Kea: A Computation Offloading System for Smartphone Sensor Data

Das, Roshan Bharath; Bozdog, Nicolae Vladimir; Makkes, Marc X.; Bal, Henri

published in
2017 IEEE 9th International Conference on Cloud Computing Technology and Science, CloudCom 2017
2017

DOI (link to publisher)
10.1109/CloudCom.2017.33

document version
Publisher's PDF, also known as Version of record

Kea: A Computation Offloading System for Smartphone Sensor Data

Roshan Bharath Das, Nicolae Vladimir Bozdog, Marc X. Makkes, Henri Bal
Department of Computer Science
Vrije Universiteit,
Amsterdam, The Netherlands
{r.bharathdas, n.v.bozdog, m.x.makkes, h.e.bal}@vu.nl

***Abstract*—Nowadays smartphones are equipped with many sensors which applications can continuously invoke to acquire real-time sensor information, such as GPS tracking. Due to the resource-constrained nature of the smartphones, it is often beneficial if the processing of the sensor data is offloaded to a remote resource. However, the decision to offload the computation depends on a multitude of factors such as the hardware capabilities of the phone, the communication energy and latency and the characteristics of the stream computations, e.g., window size, sensor frequency and operational complexity.**

In this paper we introduce Kea, a profiling-based computation offloading system that automatically decides whether offloading is beneficial for smartphones. The decision making is based on two criteria: the power consumption of the application and the elapsed time for processing the sensor data. Our evaluation results show that unexpected factors such as CPU frequency scaling and the network state also influence the decision-making process. In addition, we show that Kea's profiling overhead is negligible.

I. Introduction

Offloading computations from a smartphone to a cloud is a widely studied research topic. The combination of smartphones and clouds is particularly attractive because of their complementary features: smartphones are context-aware, smart, and mobile, but, unlike clouds, they are heavily resource-constrained. A recent PhD thesis [1] covered no less than 58 existing studies in this area. Many mechanisms exist for offloading processes, functions, tasks, or services to remote or proximate computing resources and many decision-making strategies have been proposed to determine when to compute on the phone and when to offload.

Surprisingly, however, hardly any research has been done on offloading of *sensor processing and streaming* computations, even though smartphones now have numerous sensors and are highly popular for these types of applications [2], [3]. Of the 58 studies in [1] only the work of [4] looks at offloading for stream applications. Many other smartphone sensor platforms just do all processing on the phone [2] or always send all data to a cloud [3], none of which will always be optimal. Offloading computations for stream applications differs from other applications:

- The processing has to be performed continuously, so it is even more important to take *energy consumption* on the phone into account (which is ignored in [4]) when deciding whether to offload.
- Many sensor applications are *interactive* and thus latency-sensitive, especially in domains like eHealth (real-time coaching), traffic, environmental monitoring, and safety. In these cases, an optimal trade-off needs to be found between the communication costs to the remote resource and the slow computation on the phone. Here, the rise of *edge computing* [5] is important, as more resources (cloudlets) may become available closer to the phones.
- The computations performed on the sensor streams often are simpler and more regular than for other applications that use offloading (like image analysis, sound analysis, AI games, etc.).

The goal of this paper is to better understand the trade-offs for deciding when to offload sensor computations to a cloud or edge resource, aiming to minimize energy usage and/or latency. Factors we will look at include the hardware capabilities of the phone, the type and latency of the communication network to the cloud, and characteristics of the stream computations (window size, sensor frequency, operation complexity). To study these factors, we built a simple profiling-based system called Kea that can automatically decide where to do the computation, given the architecture at hand. The results show that several factors behave as expected, while other factors make the decisions much more challenging, especially the interference with dynamic frequency scaling and the network state.

The contributions of this paper are as follows:

- We show how the decision to compute locally (on the phone) or remotely (in the cloudlet/cloud) changes based on the characteristics of the applications, the type of hardware used and the communication latency between the phone and the remote resource.
- We introduce a novel system that uses a profiling-based approach to help in automatic decision making on where to do the computation for sensor data based on power consumption and elapsed time.

The remainder of the paper is organized as follows. In Section II we discuss the related work. The Kea system is described in Section III. In Section IV we evaluate Kea using various parameters to show the importance of doing the computation both locally on the phone and remotely in the cloudlet or the cloud. The paper concludes in Section V.

II. Related Work

Offloading computation from mobile to other platforms has been heavily studied in the past decade [1]. While most of this research is focused on cloud offloading, recent development in the area of IoT made it possible to offload computation to more proximate platforms, like cloudlets or tablets. In this section we present previous work addressing the two most common types of platforms suitable for mobile computation offloading: cloud and edge.

Cloud Offloading. The high convenience and low cost makes cloud the most used platform for mobile computation offloading. There is a plethora of research on cloud offloading addressing a large variety of computational problems, like image processing, sound recognition, or sensor data processing. Other solutions are less tailored for a specific problem and let the programmer adapt their capabilities to his particular needs. Among the latter, we mention our Cuckoo framework [2], which takes advantage of the Java language applicability in both Android and server programming. With Cuckoo, parts of the mobile application can be offloaded to the cloud according to a cost function that takes into account the energy usage and latency of the code. While Cuckoo makes it more efficient to run highly computational tasks, like gaming or image recognition in the mobile device, it is less suitable for processing sensor data streams, as shown in [6].

In order to exploit cloud capabilities even further, some cloud offloading systems make use of "clones" of the mobile devices, which are cloud-based virtual machines that can take some of the computational burden from their mobile counterparts. CloneCloud [7] and Moitree [8] are notable systems that implement this paradigm. While this approach makes the transfer of executables from mobile to cloud more seamless, it might suffer from scalability issues if the number of mobile devices that have to be emulated is high.

Finally, we note some offloading systems that are tailored to particular use cases. Neurosurgeon [9] leverages the layered structure of deep neural network applications by splitting the computation of different layers between mobile and cloud. SociableSense [10] analyzes social interactions between coworkers and uses a multi objective decision system to choose whether to perform computation locally or in the cloud. Of particular interest is the work by Yang et al. [4], which addresses the problem of partitioning data stream applications between mobile and cloud to increase performance. While their solution aims for high performance and elasticity, we take a step further by also taking into account the energy usage of the mobile device when deciding whether or not to offload.

Edge Offloading. Edge computing was recently introduced in an attempt to reduce the high latencies of faraway clouds and also to improve scalability [5]. The usefulness of offloading mobile computation to edge devices was proved by systems addressing different types of real-time problems, like improving the quality of video streaming [11] or monitoring the context of the user [12]. As opposed to cloud computing, where the computation is executed in high performance datacenters, edge computing allows for computation to be done on a large variety of networked devices, like WiFi Access Points [13], cloudlets [12] or ad hoc networks of smartphones [14][15].

The research presented in this paper complements the above list with our Kea system. While edge- and cloud-assisted sensor processing systems have been studied in the past [16], to our knowledge, Kea is the first system for offloading the processing of sensor data streams that autonomously decides where to perform the computation.

III. Kea System

In this section we describe the architecture of the Kea system. Kea helps in automatic decision making for context-aware applications that use sensor data. Our system works in two steps: (1) it initially profiles the application based on the power consumption and the elapsed time for both local and remote computations (2) it makes the decision based on the weights (preferences) provided by the user.

Many smartphone sensors such as accelerometer, gyroscope, sound etc. generate data at a constant frequency. In the current system, we assume that the frequency of the sensor data and the communication latency between the phone and the remote resource are constant. We also assume that the operations are not data-dependent. With these assumptions, Kea only profiles once in the initial phase and afterwards it makes the offloading decision. In Section IV-B the overhead costs of generalizing this to more frequent (or even continuous) profiling is estimated. Finally, we assume that the code for processing the sensor data can be executed locally (on the phone) and remotely (in the cloudlet or cloud). The code offloading can be easily performed using various existing systems [1] such as our Cuckoo [17] system. Our focus is particularly on sensor data offloading based on the current architecture. Kea consists of six different components: 1) Input module 2) Sensor service 3) Decision engine 4) Local evaluation engine 5) Remote evaluation manager and 6) Remote evaluation engine, as shown in Fig. 1. Below, we describe each of the components in detail.

A. Input module

The system takes as input a function and a set of configuration parameters. The input function is applied by the evaluation engine to evaluate the incoming sensor data. The system has predefined functions such as moving average, median, maximum, minimum etc. Kea also supports adding new functions. The input configuration contains various parameters as described below:

- SENSOR_NAME - The name of the sensor that is used.
- SENSOR_VALUEPATH - The type of data that needs to be gathered from the sensor (e.g., altitude from GPS).
- SENSOR_DELAY - The frequency at which the sensor data is generated. A delay of 1 second implies that new sensor data is generated every second.
- WINDOW - The number of data elements on which the input function is applied. For example, a value of 10

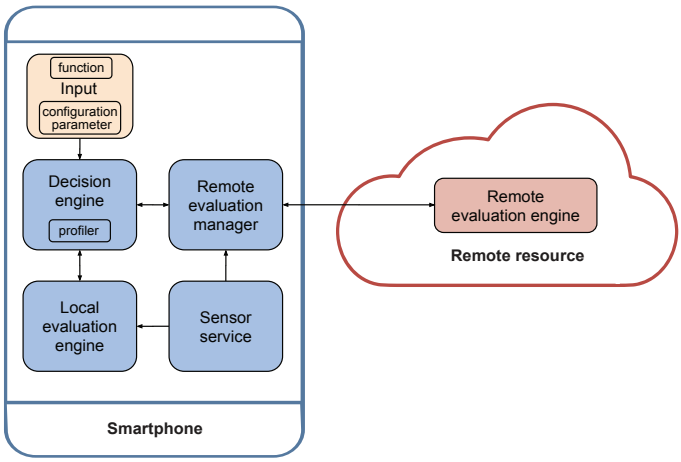

Fig. 1: Kea architecture

means that the function would use the newly generated 10 elements for processing.

- REMOTE_URL - The URL of the remote resource where the computation can be offloaded.
- POWER_WEIGHT - The weight for energy measurement used by the decision engine.
- ELAPSED_TIME_WEIGHT - The weight for computation time measurement. The total weight must be equal to 1. In case of equal preference, both POWER_WEIGHT and ELAPSED_TIME_WEIGHT are 0.5.

B. Sensor service

The sensor service is responsible for gathering data from various sensor sources (such as accelerometer, gyroscope, GPS, light, sound). We use our existing work (SWAN) to collect the data [2]. The sensor data is gathered based on the input configuration parameters such as SENSOR_DELAY. The newly generated sensor data is stored locally on the phone and fetched by other modules for evaluation.

C. Decision engine

The decision engine contains two main components: the profiler and the decision maker. The profiler measures the elapsed time and the power consumption over a period of time. The measurement is done for the input function that runs both locally on the phone and remotely in the cloudlet/cloud. For the offloading case, we measure (1) the total round trip time (elapsed time) for sending the new sensor value to the cloud, computing the function on the new window in the cloud, and receiving the result on the phone. (2) The power consumption on the phone for (1). The elapsed time is computed as an average over 5 evaluations and does not consume significant energy. The power consumption is measured using the Trepn profiler [18]. The profiler has to run for 2 minutes for an accurate power consumption measurement as recommended by the tool administrator.

After profiling, the decision maker collects the results and, based on the input weights, the decision is made on where to do the computation. The decision maker runs only once based on the elapsed time and the power consumption measured by the profiler. For the decision-making process, we use a multi-attribute decision analysis method called weighted product model [19]. We chose this method as it performs dimensionless analysis. It implies that the system does not need to normalize the data gathered from multiple attributes that contain multiple units (such as mW for power consumption, ms for elapsed time) of measurement. In general, in order to compare the alternatives A_K and A_L, the following product has to be calculated:

$$R(A_K/A_L) = \prod_{j=1}^{N} (a_{Ki}/a_{Li})^{W_j} \quad (1)$$

where N is the number of criteria, a_{ij} is the actual value of the $i-th$ alternative in terms of the $j-th$ criterion, and W_j is the weight of importance of the $j-th$ criterion. In our case, the two alternatives are the phone (A_K) and the remote resource (cloudlet or cloud) (A_L) and the criteria are the power consumption and the elapsed time. We chose power consumption as an attribute because extended battery life is critical for smartphone applications. We also note that local computation that involves CPU usage (computation) and remote computation that involves network communication (WiFi, 4G) for data transfer both reflect in power consumption. Elapsed time is also important for sensor-based applications.

D. Local evaluation engine

The local evaluation engine is responsible for the computation on the input function. Upon receiving the request from the decision engine, the local evaluation engine will start evaluating the input function. It pulls the sensor data from the sensor service and adds it to a window. The window is formed of a circular buffer. The input function then evaluates the data in the window based on the frequency at which the sensor data is generated.

E. Remote evaluation manager

The remote evaluation manager is responsible for the interaction with the remote resource (cloudlet/cloud). The configuration parameters along with the input function are initially sent to the remote resource using a HTTP POST request for processing. Upon receiving the request, a socket connection is established between the phone and the remote resource. We use WebSocket as the communication protocol for two-way full-duplex communication. Whenever new sensor data is generated on the phone, it is sent to the remote resource for evaluation. The remote resource evaluates it and the result is sent back to the phone.

F. Remote evaluation engine

An application which uses Kea can offload its computation to any resource running a Java Virtual Machine. A remote resource can be a commercial cloud solution such as Amazon EC2 or simple edge solutions such as laptops, desktops or

TABLE I: Device configuration

Device	Chipset	CPU	RAM	Operating system	Location
Nexus 5	Snapdragon 800	Quad-core 2.3 GHz Krait 400	2 GB	Android 6.0.1	Amsterdam, The Netherlands
Nexus 6p	Snapdragon 810	Octa-core 4x1.55 GHz Cortex-A53 4x2.0 GHz Cortex-A57	3 GB	Android 6.0.1	Amsterdam, The Netherlands
SciCloud	Intel Corp. Standard PC (i440FX + PIIX, 1996)	Quad-core QEMU Virtual CPU 2GHz	8 GB	Ubuntu 16.04.2 LTS	Amsterdam, The Netherlands
DAS5-AMS (Head node)	Intel(R) Xeon(R)	32 core CPU E5-2630 v3@2.40GHz	125 GB	CentOS Linux 7 (Core)	Amsterdam, The Netherlands
DAS5-LID (Head node)	Intel(R) Xeon(R)	16 core CPU E5-2630 v3@2.40GHz	125 GB	CentOS Linux 7 (Core)	Leiden, The Netherlands
EC2-t2-FRA	Intel(R) Xeon(R)	Single-core CPU E5-2676 v3@2.40GHz	1 GB	Ubuntu 16.04.2 LTS	Frankfurt, Germany
EC2-t2-OR	Intel(R) Xeon(R)	Single-core CPU E5-2676 v3@2.40GHz	1 GB	Ubuntu 16.04.2 LTS	Oregon, USA

home servers. The URL for the remote resource is provided as a configuration parameter. The user runs a simple Java application that contains the remote evaluation engine, to enable computation offloading. The remote evaluation engine behaves similar to the local evaluation engine. The sensor data collected from the phone is processed over a window by applying the given input function and the result of the evaluation is sent to the phone.

IV. Evaluation

In this section we describe various evaluations performed using our system. We address the following questions:

- What is the overhead of the profiling and the decision making?
- What is the impact of the sensor frequencies on the decision-making process?
- What is the impact of the hardware used on the phone on the decision-making process?
- Can various types of operation have an effect on the decision-making process?
- What is the impact of the round trip time (RTT) to the remote resource on the decision-making process?
- What is the impact of the network type on the decision-making process?

Next, we briefly describe the evaluation setup used and various evaluations performed.

A. Evaluation setup

In the evaluation setup we use the devices shown in Table I. For experiments using WiFi, the phone was kept in airplane mode and with WiFi on. For experiments using 3G and 4G, we switched off WiFi and disabled the airplane mode. All the other applications were either stopped or disabled on the phone and the screen was turned off. For the remote case, we chose a Virtual Machine (VM) from a local cloud (SciCloud) offered by our university with an RTT value of 6 ms, running the remote evaluation engine and with the configuration mentioned in Table I.

To test our scenarios, we apply two different functions on the sensor data: MEAN and MEDIAN. We chose these two operations because they are commonly used in sensor-based applications and their computational complexities are different. MEAN runs in $\mathcal{O}(n)$ time complexity whereas MEDIAN runs in $\mathcal{O}(n \log n)$ time complexity where n is the window size. To obtain a realistic workload, we recompute the MEAN and MEDIAN for the entire window for each new sensor value, instead of using incremental updates. We also parallelized both functions (named MEAN_PARALLEL and MEDIAN_PARALLEL) to utilize more cores for higher values of n. We use a test sensor that generates floating-point numbers at various frequencies. In case of local evaluation, the elapsed time is the time taken to process the data locally. In case of remote evaluation, the elapsed time is the total time taken to send the newly generated sensor data, process it in the remote resource and receive the result from the remote resource. Each configuration test was run 5 times and the results were averaged. The maximum standard deviation was 14 ms for the elapsed time and 50 mW for the power consumption. Next, we describe various evaluations we performed.

TABLE II: Profiling overhead for 1 hour measurement using Nexus 5

Profiling	Energy (mWh)	Overhead (%)
One-time	283.32	3
Continuous	400	45
Incremental	293.05	6.5

B. Profiling overhead

Profiling applications, i.e., analyzing application behavior to determine the energy costs, also consumes energy and is considered overhead. Here, we use the Trepn [18] Android application profiler which determines the power consumption of a given Android application. For accurate energy profiling of the application, we follow the recommended profiling time of two minutes. The reported overhead energy cost of the Trepn profiler application is 125 mW [20] on a Nexus 6 smartphone which holds a Snapdragon 805 (APQ8084) up to 2.7 GHz. In this paper, we use two similar smartphones: a

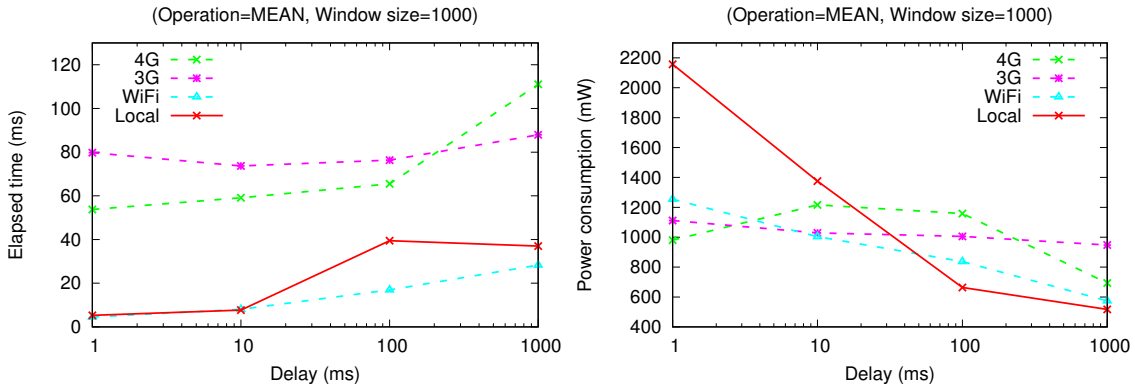

Fig. 2: Measurement of various sensor frequencies (or delays) for window size 1000

Nexus 5 and a Nexus 6p. The Nexus 5 smartphone has a Snapdragon 800 (8974-AA), a 32Bit quad-core processor up to 2.26 GHz while the Nexus 6p smartphone has Snapdragon 810 (MSM8994), a 64Bit octa-core processor up to 2.0 GHz. Here, we assume that the profiling costs are the same for the Nexus 5 and Nexus 6 CPUs as they share the same 28 nm High-Performance Mobile lithography process. However, the Nexus 6p has a 64Bit processor with two sets of cores created on a 20 nm lithography process, so the energy cost of profiling might not be the same as for the Nexus 5 and 6 models.

To get an indication of the profiling overhead, we estimate the overhead cost of profiling applications that run for 1 hour for the following 3 scenarios: one-time, continuous and incremental profiling. The first scenario profiles an application only for the first 4 minutes of the application running time, i.e. 1 hour including 4 minutes profiling. In the second scenario, the application profiling is done continuously throughout the application lifespan. In the last scenario, after 4 minutes of initial profiling, the application is profiled for the first 5 seconds of every minute incrementally. The overhead of profiling application for these different scenarios is presented in Table II. In this work, we primarily use one-time profiling as the overhead is relatively low (3%) for running the application for one hour.

C. Delay

We first study the impact of the sensor frequency (or delay). Figure 2 shows the elapsed time and the power consumption measurement for a window size of 1000 elements. Note that the figure uses a log scale in the x-axis and dashed lines for remote executions. We notice that for lower frequencies (higher delay) the elapsed time increases. This unexpected behavior is due to the automatic CPU frequency scaling done by Android's interactive CPU governor for saving battery consumption. The CPU runs at a lower frequency when there is no more processing. In the remote case, the network goes to idle state when there is no frequent data transfer. There is a latency involved in bringing the network from idle to active state. For local and WiFi we see a crossover point in both elapsed time and power consumption. For delays higher than 10 ms, in terms of computation time it is better to do the computation remotely. In terms of power consumption we notice that the crossover occurs between a delay of 10 ms and 100 ms. It is better to perform the evaluation locally with a delay of more than 100ms. For 3G and 4G, for all delay ranges it is still better to do the computation locally with respect to the elapsed time. However, in terms of power consumption, 3G and 4G are better for lower delay. For all these scenarios, as both elapsed time and power consumption results are conflicting, it becomes difficult to decide what is the optimal solution. Kea therefore allows the user to provide relative weights for time and energy.

D. Window size

Figure 3, 4 and 5 show the elapsed time and power consumption for various scenarios on multiple window sizes. For a given window size, we start running the test after the window is filled with sensor data. For example, for a window size of 10000 elements we start the test once the window is filled with 10000 elements both in the local and in the remote case. Based on the frequency of the sensor, a newly generated element will be added to the window by replacing the oldest element.

When the window size increases, both the elapsed time and the power consumption for local computation increase. Since the processor in the remote resource is more powerful than in the local case, we notice no change in the elapsed time for remote computation for a window size of up to 10000 data elements. In Figure 3 the crossover point of local and remote for elapsed time appears to be at a window size of 1000 data elements for Nexus 5. In terms of power consumption, the crossover happens between the window size of 100 and 1000 data elements for Nexus 5. The local computation is better for a lower window size.

E. Phone type

Figure 3 also shows the elapsed time and the power consumption on two smartphones (Nexus 5 and Nexus 6p) for a sensor data delay of 10 ms and for the MEAN operation. The elapsed time for local and remote is measured separately for

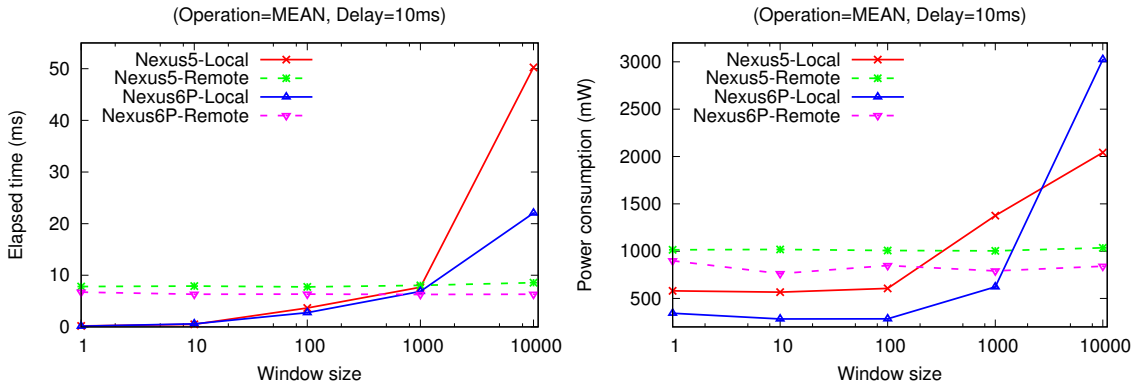

Fig. 3: Measurement of two devices (Nexus 5 and Nexus 6p) with different hardware capability

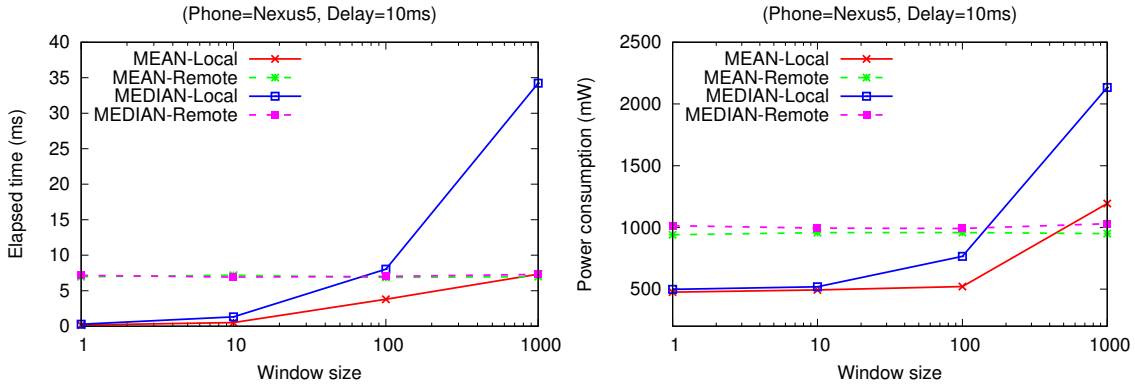

Fig. 4: Measurement of two different operations MEAN (O(n)) and MEDIAN (O(nlogn)) on a Nexus 5 device

both devices. For a window size of up to 1000 data elements, both devices have similar elapsed time for local. As the window size is small, the Android system runs the application at a lower CPU frequency for both devices, causing the elapsed time to be almost the same. For a larger window size of 10000 elements, as both devices are running at different high CPU frequency, we see a significant increase in the elapsed time for Nexus 5 compared to Nexus 6p. The reason is that Nexus 6p has a better chipset compared to Nexus 5. The throughput of the local computation for Nexus 6p is 45% compared to Nexus 5 which has only 20%. In terms of power consumption, for 10000 elements we see an increase for Nexus 6p because its throughput is higher compared to Nexus 5. It implies that more computations occur for Nexus 6p compared to Nexus 5 for a given period of time. Therefore, for applications with a large window size, it is better to offload computation for phones with low-end chipsets.

F. Function

In this section we compare the elapsed time and the power consumption for two functions: MEAN and MEDIAN, as shown in Figure 4. The MEAN operation calculates the moving average over a window of n data elements. MEDIAN uses sorting to identify the data element in the center position. The evaluation is done on a Nexus 5 device with a sensor data delay of 10 ms. The crossover points for MEAN and MEDIAN are significantly different. For the MEAN operation with $\mathcal{O}(n)$ time complexity, the crossover for the elapsed time occurs at a window size of 1000 elements whereas for MEDIAN with $\mathcal{O}(n \log n)$ time complexity, the crossover occurs at 100 elements. The crossover points for power consumption are between 100 and 1000 for both operations, closer to 100 elements for MEDIAN and closer to 1000 elements for MEAN operation. Hence, if we increase the window size, the possibility to do remote computation also increases for operations with higher time complexity.

We also implemented the parallel version using Java Thread pool for both MEAN and MEDIAN operations: MEAN_PARALLEL and MEDIAN_PARALLEL. Figure 5 shows the elapsed time and the power consumption for the sequential and parallel versions of the MEDIAN operation on Nexus 5. Since Nexus 5 is quad-core, we created a thread pool of 4 worker threads to do the computation. For a window size of up to 1000 data elements, the elapsed time of the sequential version is similar to the parallel version. There is a communication overhead for assigning tasks to the worker threads and getting back the results on the master thread. For 10000 elements, the parallel version performs better compared to the sequential version. So, the number of tasks assigned per thread is high and the time taken to complete the task

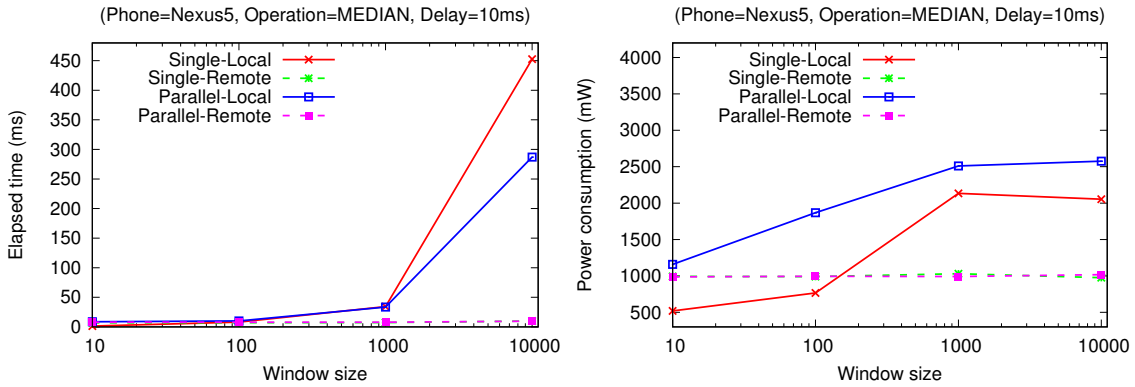

Fig. 5: Measurement of sequential and parallel algorithm for MEDIAN on a Nexus 5 device

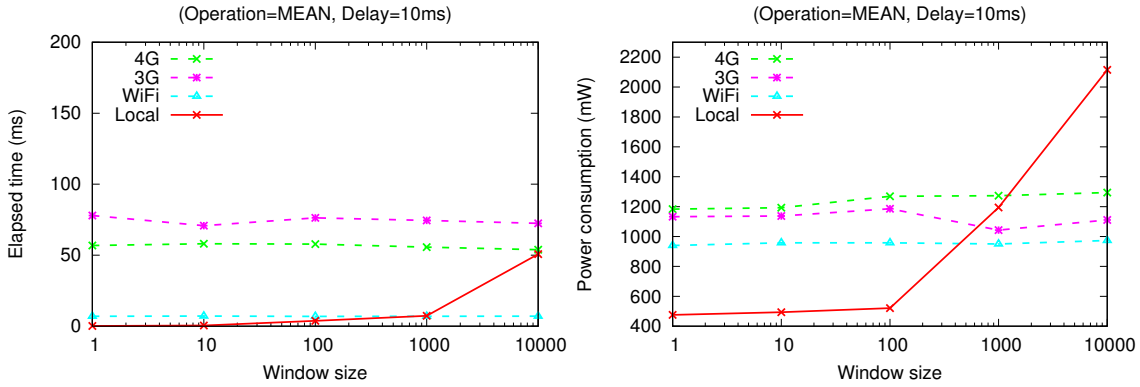

Fig. 6: Measurement of network technologies on SciCloud

exceeds the time taken to assign new tasks. The power consumption for the sequential version is lower than for the parallel version because the parallel version is using all CPU cores simultaneously. On the phone, if the power consumption is the preferred setting, the sequential version is better for all window sizes. If the elapsed time is the preferred setting, then the parallel version can be chosen for a higher window size. Of course it depends on whether the function can be parallelized or not.

G. Network type

Figure 6 shows the comparison of using 3G, 4G and WiFi for the MEAN operation with a delay of 10 ms on a Nexus 5 device. The SciCloud node is in the same network as the WiFi. 3G and 4G are multiple hops away from the SciCloud node. The elapsed time using WiFi is the lowest and for 3G it is the highest. We also notice a crossover between local and 4G when the window size reaches 10000 elements. In terms of the elapsed time, for up to 10000 elements it is better to do the computation locally than using 3G. In terms of power consumption, for remote case, WiFi consumes the lowest. The crossover point between local and remote is at 1000 elements. In the future, with 5G the chance for remote computation can even get better for a lower window size. If we choose local and 4G, we see a conflict between 1000 and 10000 window size.

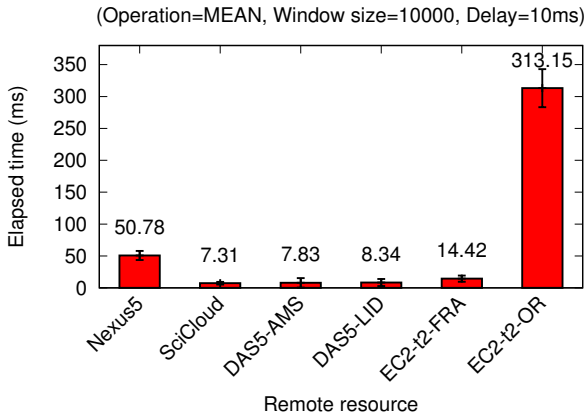

Fig. 7: Elapsed time for various remote resources

Local has lower elapsed time whereas 4G has lower power consumption. In such scenarios, it becomes difficult to decide which solution to choose. The choice of the network type has an impact in the decision-making process.

H. Network latency

Figure 7 shows the elapsed time for different machines located at various physical locations. The experiment uses

a Nexus 5 phone connected to our university network (at Amsterdam, The Netherlands) using WiFi. The SciCloud and DAS5-AMS [21] were also connected to the same network (which makes it an edge resource). The DAS5-LID is located at Leiden, The Netherlands. EC2-t2-FRA and EC-t2-OR are Amazon EC2 machines with t2.micro (lowest-cost general purpose instance type) instance are located at Frankfurt, Germany and Oregon, USA respectively. From Table I, we note that the device configurations for these machines are diverse. The experiment uses the operation MEAN on a window size of 10000 elements and a sensor data delay of 10 ms. For this configuration, the elapsed time between SciCloud (standard machine) and DAS5-AMS (cluster) is similar. The elapsed time for EC2-t2-FRA is 14.42 ms and for EC2-t2-OR is 313.15 ms. As both machines have the same configuration, it implies that for this scenario, the distance to the remote machine is more important than the configuration of the machine itself. We see a significance in the location of the remote device when making the offloading decision. The closer the remote resource is to the smartphone the better the chance for remote offloading.

I. Summary

We now summarize the main insights from the evaluation. Profiling has very little overhead (3% for 1 hour execution) in the way we use it in this paper (one-time profiling). The change in the sensor data frequency (delay) shows unexpected behavior due to the dynamic CPU frequency scaling and the change in the network state. Offloading to a remote resource is better for a higher window size. Apart from the frequency and the window size, the decision to offload also depends on the type of hardware, the operational complexity, the network latency and the network type.

V. Conclusion

We built a system called Kea that uses a profiling-based approach to automatize the decision making on whether to offload the computation to a remote resource. Kea is the first system that focuses on offloading the processing of streaming (sensor) data. We showed that the crossover point for offloading computation changes based on various parameters such as the application characteristics, the type of hardware used and the communication latency. We note that the one-time profiling overhead is 3% for a 1 hour execution.

As future work we plan to evaluate our system in dynamic scenarios where the network characteristics and the sensing frequency change over time. We also plan to use other profilers to minimize the overhead for such dynamic decisions. Finally, we plan to study the impact of GPU based context-aware applications on the offloading decision.

Acknowledgment

This work is funded by the municipality of Alkmaar, The Netherlands. It takes place in the context of Data Science Alkmaar led by Frans Feldberg.

References

[1] G. A. Lewis, "Software architecture strategies for cyber-foraging systems," Ph.D. dissertation, Vrije Universiteit Amsterdam, 2016.

[2] R. Kemp, "Programming frameworks for distributed smartphone computing," Ph.D. dissertation, Vrije Universiteit Amsterdam, 2014.

[3] "Sense-os," http://developer.sense-os.nl/, accessed: 2017-07-06.

[4] L. Yang, J. Cao, Y. Yuan, T. Li, A. Han, and A. Chan, "A framework for partitioning and execution of data stream applications in mobile cloud computing," *ACM SIGMETRICS Performance Evaluation Review*, vol. 40, no. 4, pp. 23–32, 2013.

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

[6] R. Bharath Das, N. V. Bozdog, and H. Bal, "Cowbird: A flexible cloud-based framework for combining smartphone sensors and iot," in *Proceedings of the 5th IEEE International Conference on Mobile Cloud Computing, Services, and Engineering*. IEEE, 2017.

[7] 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.

[8] M. A. Khan, H. Debnath, N. R. Paiker, N. Gehani, X. Ding, R. Curtmola, and C. Borcea, "Moitree: A middleware for cloud-assisted mobile distributed apps," in *Mobile Cloud Computing, Services, and Engineering (MobileCloud), 2016 4th IEEE International Conference on*. IEEE, 2016, pp. 21–30.

[9] Y. Kang, J. Hauswald, C. Gao, A. Rovinski, T. Mudge, J. Mars, and L. Tang, "Neurosurgeon: Collaborative intelligence between the cloud and mobile edge," in *Proceedings of the Twenty-Second International Conference on Architectural Support for Programming Languages and Operating Systems*. ACM, 2017, pp. 615–629.

[10] K. K. Rachuri, C. Mascolo, M. Musolesi, and P. J. Rentfrow, "Sociablesense: exploring the trade-offs of adaptive sampling and computation offloading for social sensing," in *Proceedings of the 17th annual international conference on Mobile computing and networking*. ACM, 2011, pp. 73–84.

[11] P. P. M. Silva, J. Rodrigues, J. Silva, R. Martins, L. Lopes, and F. Silva, "Using edge-clouds to reduce load on traditional wifi infrastructures and improve quality of experience," in *Proceedings of the 1st International Conference on Fog and Edge Computing*. IEEE, 2017.

[12] K. Alanezi, X. Zhou, L. Chen, and S. Mishra, "Panorama: A framework to support collaborative context monitoring on co-located mobile devices," in *International Conference on Mobile Computing, Applications, and Services*. Springer, 2015, pp. 143–160.

[13] P. Liu, D. Willis, and S. Banerjee, "Paradrop: Enabling lightweight multi-tenancy at the networks extreme edge," in *Edge Computing (SEC), IEEE/ACM Symposium on*. IEEE, 2016, pp. 1–13.

[14] W. Zhang, Y. Wen, J. Wu, and H. Li, "Toward a unified elastic computing platform for smartphones with cloud support," *IEEE Network*, vol. 27, no. 5, pp. 34–40, 2013.

[15] C. Shi, V. Lakafosis, M. H. Ammar, and E. W. Zegura, "Serendipity: enabling remote computing among intermittently connected mobile devices," in *Proceedings of the thirteenth ACM international symposium on Mobile Ad Hoc Networking and Computing*. ACM, 2012, pp. 145–154.

[16] M. Jang, H. Lee, K. Schwan, and K. Bhardwaj, "Soul: an edge-cloud system for mobile applications in a sensor-rich world," in *Edge Computing (SEC), IEEE/ACM Symposium on*. IEEE, 2016, pp. 155–167.

[17] R. Kemp, N. Palmer, T. Kielmann, and H. E. Bal, "Cuckoo: A computation offloading framework for smartphones." in *MobiCASE*. Springer, 2010, pp. 59–79.

[18] "Trepn power profiler," https://developer.qualcomm.com/software/trepn-power-profiler, accessed: 2017-07-06.

[19] E. Triantaphyllou, B. Shu, S. N. Sanchez, and T. Ray, "Multi-criteria decision making: an operations research approach," *Encyclopedia of electrical and electronics engineering*, vol. 15, no. 1998, pp. 175–186, 1998.

[20] "Trepn power profiler overhead for nexus 6," https://developer.qualcomm.com/forum/qdn-forums/software/trepn-power-profiler/34495, accessed: 2017-07-25.

[21] H. Bal, D. Epema, C. de Laat, R. van Nieuwpoort, J. Romein, F. Seinstra, C. Snoek, and H. Wijshoff, "A medium-scale distributed system for computer science research: Infrastructure for the long term," *Computer*, vol. 49, no. 5, pp. 54–63, 2016.