FADES: Fine-Grained Edge Offloading with Unikernels

Vittorio Cozzolino
Technical University of Munich
cozzolin@in.tum.de

Aaron Yi Ding
Technical University of Munich
ding@in.tum.de

Jörg Ott
Technical University of Munich
ott@in.tum.de

ABSTRACT

FADES is an edge offloading architecture that empowers us to run compact, single purpose tasks at the edge of the network to support a variety of IoT and cloud services. The design principle behind FADES is to efficiently exploit the resources of constrained edge devices through fine-grained computation offloading. FADES takes advantage of MirageOS unikernels to isolate and embed application logic in concise Xen-bootable images. We have implemented FADES and evaluated the system performance under various hardware and network conditions. Our results show that FADES can effectively strike a balance between running complex applications in the cloud and simple operations at the edge. As a solid step to enable fine-grained edge offloading, our experiments also reveal the limitation of existing IoT hardware and virtualization platforms, which shed light on future research to bring unikernel into IoT domain.

KEYWORDS

Edge Computing, Virtualization, IoT

1 INTRODUCTION

Edge computing and Internet of Things (IoT) have become closely coupled in recent developments. IoT was initially conceived as extending the Internet with a new class of devices and use cases (e.g., personal devices, constrained networks). Early architectures and frameworks introduced the notion of cloud-dependent IoT deployments, with the assumption that most/all IoT edge networks need to be connected to the cloud. However, there are some cases in which this tight coupling between cloud and IoT is not desirable. For example, when: a) data needs to be processed at the edge, b) delay sensible applications require real-time responses, or c) the amount of data is too large to upload to the cloud (in real-time) without congesting the backhaul. Based on this observation, we advocate a *divide and conquer* approach where IoT and edge devices actively participate in the completion of a task instead of being passively polled by cloud services.

In this context, a key domain that will benefit from the interplay of edge and IoT is Smart Cities: complex environment with manifold IoT devices deployed by different providers and serving many purposes. Moreover, each device would posses different computational and sensory capabilities with varying geographic locations adding to the system convolution. Edge devices will have the responsibility to handle adequately the IoT resources and execute tasks following the back-end instructions. Meanwhile, not everything can be offloaded. A classification based upon application complexity, priority, criticality, power consumption and required physical resources can help in assessing which task to offload. The combination of hardware capabilities and software requirements will ultimately guide the choice.

We define this approach as *edge offloading*. Edge [1] offloading revisits the conventional cloud-based computation offloading, where mobile devices resort to resourceful servers to handle heavy computation [6]. To cater for the demands of IoT services, our approach is reversed: we promote a paradigm where computation is dispatched by the servers to constrained devices deployed at the network edge, close to users and data generators.

To enable edge offloading, recent technological breakthrough in virtualization has provided us new opportunities. For instance, Docker completely revisited the concept of VM by introducing containers. Containers focus on virtualizing at the operating system level, whereas other hypervisor-based solutions focus on abstracting the hardware layer. In the process of creating smaller and more specialized VM, unikernels have emerged as a promising technology. In essence, unikernels are single-purpose appliances that are compile-time specialized into standalone kernels [7]. Unikernels contain exclusively the application code guaranteeing a reduced image size, improved security and greater manageability.

Our main contribution is FADES (*F*unction virtuliz*A*tion bas*ED* *S*ystem), a modular system architecture designed for IoT edge offloading. To achieve reliability, scalability and flexibility, FADES takes advantage of MirageOS unikernels to isolate and embed application logic fragments in small, Xen-bootable images. We decided to adopt the unikernel technology for our implementation because it fits exactly our requirement to run single-purpose tasks without offloading complete application logic. Security is a core benefit but not the main motif behind our choice. Besides system design, the lessons learned from our experiments across different IoT hardware and platforms also shed light on future research to integrate unikernel into the IoT domain.

The rest of the paper is structured as follows: motivation and background (§ 2), related work (§ 3), system architecture description (§ 4), system implementation (§ 5), system evaluation (§ 6), discussion (§ 7) and finally future work and conclusions (§ 8).

2 MOTIVATION AND BACKGROUND

Simplicity is key to the IoT. Regardless of the back-end services, edge devices have to execute simple operations on data locally available. Therefore, by splitting a complex application into manifold simple and single-purpose tasks we can ship them in the shape of lightweight containers (specifically, unikernels). Task fragmentation grants also the possibility to hide the complete application logic for security concern.

IoT diversity and cardinality are like double-edged blades. On one hand we face the problem of heterogeneity and lack of standards. On the other hand, the massive scale of envisioned devices is a powerful source to harness. The core motive behind our system is

[1] Currently, there is no accepted definition of the difference between Edge and Fog. In this paper, we will use the term Edge to refer to groups of constrained devices deployed at the edge of network and able to process local data.

exactly to leverage this power in the right way. The combination of locally available resources, computational power and capabilities are key elements to properly offload tasks and, therefore, exploit IoT resources. Moreover, heterogeneity is less troublesome when we have intermediate nodes supervising and managing clusters of IoT devices instead of entrusting all this knowledge and responsibility to the cloud. This multi-level (cloud, edge and IoT) information pipeline is also motivated by the necessity of reducing the uplink access parallelism. By offloading computation we progressively aggregate data along the pipeline reducing massively the data to be uploaded and easing the burden on the cloud. Latency is also affected when we move computation closer to the data to be manipulated. However, not everything can be offloaded. Parameters as application complexity, priority, criticality, power consumption and required physical resources have to be taken into account.

Last but not least, security and privacy in IoT are increasingly important [12]. Small IoT devices are not powerful enough to guarantee the required degree of security. Therefore, we advocate the presence of intermediate control units across the communication pipeline between IoT and Cloud in a way to introduce additional resiliency and control. Privacy enabling modules can be deployed directly next to the source of the data, combined with security controls. Still, these features wouldn't be possible without a hardened execution environment. Our design principles offer functionality isolation and reduced attack surface. Moreover, deployed tasks run inside a virtualization platform (hypervisor) adding an additional layer of isolation and protection (subverting the system requires to find vulnerabilities in the hypervisor rather than the OS, which is more difficult).

3 RELATED WORK

Our work follows the trend of exploiting computational resources outside cloud deployment combined with use of unikernels [2, 7, 14]. Computation and data offloading has been explored in different flavors in [4, 5, 13] for aspects as energy efficiency and offloading decision policies. Instead, [15] is a pioneer in the field of computation offloading supported by VMs migration. Their approach is based on leveraging infrastructure resources based on mobile nodes proximity with the support of cloudlets: a trusted, resource-rich computer or cluster of computers. To this end, they adopt dynamic VM synthesis: a process of merging a static part of a VM with a dynamic component provided by a mobile device.

Unikernels have been explored in multiple research fields. Some research directions that make use of unikernels to improve existing services or propose new system architectures are [1], [10], [11], and [17].

More recently, Madhavapeddy et al. [8] proposed on-demand specialized VM instantiation. We leverage the idea of the *embedded cloud* presented in this paper and bring it further into the IoT domain. Compared with Jitsu, our work is geared towards IoT services and focuses on the system architecture. Additionally, our evaluation revealed the limits of unikernels by studying the performance with bigger payloads and in different network settings.

Airbox [3] presents a software platform based on onloading and backend-driven cyberforaging. It shares the general direction presented in our paper in terms of offloading the Edge Functions (EF). Compared with their solution, FADES achieves fine-grained offloading by using unikernels instead of Docker technology.

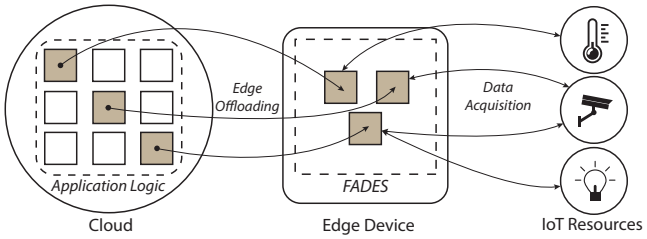

Figure 1: Edge Offloading

Databox [9] proposes a hybrid physical and cloud-hosted system for personal data management. The prototype of Databox is based on Docker. The authors have indicated MirageOS as a possible candidate to implement future extensions. To this end, FADES is a unikernel-based system dedicated for IoT edge offloading. Inspired by one of the use-cases of Databox, we further share our insights of processing sensors data at the edge through unikernels.

4 SYSTEM DESIGN

Data locality is the key property that drives our system design principles. We see into this physical distance between cloud and IoT a gap to be exploited. Therefore, our design introduces an intermediate unit to enrich and augment the interaction between IoT resources and External Services (ES), as depicted in Figure 2.

The IoT resources offer physical capabilities to interact with the environment and carry out basic tasks. The ES are back-end applications interacting with our system by offloading parts of their operation logic. In our design, we consider ES as a repository of deployment-ready tasks designed for different scenarios and purposes.

Recalling the Smart City example, pollution control is achieved by querying pollution sensors in the IoT, aggregating the information at the edge, and sending the final result to the cloud. In this regard, FADES resembles a middlebox and oversees groups of IoT devices based on spatial proximity, as Figure 1 shows. Therefore, each FADES unit supervises a subset of IoT devices (e.g. pollution sensors) scattered across the Smart City. Moreover, it has the responsibility to map and monitor the available IoT resources and retrieve data from them following the offloaded task requirements.

Based on the definition of edge offloading, our system is designed to support only pull workflows (from the cloud to the IoT). Push workflows, where edge devices offload tasks to the cloud, are not yet supported mostly because covered already in mobile offloading research.

The main components of FADES include Orchestrator (ORC), Data Resource Broker (DRB) and Data Manipulation Functions (DMF). As shown in Figure 2, FADES is an event-based system that responds to external commands referred to as *Metadata Task Wrapper* (MTW), which are issued on-demand by services and applications. The MTW can be divided into 3 types:

MTW Credentials: This type contains passport-like information (e.g., task ID, associated user or service, priority). It's mainly used to keep track of the received MTW and schedule its execution.

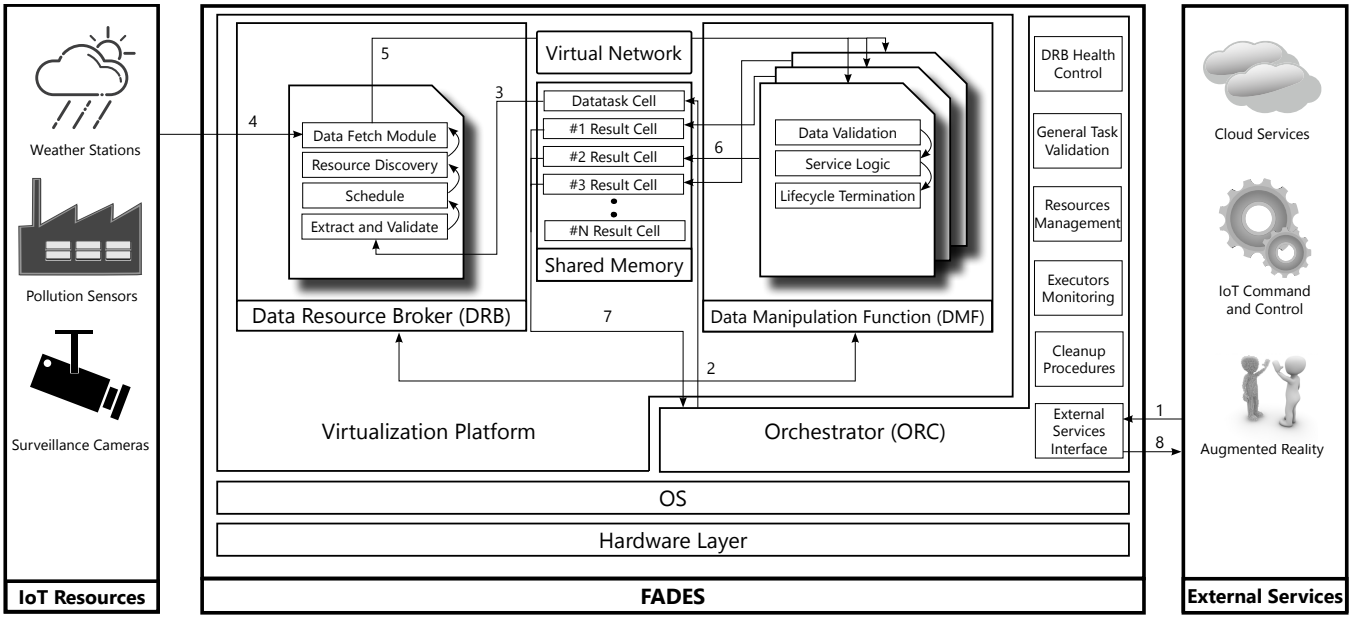

Figure 2: System Design

DRB Metadata: This type contains a list of *data retrieval operations* (DTO), which conveys details about what data to retrieve. DRB Metadata also specifies the source and destination of the data.

DMF Metadata: This type contains application specific details about the DMF. For instance, the MTW issuer can specify additional information about extra runtime configuration parameters or minimal required hardware resources.

4.1 FADES

The core of FADES consists of three components:

Orchestrator (ORC). The Orchestrator is the interface between the system and the outside world. Being a "supervisor" that monitors and controls the system, ORC frequently checks that both DRB and DMF are running correctly. ORC also takes care of dispatching the required information to the DRB the moment a new MTW arrives. It ensures the overall system integrity by monitoring the DRB, following the life-cycle of each DMF, validating uploaded tasks and decommissioning terminated DMFs.

Data Resource Broker (DRB). The DRB module localizes and extracts resources from a groups of IoT devices. Hence, it possesses the required knowledge regarding which IoT devices to query. The DRB is completely computation agnostic, whose execution cycle is event-based and driven by DRB Metadata contained into the MTW and received by the orchestrator.

Data Manipulation Functions (DMF). A DMF embeds exclusively the relative service logic and stays in a dormant state until it receives the correct data from the DRB. DMFs can be persistent or ephemeral, depending on the embedded application logic. Long running or recurrent tasks might exhibit a persistent behavior while single-execution tasks have to be deallocated after completion.

Recalling the pollution control example, we can map each FADES component onto the following roles: 1) the ORC receives from cloud the tasks to be executed and additional metadata, 2) the DRB will locate the correct pollution sensors (or weather stations) to be queried and retrieve the data, 3) the DMF will manipulate the pollution data received from the DRB, produce the final results and send them to the ORC.

5 IMPLEMENTATION

FADES is a virtualized, unikernel-based system hosted by the Xen hypervisor (except for ORC). Our tool of choice is the MirageOS library operating system, which is specifically designed to build modular systems and runs natively on Xen. MirageOS offers static type-safety combined with single-address space layout. Moreover, being compile-time defined and sealed, any code not presented inside a MirageOS unikernel during compiling time will never be executed, and hence preventing any sort of code injection attack [7].

In FADES, both the DRB and DMFs are MirageOS unikernels running on Xen as Para-Virtualized Machines (PVM). The former exclusively retrieves information and the latter process the received data. We enforce component isolation and independent development by confining functionality overlapping.

The DRB is implemented as a daemon unikernel. For internal communications, it uses two modules offered by Xen: the virtual network and the XenStore. The virtual network is used to internally transfer data to the DMFs. On the other hand, XenStore is exclusively used to exchange synchronization messages with the ORC. The DRB validates and schedules each MTW received from the ORC. In our implementation, we implemented different scheduling policies (e.g. sorting by task priority, execution deadline, task ID) but in the current stage we used a simple FIFO. Currently, the DRB is able to process in parallel multiple *data retrieval operations* but is only able to prosecute a single MTW at a time.

The DMFs used in our implementation execute simple aggregation operations over streams of sensors data (Section 6 will cover more details about this choice). The result of the computation is sent to the ORC through Xenstore. DMFs are hooked on the same virtual network of the DRB and do not have external network access. One limitation of using the virtual network is that we need to manage carefully the IP addresses to avoid collisions. In our current implementation, we didn't implement any automated deployment functionality covering this matter. The system currently supports parallel execution of multiple DMFs even thought it has not been fully tested on that regard.

The ORC is developed in Python and it's the only non-virtualized module in FADES. We choose Python because it has the right combination of performance and features that make prototyping fast and flexible. Moreover, the ORC is the only module that handles reads and writes towards the persistent storage. Our design choice focused on establishing a loose coupling between the host system and the unikernels managed by FADES, with the latter being exclusively dependent on *virtual* resources: CPU, RAM and network. In order to be independent from the hosting edge device, the DRB and DMF should be as flexible as possible and ephemeral.

6 EVALUATION

The goal of our evaluation is answering the following questions:

Q1. How different architectures (x86, ARM) affect the performance of MirageOS? What are the Unikernel PVM memory sizing requirements in relation to the amount of data to be manipulated?

Q2. How does our system perform under different workloads and what is the overhead introduced by using multiple modules?

Q3. How much edge deployed services can benefit from data locality? What is the trade-off?

For our tests, we selected three different devices: a Cubietruck (Allwinner A20 ARM Cortex-A7 dual-core @ 1GHz, 1GB RAM, 100Mb Ethernet), an Intel NUC (Intel(R) Core(TM) i5-6260U CPU @ 1.80GHz, 16GB RAM, 1000Mb Ethernet) and a Dell PowerEdge R520 (Intel(R) Xeon(R) CPU E5-2640 v3 @ 2.60GHz, 140GB RAM, 1000Mb Ethernet) running Xen 4.4, InfluxDB v1.0 and an Ubuntu 14.04 dom0. Additionally, we used MirageOS 2.9.1 (the latest stable version at the time of writing), OCaml 4.02.3, OPAM 1.2.2 and Python 2.7.6.

We gleaned the data for the tests from our Intel Edison IoT testbed. The testbed is composed by 5 Intel Edison boards deployed in different office rooms on the campus. Each board continuously collects environmental data through a set of sensors (humidity, temperature, light intensity, audio, proximity). At the time of writing, the testbed collected roughly 300 millions of data points equally divided among the 5 sensor classes. Each data point is a row in our database containing: timestamp, sensor value, sensor type, measurement unit and location. The deployment has been running constantly since June 2016.

The testbed is to carry out user-context modeling and correlate sensors readings with human behaviors/actions in each room. Some of the aggregation operations carried out on the testbed have inspired the algorithm embedded in our unikernels. Therefore, the DMFs used for our test execute simple manipulation and aggregation functions (e.g., calculate minimum, maximum, average) over sensors data streams.

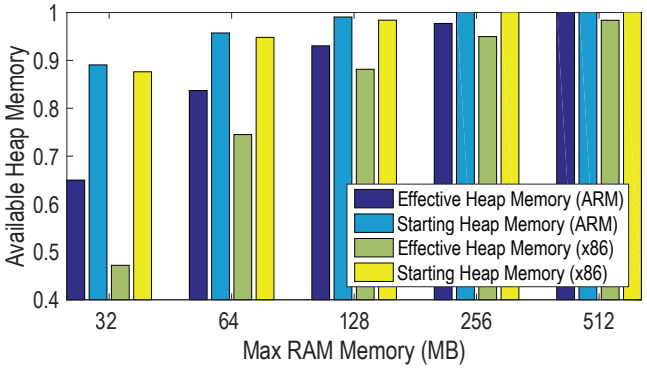

Figure 3: Memory Analysis

Memory Analysis. When offloading to resource constrained devices, it's important not to abuse the already limited available hardware resources. In this section we analyze how the available RAM memory affects DMFs performance. This study serves as a guideline to the process of correctly dimensioning a MirageOS PVM. Therefore, we aim at avoiding the over/under dimensioning issues that could lead to, respectively, waste of resources and out of memory exceptions.

Figure 3 shows the ratio between available heap memory and pre-allocated RAM with different architectures (x86 and ARM). On both architectures the effective available memory is lesser than the amount allocated at the beginning but the behavior varies between x86 and ARM. In the first case, the gap is much more prominent, and this directly affects the amount of data processable by the Unikernel especially in the case of PVM with low allocated RAM memory.

Two main factors influence the available memory for an unikernel PVM: underlying architecture and imported libraries. We cannot economize on the latter, given that it depend on the specific application logic. On the other hand, different system architectures constantly generate different unikernel images. While on ARM the building output is a Linux kernel ARM boot executable zImage (.xen) plus a ELF 32-bit LSB executable (.elf), on x86 we have a single .elf file. The ultimate difference in size of the generated Xen image on the two architectures affects directly the available memory at runtime.

Figure 4 shows the correlation between pre-allocated RAM and maximum amount of processable data. These data has been obtained by studying in details the performance of a single DMF completely isolated from the system. We monitored its limits by feeding an increasing amount of data to process. From our tests, we noticed that the device resources doesn't influence at all the maximum amount of processable data. Hence, the graph shows a generic comparison between ARM and x86.

The gap between these two architecture is much more prominent in the case of PVM equipped with low RAM memory, whereas it tends to disappear with 512MB. In some cases, we are only able to process half of the data on x86 architecture. The red error bar is the standard deviation.

System Analysis: Overhead and Offloading. The purpose of this section is to show a performance comparison when executing task at edge instead of in the cloud.

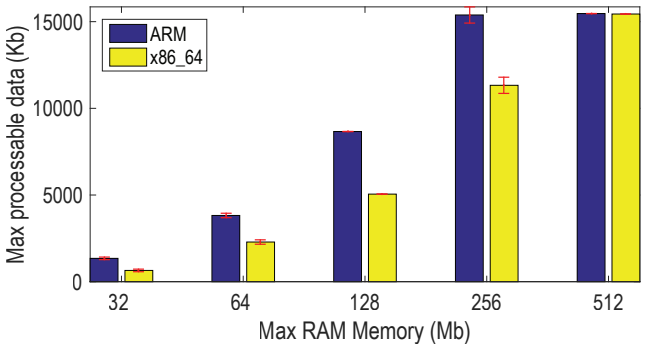

Figure 4: Data processing limits (memory)

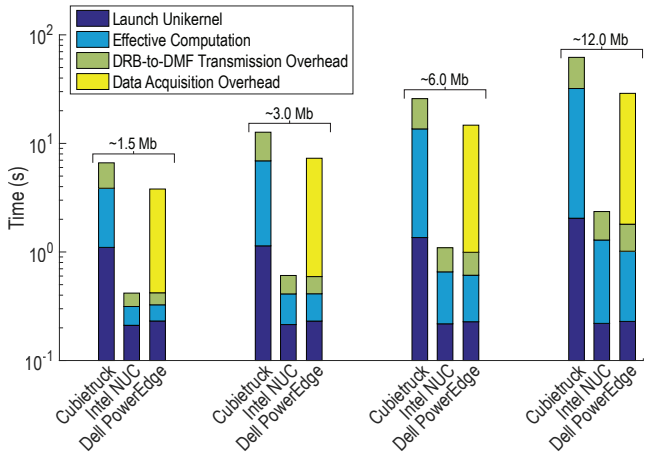

Figure 5: System Performance

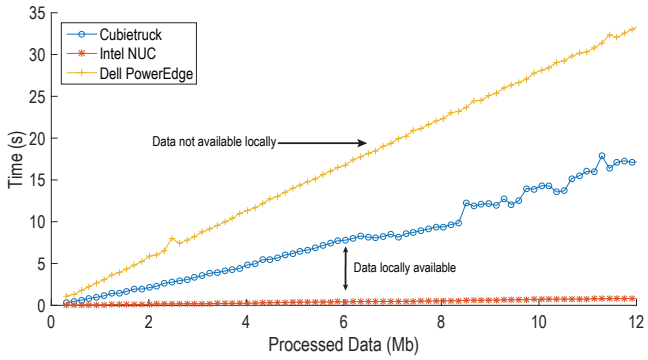

Figure 6: Effect of Data Locality on Computation Time

Figure 5 shows an answer to **Q2** by highlighting the system performance with different data payload sizes and providing a detailed breakdown of the execution time of a task in FADES. Four main factors affect the overall execution time: 1) required time to boot-up the DMF unikernels, 2) time required to transfer the data between the DRB and the DMF, 3) computation time of the DMF and 4) the time required to retrieve the data to manipulate.

In particular, factor 1) is an aggregated value representing the overhead introduced by the ORC module. For now, the ORC is a thin layer that handles requests from the external services and doesn't process data or applies any changes to the tasks executed by the DMFs. It checks that everything works correctly but doesn't take part in any computation phases. Factor 4) only affects the scenario where the Dell PowerEdge server has to retrieve data from a remote network. The bars are grouped by amount of data to be processed. For example, the first group shows the performance for each devices given a payload of 1.5Mb.

The results show that the presence of a sufficiently powerful device at the edge of the network combined with data locality makes edge offloading the best decision. Particularly, the Intel NUC outperforms the Dell PowerEdge while the Cubietruck is highly hindered by the intra-unikernel transmission time overhead.

Figure 6 shows an answer to **Q3** by presenting in details how data locality strongly affects performance regardless of the hardware capabilities. Therefore, we focused on observing how data locality affects computation time. The Cubietruck and the Intel NUC had a local copy of the sensors data while the Dell PowerEdge (the cloud) was forced to retrieve the data from a remote location. To this avail, we deployed our Intel NUC and the Cubietruck into another building and used them as a data source. In the graph we can clearly see that having data locally can improve performance even on resource constrained devices.

The remote data location is accessed by the Dell PowerEdge through a standard broadband connection with the following specifications: 45.38 Mbps downlink and 0.574 Mbps uplink (effective).

Connection speed is a critical factor in our evaluation; we are aware that faster/slower connections will lead to different result. Still, we want to point out that even by reducing to zero the data acquisition overhead factor, the Intel NUC performance are surely vying with the Dell PowerEdge.

The cost of uploading the unikernel has been removed from our evaluation. This factor it's directly bound to the nature of the application logic. In a real scenario, it is very well possible that a DMF could actually be reused multiple times (and by multiple users) before it's updated. In other words, the frequency at which the data changes is greater than the one of the deployed task. On the other hand, the task might need to be updated constantly rendering it obsolete at every new compute cycle. The chances for the latter to happen are dim, yet possible.

Last but not least, we are aware of the asymmetric nature of common broadband subscriptions with a ratio of 1:10 between upload/download speed. Hence, the cost of uploading data from the Cloud to the edge will generally be marginal compared to the opposite.

7 DISCUSSION

We have learned several practical lessons from system development and experiments, which can be summarized as *hardware limitations*, *platform support*, and *security concern*. First of all, it's demanding to find suitable embedded boards that can support Xen and MirageOS, regardless of x86 and ARM architectures. In most cases, the main culprits are hardware incompatibilities and poor documentation (if

there is any). Our experiments on Cubietruck have benefited from the MirageOS discussion group, where we found detailed setup guidelines. On the other hand, the deployment on mini-PC (e.g. Intel NUC) is much easier, since those devices essentially resemble the standard PC.

Secondly, our system performance is affected by the limitation of MirageOS. In particular, we struggled with the network API when transferring data between two unikernels. The main issue comes from a bug in the TCP/IP MirageOS stack that doesn't handle properly writing packets larger than the MTU. In consequence, we had to introduce an extra chunking function at the application layer to split, and later reconstruct the data. This negative effect is reflected in Figure 5 where the overhead of this operation prolongs the completion time, especially on constrained devices like the Cubietruck.

Finally, it takes extra steps to enhance system security. FADES enables an execution paradigm where single-purpose functionalities are offloaded from the service provider (e.g., cloud) to edge devices. We hence need to guarantee the authenticity and validity of the offloaded tasks. Without a signing and validation infrastructure to discriminate legit from tampered unikernels, we might risk executing malicious code and infringe the security requirements. Furthermore, if a FADES module is compromised or hijacked, regardless of the offloaded code, the attacker is able to manipulate the results of trustworthy DMFs. Therefore, a strict control over the system execution pipeline and constant monitoring of FADES is mandatory, as suggested in [16].

8 CONCLUSION AND FUTURE WORK

FADES is a modular offloading architecture that leverages lightweight virtualization to enable fine-grained edge offloading for IoT. The underlying idea is to bridge the gap between complex applications running in the Cloud and simple operations running at the edge. It's in this gap that we spot the opportunity to utilize unikernels as an ideal vessel to ship single-purpose tasks for achieving modularity, flexibility and multi-tenancy. As a first step to explore the potential of lightweight virtualization in IoT and edge computing, our experimental insights shed light on the hardware deployment and performance optimization for both system engineers and researchers.

Our future work will focus on three aspects. Firstly, we plan to investigate the system scalability by running multiple function instances in parallel. Secondly, we will evaluate FADES against real-world applications. Lastly, we will harden the system security design to meet the security requirements of IoT. In addition, we will also compare our system with other existing solutions.

ACKNOWLEDGEMENT

This research was partially supported by Google Internet of Things (IoT) Technology Research Award Pilot. Moreover, it is part of the TUM Living Lab Connected Mobility (TUM LLCM) project and has been funded by the Bavarian Ministry of Economic Affairs and Media, Energy and Technology (StMWi) through the Center Digitization Bavaria, an initiative of the Bavarian State Government.

We thank our shepherd, Ryan Huang for the valuable feedback and support. Moreover, we thank also our colleagues from UCI who provided insight and expertise that greatly assisted the research.

REFERENCES

[1] Bob Duncan, Andreas Happe, and Alfred Bratterud. 2016. Enterprise IoT Security and Scalability: How Unikernels Can Improve the Status Quo. In *Proceedings of ACM UCC '16*.
[2] Bratterud et al. 2015. IncludeOS: A minimal, resource efficient unikernel for cloud services. In *Proceedings of IEEE CloudCom '15*. IEEE.
[3] Bhardwaj et al. 2016. Fast, scalable and secure onloading of edge functions using AirBox. In *IEEE/ACM Symposium on Edge Computing (SEC)*. IEEE.
[4] Ding et al. 2013. Enabling energy-aware collaborative mobile data offloading for smartphones. In *Proceedings of IEEE SECON '13*.
[5] Kosta et al. 2012. Thinkair: Dynamic resource allocation and parallel execution in the cloud for mobile code offloading. In *Proceedings IEEE INFOCOM '12*.
[6] Kumar et al. 2013. A survey of computation offloading for mobile systems. *Mobile Networks and Applications* 18, 1 (2013), 129–140.
[7] Madhavapeddy et al. 2013. Unikernels: Library operating systems for the cloud. In *ACM SIGPLAN Notices*, Vol. 48. ACM.
[8] Madhavapeddy et al. 2015. Jitsu: Just-In-Time Summoning of Unikernels.. In *Proceedings of NSDI '15*.
[9] Mortier et al. 2016. Personal Data Management with the Databox: What's Inside the Box?. In *Proceedings of the 2016 ACM Workshop on Cloud-Assisted Networking*.
[10] Sathiaseelan et al. 2015. SCANDEX: Service Centric Networking for Challenged Decentralised Networks. In *Proceedings of ACM DIYNetworking '15*.
[11] Siracusano et al. 2016. On the Fly TCP Acceleration with Miniproxy. In *Proceedings of ACM HotMiddlebox '16*.
[12] Jorge Granjal, Edmundo Monteiro, and Jorge Sá Silva. 2015. Security for the Internet of Things: a survey of existing protocols and open research issues. *IEEE Communications Surveys & Tutorials* 17, 3 (2015), 1294–1312.
[13] Esa Hyytiä, Thrasyvoulos Spyropoulos, and Jörg Ott. 2015. Offload (only) the right jobs: Robust offloading using the markov decision processes. In *Proceedings of IEEE WoWMoM '15*.
[14] Avi Kivity, Dor Laor Glauber Costa, and Pekka Enberg. 2014. OS v—Optimizing the Operating System for Virtual Machines. In *Proceedings of USENIX ATC'14*.
[15] Mahadev Satyanarayanan, Paramvir Bahl, Ramón Caceres, and Nigel Davies. 2009. The case for vm-based cloudlets in mobile computing. *IEEE Pervasive Computing* 8, 4 (2009).
[16] Rolf H Weber. 2010. Internet of Things–New security and privacy challenges. *Computer Law & Security Review* 26, 1 (2010), 23–30.
[17] Dan Williams and Ricardo Koller. 2016. Unikernel monitors: extending minimalism outside of the box. In *8th USENIX Workshop on Hot Topics in Cloud Computing (HotCloud 16)*.