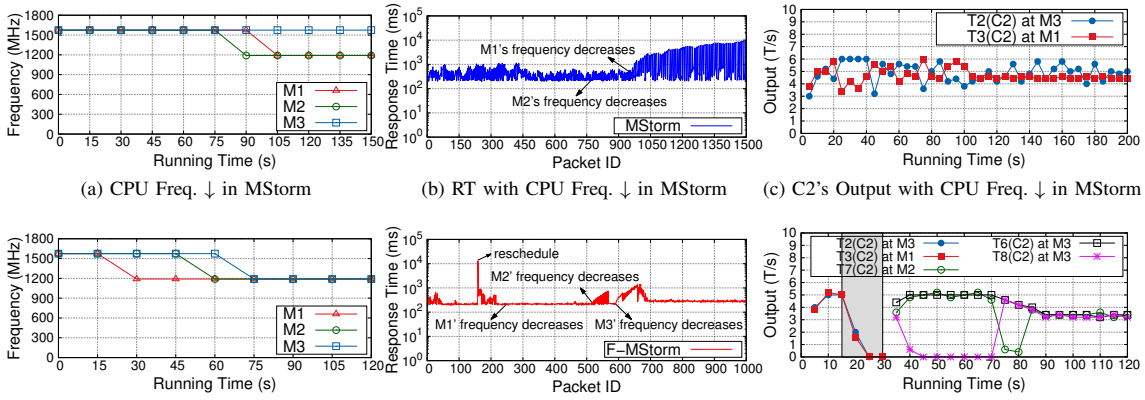
6) Varying Processing Time: Fig. 15 (a)-(c) show the results when we adopt the constant input rate (10T/s) and vary the processing time. Fig. 15 (a) shows the response time. As we can see, with uniform distribution of processing time, both MStorm and F-MStorm achieve stable and low latency. However, with Gaussian and Pareto distribution of processing time, F-MStorm achieves much lower and stable response time than MStorm, because F-MStorm increases the parallelism of the computing-intensive component, which eliminates the congestion. Fig. 15 (b) shows the results for throughput with Pareto distribution. MStorm always achieves 10% lower output rate than the input rate, while F-MStorm achieves an output rate that is the same as the input rate after rescheduling. Due to space limitation, we omit the throughput results for other distributions because they are similar. Fig. 15 (c) shows the CDF of the stable response time in F-MStorm, TStorm and RStorm with uniform, Gaussian and Pareto distribution of processing time respectively. With uniform distribution of processing time, the response time of F-MStorm, TStorm and RStorm are similar with each other. However, with Gaussian distribution of processing time, TStorm achieves up to 1/4 shorter response time than RStorm but 1/5 longer response time than F-MStorm. With Pareto distribution of processing time, the advantage of F-MStorm becomes more obvious. The mechanism behind is that: more tasks and inter-device communication exist in the Gaussian and Pareto distribution of processing time cases, so the advantages of F-MStorm, which takes inter-device communication diversity into account becomes more obvious.

7) CPU Frequency Decrease: The CPU overheating protection mechanism on Android Phones will reduce the CPU frequency when its temperature goes too high [25]. The CPU frequency of smart phones we use might drop from 1574MHz to 1190MHz when it is overheating. To compare MStorm and F-MStorm in this situation, we first run the resource-intensive application for a while such that the CPU temperature gets high. Then, we start MStorm/F-MStorm and the benchmark application with constant IAT and workload with processing time ratio 1:25:1. The tasks are assigned to mobile devices as shown in Fig. 16. In F-MStorm, an initial rescheduling happens before the CPU frequency drops. The tasks of C1 change from T1 to T5; the tasks of C2 change from T2, T3 to T6, T7, T8; and the tasks of C3 change from C4 to C9.

Fig. 17 (a) and (d) show the time when the CPU frequency of mobile phones drops in MStorm and F-MStorm respectively. We are interested in how the systems may react to it. Fig. 17 (b) and (c) show the results for MStorm. When the CPU frequency of M2, which runs T1 and T4 (light tasks), drops, the end-to-end response time is not impacted. However, when the CPU frequency of M1, which runs T3 (heavy task), drops, the end-to-end response time increases instantly and the total output rate of C2 (9.5T/s) becomes lower than its input rate (10T/s). On the other hand, Fig. 17 (e) and (f) shows the results for F-MStorm. In F-MStorm, when the CPU frequency of M1, which runs T5 and T9 (light tasks), drops, the end-to-end response time is not impacted. When the CPU frequency of M2 and M3, which run T6, T7 and T8 (heavy tasks), drops, the end-to-end response time only increases a little bit at the beginning, but soon returns to normal. This is due to the fact that, when the CPU frequency of M2 drops, T5 directs more stream tuples to T8; and when the CPU frequency of M3 drops, T5 directs more stream tuples to T7. The total output rate of C2 keeps at 10T/s.

V. Related Work

Computation Offloading and Edge Computing: Computation offloading [8]–[10], [14], [15], [26]–[30] has been a popular research area. Based on the offloading destinations, the existing work can be categorized into cloudlet offloading, cloud offloading and hybrid offloading. Cloudlet offloading is named as edge computing [31]–[34] as well, which aims at reducing the application response time and saving the Internet communication bandwidth by taking the control of computing applications, data, and services away from some central nodes ("core") to the logical extreme ("edge") of the Internet [35]. CloneCloud [27], JustInTime [28] and Odessa [9] are cloudlet offloading systems that leverage virtual machine migration mechanisms to reduce the application response time. However,



(a) CPU Freq. ↓ in MStorm (b) RT with CPU Freq. ↓ in MStorm (c) C2's Output with CPU Freq. ↓ in MStorm

(d) CPU Freq. ↓ in F-MStorm (e) RT with CPU Freq. ↓ in F-MStorm (f) C2's Output with CPU Freq. ↓ in F-MStorm

Fig. 17: How MStorm and F-MStorm deal with CPU frequency decrease

all these systems rely on powerful nearby cloudlets, which are not always available in some critical scenarios like military operations and disaster response, because soldiers or first responders have to move from place to place and they are only allowed to carry on some lightweight mobile devices. Different from the above systems, *F-MStorm only utilizes nearby mobile devices as offloading destinations*, which is more practical for the military and disaster response scenarios, because soldiers and first responders always work together as a group and their mobile devices can be connected together as a cluster. MAUI [26] is a cloud offloading system which enables energy-aware offloading by using remote function calls to the cloud. MCDNN [8] is a cloud offloading framework that employs a runtime scheduler to trade off the application accuracy for resource usage and latency. Orbit [30] and LEO [10] are similar systems that utilize profile-based partitioning of applications to offload computation tasks to hybrid computing resources. All these systems rely on the Internet access, which however are not always available in some critical scenarios, because soldiers and first responders always work in some extremely difficult environments that do not provide any access to the Internet. Different from these systems, *F-MStorm does not rely on the Internet*. Instead, it only needs a mobile device to be set up as a hotspot, such that all the mobile devices in a group can be connected together. Some existing work like Hyrax [14] and Serendipity [15] also offload computing tasks to the nearby mobile devices. However, their work focuses on processing bounded batch jobs that have relatively low requirements on the latency. Instead, *F-MStorm focuses on processing unbounded stream data, which has a higher requirement on the latency.*

Distributed Stream Processing System: Apache Storm [11] is a distributed stream processing system deployed on cloud servers. Many improvements based on Storm have been proposed [19], [20], [36]–[39]. Among them, AdaptiveStorm [36] continuously monitors the system performance and reschedules tasks at run-time to reduce the overall response time. T-Storm [19] accelerates stream processing by using traffic-aware scheduling, which minimizes the inter-device and inter-process traffics. R-Storm [20] improves the throughput and minimizes the network latency by maximizing the resource utilization. *All these works are closely related to F-MStorm, but they do not consider the detailed differences among inter-device links, which however are essential in stream processing at the edge.* The authors in [37] propose a scalable centralized scheme for job reconfiguration, which minimizes the communication cost while keeping the nodes below a computational load threshold. The authors in [38] propose a dynamic resource scheduler for cloud-based distributed stream processing systems, which measures the system workload with the minimal overhead and provisions the minimum resources to meet the response time constraints. The authors in [39] provide a general formulation for the optimal data stream processing placement and takes explicitly into account the heterogeneity of computing and networking resources. Similar to these works, in F-MStorm, we propose a general framework for stream task assignment that takes delay, energy and load balance all into account. *The difference is that, our problem is more challenging, because in stream processing at the edge, the inter-device delay is dynamic and the users' own application may cause disturbance to F-MStorm.* Except for Storm and its subsequent improvements, there are other popular stream processing systems like Apache Spark Stream [12], Flink [13], Samza [40], etc. However, *all these systems are designed and implemented to run on cloud servers instead of mobile devices.* To the best of our knowledge, MStorm [1] is the first work that performs online distributed mobile stream processing at the edge. However, it is inefficient in the system configuration, task scheduling and execution aspects. *F-MStorm improves its efficiency by using the feedback information.*

VI. Conclusion and Future Work

In this paper, we present a feedback-based online distributed mobile stream processing system running at the edge called F-MStorm. F-MStorm adopts feedback-based approach at many system design levels including configuration, scheduling and execution. We implement F-MStorm on a cluster of Android phones and evaluate its performance through extensive experiments. The experimental results show that F-MStorm achieves up to 3x shorter response time, 10% higher throughput and 23% less energy consumption than the state-of-the-art mobile stream processing systems. It should be noted that, currently,

we only focus on the CPU bounded applications and assume all mobile devices to be homogeneous in hardware. We also assume that the users in a group always stay in the wireless range of each other. In the future, we will consider a heterogeneous computing environment, take into account other resource constraints like memory usage or network bandwidth and deal with the node failure case.

ACKNOWLEDGMENT

This material is based upon work supported by National Institute of Standards and Technology (NIST) under Grant NO. (#70NANB17H190). We also appreciate the suggestions from the reviewers and our shepherd Dr. Eyal de Lara.

REFERENCES

[1] Q. Ning, C.-A. Chen, R. Stoleru, and C. Chen, "Mobile storm: Distributed real-time stream processing for mobile clouds," in *CloudNet '15*. IEEE, pp. 139–145.

[2] D. M. Chen, S. S. Tsai, R. Vedantham, R. Grzeszczuk, and B. Girod, "Streaming mobile augmented reality on mobile phones," in *ISMAR '09*. IEEE, pp. 181–182.

[3] H. Lu, D. Frauendorfer, M. Rabbi, M. S. Mast, G. T. Chittaranjan, A. T. Campbell, D. Gatica-Perez, and T. Choudhury, "Stresssense: Detecting stress in unconstrained acoustic environments using smartphones," in *UbiComp '12*. ACM, pp. 351–360.

[4] Y. Lee, C. Min, C. Hwang, J. Lee, I. Hwang, Y. Ju, C. Yoo, M. Moon, U. Lee, and J. Song, "Sociophone: Everyday face-to-face interaction monitoring platform using multi-phone sensor fusion," in *Moibsys '13*. ACM, pp. 375–388.

[5] G. Chen, C. Parada, and G. Heigold, "Small-footprint keyword spotting using deep neural networks," in *ICASSP '14*. IEEE, pp. 4087–4091.

[6] N. D. Lane, P. Georgiev, and L. Qendro, "Deepear: Robust smartphone audio sensing in unconstrained acoustic environments using deep learning," in *UbiComp '15*, pp. 283–294.

[7] I. Damian, C. S. S. Tan, T. Baur, J. Schöning, K. Luyten, and E. André, "Augmenting social interactions: Realtime behavioural feedback using social signal processing techniques," in *CHI '15*, 2015, pp. 565–574.

[8] S. Han, H. Shen, M. Philipose, S. Agarwal, A. Wolman, and A. Krishnamurthy, "Mcdnn: An approximation-based execution framework for deep stream processing under resource constraints," in *Mobisys '16*. ACM, pp. 123–136.

[9] M.-R. Ra, A. Sheth, L. Mummert, P. Pillai, D. Wetherall, and R. Govindan, "Odessa: enabling interactive perception applications on mobile devices," in *Mobisys '11*. ACM, pp. 43–56.

[10] P. Georgiev, N. D. Lane, K. K. Rachuri, and C. Mascolo, "Leo: scheduling sensor inference algorithms across heterogeneous mobile processors and network resources," in *Mobicom '16*, pp. 320–333.

[11] A. Toshniwal, S. Taneja, A. Shukla, K. Ramasamy, J. M. Patel, S. Kulkarni, J. Jackson, K. Gade, M. Fu, J. Donham *et al.*, "Storm@ twitter," in *SIGMOD '14*. ACM, pp. 147–156.

[12] M. Zaharia, T. Das, H. Li, S. Shenker, and I. Stoica, "Discretized streams: An efficient and fault-tolerant model for stream processing on large clusters." *HotCloud '12*, vol. 12, pp. 10–10.

[13] P. Carbone, A. Katsifodimos, S. Ewen, V. Markl, S. Haridi, and K. Tzoumas, "Apache flink: Stream and batch processing in a single engine," *Bulletin of the IEEE Computer Society Technical Committee on Data Engineering*, vol. 36, no. 4, 2015.

[14] E. E. Marinelli, "Hyrax: cloud computing on mobile devices using mapreduce," DTIC Document, Tech. Rep., 2009.

[15] C. Shi, V. Lakafosis, M. H. Ammar, and E. W. Zegura, "Serendipity: enabling remote computing among intermittently connected mobile devices," in *Mobihoc '12*. ACM, pp. 145–154.

[16] J. Tang and T. Q. Quek, "The role of cloud computing in content-centric mobile networking," *IEEE Communications Magazine*, vol. 54, no. 8, pp. 52–59, 2016.

[17] J. Dean and S. Ghemawat, "Mapreduce: a flexible data processing tool," *Communications of the ACM*, vol. 53, no. 1, pp. 72–77, 2010.

[18] S. Venkataraman, A. Panda, K. Ousterhout, M. Armbrust, A. Ghodsi, M. J. Franklin, B. Recht, and I. Stoica, "Drizzle: Fast and adaptable stream processing at scale," in *Proceedings of the 26th Symposium on Operating Systems Principles*. ACM, 2017, pp. 374–389.

[19] J. Xu, Z. Chen, J. Tang, and S. Su, "T-storm: Traffic-aware online scheduling in storm," in *ICDCS '14*. IEEE, pp. 535–544.

[20] B. Peng, M. Hosseini, Z. Hong, R. Farivar, and R. Campbell, "R-storm: Resource-aware scheduling in storm," in *Middleware '15*, pp. 149–161.

[21] L. Sun, R. K. Sheshadri, W. Zheng, and D. Koutsonikolas, "Modeling wifi active power/energy consumption in smartphones," in *ICDCS '14*. IEEE, pp. 41–51.

[22] "Network-Connection," https://github.com/facebook/network-connection-class, 2016, [Online; accessed 14-Dec-2016].

[23] "API: CPLEX," https://www.ibm.com/, 2016, [Online; accessed 14-Dec-2016].

[24] K. Deb, A. Pratap, S. Agarwal, and T. Meyarivan, "A fast and elitist multiobjective genetic algorithm: Nsga-ii," *IEEE transactions on evolutionary computation*, vol. 6, no. 2, pp. 182–197, 2002.

[25] G. P. Srinivasa, R. Begum, S. Haseley, M. Hempstead, and G. Challen, "Separated by birth: Hidden differences between seemingly-identical smartphone cpus," in *HotMobile '17*. ACM, pp. 103–108.

[26] E. Cuervo, A. Balasubramanian, D.-k. Cho, A. Wolman, S. Saroiu, R. Chandra, and P. Bahl, "Maui: Making smartphones last longer with code offload," in *MobiSys '10*. ACM, pp. 49–62.

[27] B.-G. Chun, S. Ihm, P. Maniatis, M. Naik, and A. Patti, "Clonecloud: elastic execution between mobile device and cloud," in *Proceedings of the sixth conference on Computer systems*. ACM, 2011, pp. 301–314.

[28] K. Ha, P. Pillai, W. Richter, Y. Abe, and M. Satyanarayanan, "Just-in-time provisioning for cyber foraging," in *MobiSys '13*, pp. 153–166.

[29] P. Georgiev, N. D. Lane, K. K. Rachuri, and C. Mascolo, "Dsp. ear: Leveraging co-processor support for continuous audio sensing on smartphones," in *SenSys '14*. ACM, pp. 295–309.

[30] M.-M. Moazzami, D. E. Phillips, R. Tan, and G. Xing, "Orbit: a smartphone-based platform for data-intensive embedded sensing applications," in *IPSN '15*. ACM, pp. 83–94.

[31] M. Satyanarayanan, "The emergence of edge computing," *Computer*, vol. 50, no. 1, pp. 30–39, 2017.

[32] K. Ha, Y. Abe, T. Eiszler, Z. Chen, W. Hu, B. Amos, R. Upadhyaya, P. Pillai, and M. Satyanarayanan, "You can teach elephants to dance: agile vm handoff for edge computing," in *SEC '17*. ACM, p. 12.

[33] L. Chaufournier, P. Sharma, F. Le, E. Nahum, P. Shenoy, and D. Towsley, "Fast transparent virtual machine migration in distributed edge clouds," in *SEC '17*. ACM, 2017, p. 10.

[34] Z. Chen, W. Hu, J. Wang, S. Zhao, B. Amos, G. Wu, K. Ha, K. Elgazzar, P. Pillai, R. Klatzky *et al.*, "An empirical study of latency in an emerging class of edge computing applications for wearable cognitive assistance," in *Proceedings of the Second ACM/IEEE Symposium on Edge Computing*. ACM, 2017, p. 14.

[35] P. Garcia Lopez, A. Montresor, D. Epema, A. Datta, T. Higashino, A. Iamnitchi, M. Barcellos, P. Felber, and E. Riviere, "Edge-centric computing: Vision and challenges," *SIGCOMM Computer Communication Review*, vol. 45, no. 5, pp. 37–42, 2015.

[36] L. Aniello, R. Baldoni, and L. Querzoni, "Adaptive online scheduling in storm," in *DEBS '13*. ACM, pp. 207–218.

[37] A. Chatzistergiou and S. D. Viglas, "Fast heuristics for near-optimal task allocation in data stream processing over clusters," in *CIKM '14*. ACM, pp. 1579–1588.

[38] T. Z. Fu, J. Ding, R. T. Ma, M. Winslett, Y. Yang, and Z. Zhang, "Drs: dynamic resource scheduling for real-time analytics over fast streams," in *ICDCS '15*. IEEE, pp. 411–420.

[39] V. Cardellini, V. Grassi, F. Lo Presti, and M. Nardelli, "Optimal operator placement for distributed stream processing applications," in *DEBS '16*. ACM, pp. 69–80.

[40] "Apache Samza," https://samza.apache.org/, 2017, [Online; accessed 14-Dec-2017].