Demonstrating the Practicality of Unikernels to Build a Serverless Platform at the Edge

Chetankumar Mistry*, Bogdan Stelea†, Vijay Kumar‡ and Thomas Pasquier§
Department of Computer Science, University of Bristol
Email: *cm16161@bristol.ac.uk, †bs17580@bristol.ac.uk, ‡vijay.kumar@bristol.ac.uk, §thomas.pasquier@bristol.ac.uk

***Abstract*—The rise of IoT has led to large volumes of personal data being produced at the network's edge. Most IoT applications process data in the cloud raising concerns over privacy and security. As many IoT applications are event-based and are implemented on cloud-based, serverless platforms, we've seen a number of proposals to deploy serverless solutions at the edge to address concerns over data transfer. However, conventional serverless platforms use container technology to run user-defined functions. Containers introduce their own issues regarding security – due to a large trusted computing base –, and performance issues including long initialisation times. Additionally, `OpenWhisk` a popular and widely used container-based serverless platform available for edge devices perform relatively poorly as we demonstrate in our evaluation.**

In this paper, we propose to investigate unikernel as a solution to build serverless platform at the edge, addressing in particular performance and security concerns. We present `UniFaaS`, a prototype edge-serverless platform which leverages unikernels – tiny library single-address-space operating systems that only contain the parts of the OS needed to run a given application – to execute functions. The result is a serverless platform with extremely low memory and CPU footprints, and excellent performance. `UniFaaS` has been designed to be deployed on low-powered single-board computer devices, such as Raspberry Pi or Arduino, without compromising on performance.

I. Introduction

As edge devices often generate a large amount of personal data; users expect secure and fast processing of the data, but typically, distant cloud servers process the data produced at the edge. Employing computation at the edge devices can significantly minimise the amount of network I/O incurred by sending data to remote cloud servers. Further, a centralised solution may represent unnecessary privacy and security risks. Edge computing [1] has emerged as a solution to bring computation as close as possible to the source of data.

Serverless platforms [2] are an ideal technology to develop the event-based solutions, required by a significant number IoT applications. They let developers focus on the core functions needed to build a service, abstracting away scaling, provisioning and infrastructure management concerns. Developers can combine stateless functions triggered by events (e.g., incoming IoT sensor readings) into complex workflows to realise applications. This paradigm is often referred to as Function as a Service (FaaS). Popular offerings include AWS Lambda, Google Cloud Functions, Microsoft Azure Functions, IBM/Apache's OpenWhisk and Oracle Cloud Fn.

An ideal solution would be to deploy serverless platforms at the edge [3] (e.g., within Internet Gateway) to handle the stream of data coming from multiple IoT devices. Many attempts [3–6] in this space consist of redesigning container-based solutions used in the cloud to bring them to edge devices. However, this is known to cause significant performance problems on resource-constrained devices [7].

In this paper, we propose a more radical redesign, leveraging instead recent unikernel technologies [8–10]. Unikernels are tiny single address space operating systems (OSs) (<10MB in size) executing on top of the hypervisor. Unikernels have very low memory and CPU footprints and extremely fast boot times. As far as pure performance is concerned, unikernels outperform containers [11].

Unikernels also provide significant security advantages. They are immutable by nature, run directly on top of the hypervisor, and only take the parts of the OS that they need in order to operate. This creates a significantly smaller trusted computing base (TCB) when compared to container-based solutions running on top of a fully fledge general-purpose OS. Additionally, unikernels leverage hypervisor-level isolation, while containers notoriously suffer from poor isolation support [12].

Contributions:

- we design a unikernel-based Serverless Platform for edge devices;
- we demonstrate its relative performance gain compared to `OpenWhisk`, a widely popular open-source, container-based solution;
- we demonstrate the practicality of the approach by building an environment monitoring system (we build on top of the H2020 Smart Citizen Kit [13, 14]);
- finally, we propose an open-source implementation of the work presented in this paper.

The remainder of this paper is organised as follows: § II provides a technical background. § III describes the implementation of `UniFaaS`. § IV provides an evaluation by comparing its throughput, memory and CPU usage when compared with an alternative. § V demonstrates `UniFaaS` by applying it to a real use case. § VII performs a comparison of unikernel and container technology and discusses related work in the fields of serverless and edge computing.

II. Background

In this section, we provide technical background on serverless platforms and the core technologies used to build our solution: Unikernels and the Solo5 Hypervisor.

2330-2186/20/$31.00 ©2020 IEEE
DOI 10.1109/CloudCom49646.2020.00001

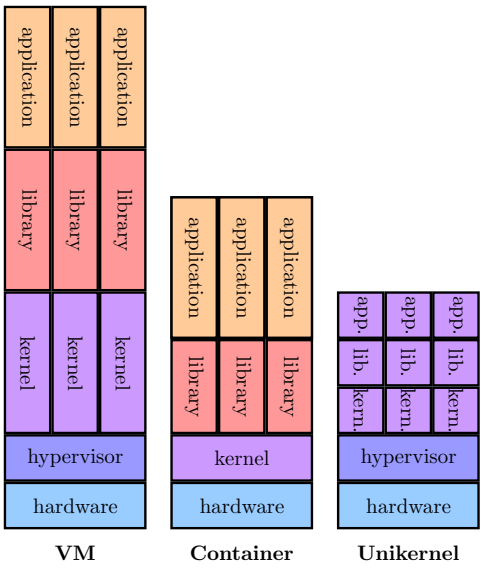


Fig. 1. Shared resources for VMs, containers and unikernels [11]

A. Serverless Platforms

Serverless platforms allow the user to run code without provisioning or managing servers with functionalities such as scaling, load balancing, fault tolerance and availability managed by the platform. Functions are triggered based on conditions and can be chained in complex workflows to build applications. Those functions are stateless and designed to be short-lived. The billing model measures the number of times a function is invoked and charges the customer accordingly. Serverless platforms rely on a sand-boxed execution engine to execute functions [2, 15]. Container technologies have been a popular means to build serverless platforms which pose performance concerns on resource-constrained edge devices [7].

B. Unikernels

A unikernel is a single-purpose and single address space OS linking only the components from an OS stack it needs to run a high-level application. This results in less memory wastage and a smaller TCB, which improves security. Fig. 1 compares the shared resources between VMs, containers and unikernels. Unikernels are programs compiled into immutable kernels, capable of running directly on a hypervisor. Unikernels are viable because FaaS workloads are typically small, single-use functions, whereas OSs are - by design - general-purpose systems.

Unikernels are of two categories: language-based and POSIX-based. Language-based unikernels (e.g. MirageOS and IncludeOS) typically use a bespoke library OS written entirely in the unikernel language of choice, such as OCaml [9], C++ [8] or Haskell [16]. POSIX-based unikernels such as OSv [17], HermiTux [18] and Lupine Linux [19] focus on compatibility with conventional Linux applications. The main difference between the two types is how specialised the unikernel is. Language-based unikernels are highly specialised, fast and secure, but lack compatibility with conventional systems. Whereas POSIX-based unikernels offer a versatile and flexible application, but they have a larger memory footprint, are slower to boot and are less secure, due to their relatively larger legacy TCB. Despite some of the inherent advantages of unikernels for FaaS workloads, current serverless platforms have not been designed to run unikernels [20].

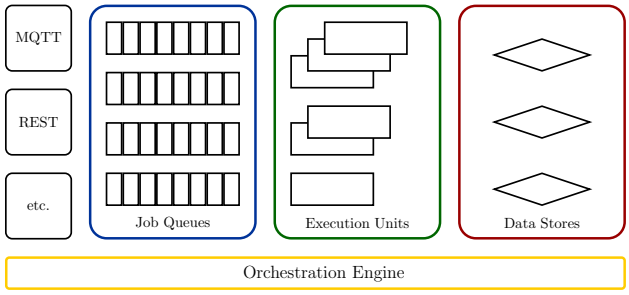


Fig. 2. High-level `UniFaaS` architecture.

In the proof of concept system presented in this paper we leverage the MirageOS [9] unikernel. MirageOS adopts a defence-in-depth approach. The compile-time specialisation, use of type-safe languages, hypervisor security measures and toolchain extensions provide a high degree of security.

C. Solo5 - Hypervisor

Solo5 [21] is a sandboxed, re-targetable execution for unikernels, enabling extremely fast booting with the ability to debug them. Solo5 provides a minimalist, legacy-free interface with bindings to a) microkernels (Genode), separation kernels (Muen); b) virtio-based hypervisors; and c) monolithic kernels (Linux, FreeBSD, OpenBSD). On monolithic kernels, a tender[1] is used to sandbox the unikernel strongly. Hardware virtualised tender (`hvt`) uses hardware virtualisation as an isolation layer. In contrast, Sandboxed process tender (`spt`) uses process isolation with seccomp-BPF [22] as an isolation layer. Solo5 has small implementation size (around 3 kLOC) and fast startup time (Solo5 hvt/spt $<$ 50 ms; QEMU Linux VM around 1000 ms; Cloud-managed VMs several seconds). In this paper, we leverage Solo5 `hvt` to run unikernels.

III. Implementation

The core components required to build a serverless platform are [15, 23]:

- Execution Units;
- Orchestration Engine;
- Job Queues;
- Datastore.

Fig. 2 shows the architecture of `UniFaaS`. We discuss these components in the rest of this section.

[1]Tender is the component responsible for "tending to" the guest at load/run time. In the case of hvt, the tender is loosely equivalent to QEMU. In the case of spt, `solo5-spt` tender loads the guest into memory; install a seccomp sandbox and passes control to the guest.

A. Execution Units

The execution units are responsible for executing the user-defined functions on demand. `UniFaaS` uses MirageOS [9]. We run these unikernels on top of the Solo5 hypervisor [20] which provides debugging facilities (close to those experienced when debugging standard processes on Linux) and performance optimizations, tailored to support unikernel VMs.

The developer writes a stateless function that will be run as a unikernel. The developer can leverage `UniFaaS`' provided library to interact with the elements of our platforms. For example, `UniFaaS`' API abstracts how to `get`, `set`, `push`, `pop` and `increment` values to and from the event queues and datastores, or to send data to remote end-points.

B. Job Queues

Serverless workloads are *event-driven* and trigger functions executions as the job queues are filled (a given execution unit is associated with at least one job queue). `UniFaaS` operates under a Pub/Sub messaging pattern [24], which allows for the decoupling of producers and consumers. Functions pop events from the queue they listen to and process them, and can push events to other queues. In addition to user-provided functions, remote interfaces (e.g., REST API, MQTT etc.) listen to specific queues and push/pull events received from or to be sent to the outside world (to send and/or receive data). The job queue system is implemented using a Redis server. Access to the job queue system is restricted to a subnetwork only accessible to our executions units and the previously mentioned external interface.

C. Datastore

Although not essential to all workflows, we provide access to datastores that allow us to develop more complex applications. Indeed, as execution units are stateless, the datastores can be used by applications to maintain states. We provide a key-value datastore in our current implementation – although providing access to an SQL database could be possible. We leverage, once again, Redis' in-memory, key-value store function and asynchronously back up the database to disk. The platform states are restored during boot.

D. Orchestration Engine

The orchestration engine is the core component of `UniFaaS` glueing all those pieces together. It is responsible for pairing external interfaces to event queues, instantiating execution units as required, and managing datastores. In this section, we particularly discuss how execution units are managed as the other tasks are more trivial.

Monitoring the event queues: The orchestration engine monitors the event queues to decide when new execution units need to be instantiated or when execution units need to be shut down. To do so it instantiates triggers within the event queues. These triggers notify the orchestration engine when a new element is pushed to an empty queue, when a queue is emptied or when a queue size reaches a certain threshold. These events drive the logic behind new execution instantiations and are fully customisable to improve performance.

Instantiating an execution unit: One of the key features of serverless platforms [23, 25–28], is the ability to scale functions automatically depending on the workload. On noticing a trigger, `UniFaaS` follows a sequence of checks before a new execution unit is instantiated

1) Check that the maximum number of execution units allocated to a given workflow and/or step is not exhausted.
2) That no similar event queue trigger has been witnessed within a given grace period. This is, for example, to give time to a newly-spawned execution unit to boot and start processing events on its associated queue(s).
3) MirageOS unikernels use TAP devices (i.e., virtual network devices) to enable networking capabilities. Each unikernel requires a unique TAP interface to function correctly. `UniFaaS` manages a list of available TAP devices and assigns one to a unikernel before it executes and frees it when the corresponding unikernel terminates.

When/if these steps are satisfied, a new execution unit is instantiated. We effectively implement a "Static Threshold-based" rule for auto-scaling [29] where the number of execution units associated with a job queue is a function of the job queue size. We note that, given the modular nature of `UniFaaS` implementation, it is possible to implement other and potentially more complex scheduling algorithms.

Terminating an execution unit: `UniFaaS` monitors running execution units to keep an up-to-date record of the number of running instances, set the associated grace period to zero when an execution unit is terminated, and account for available TAP interfaces. `UniFaaS`' working assumption is that execution units are good citizens and terminate when their respective event queue(s) is empty. However, when an execution unit hangs or stays idle on an empty queue, it is forcefully terminated.

Similarly, if a given workflow step reaches its maximum number of execution units and the queue continues to grow, the entire workflow is terminated, and an error is logged (this is to avoid a particularly heavy workload exhausting all resources). The developer should either allocate more resource to this particular workload, look for a bug and/or attempt performance optimization.

E. Debugging

Execution units `stdout` outputs can be logged and analysed to understand a workflow/step behaviour. However, debugging serverless platforms in such a fashion is notoriously difficult [26, 30]. Further, as unikernels are isolated, self-contained execution units, they further increase the complexity of debugging activities [20]. In order to alleviate those issues, we leverage the Solo5 hypervisor as it provides `gdb` and `dumpcore` modules to allow for live and post-mortem debugging of Solo5-based execution units.

TABLE I
TABLE SHOWING THE DIFFERENT IOT DEVICES USED IN OUR EXPERIMENTS.

Device	Generate/Consume Data	Invocation Pattern
Philips Hue Connected Bulb	Consume	Burst
Belkin WeMo motion sensor and switch kit	Both	Burst
Nest smoke-alarm	Generate	Repetitive
Withings Smart Body Analyzer	Generate	Single

F. Limitations

MirageOS unikernels do not have TLS support for HTTPS connections. We communicated with the MirageOS team [31] who are actively working on this. We plan to implement TLS support as soon as the feature is available.

IV. EVALUATION

For evaluation of `UniFaaS`, we performed benchmark tests monitoring the resource usage on different sets of hardware. Two different hardware were used: 1) `x86-64 machine` with 16GB RAM, a 6 cores i7 running at 4.1GHz; and a 2) `RPi 4` Model B with 4GB RAM. The evaluation compares the performance and resource usage of `UniFaaS` with the same, or equivalent workload using Apache `OpenWhisk` (we followed `RPi 4` guideline provided by IBM employees [32]) on the same sets of hardware. The evaluation aims to address the following questions:

Q1: Is `UniFaaS` more efficient than `OpenWhisk`?

Q2: Does the `UniFaaS` platform operate with a small footprint?

A. Evaluation Workloads

Due to the proprietary nature of current serverless platforms, there is little public information on production workloads [33]. Sharhad et al. [33] categorised the *types* of workloads typically used on the Azure platform. From this study, we selected applications relevant to IoT devices at the edge. In Table I, we show the different smart-home IoT devices [34] which are used in our evaluation workloads. Our workloads are as follow:

Lights: On receiving data from the Belkin WeMo motion sensor at the front door, `UniFaaS` sends the appropriate signal to the "Philips Hue Connected Bulb" to turn on/off the lights in a house.

Daily Health Statistics (DHS): `UniFaaS` output daily, weekly and monthly progress every day after measuring weight by a "Smart Body Analyzer".

Alert: Send an alert message when smoke is detected by the "Nest smoke-alarm" in the house. `UniFaaS` reads the data and sends an alert to the user if the measured value is above a certain threshold.

Fridge Door Count (FDC): The workload consists of incrementing a counter every time the smart fridge door is opened and producing usage habits for the user.

Additionally, another workload used in this evaluation is calculating the value of the Fibonacci sequence at a given index. This workload is often used for benchmarking the performance of serverless platforms [35–37].

B. Processing performance

In this section, we compare `UniFaaS` and `OpenWhisk` processing speed on both the `x86-64 machine` and the `RPi 4`. We trigger an event one hundred times and report the average time taken to process those events. The results are reported in Fig. 3. Fig. 3(a) shows that `UniFaaS` outperforms `OpenWhisk`.

More interesting are the results on `RPi 4` shown in Fig. 3(b). We notice that `UniFaaS` clearly outperform `OpenWhisk`, but `OpenWhisk`'s performance are unexpectedly poor. Fig. 4 gives a clearer explanation on what is happening. While a majority of events are processed between 11 and 100ms, one event took up to 52s, with a significant proportion taking between 1s and 10s. `OpenWhisk` 90th percentile is $\sim$10s compared to `UniFaaS` $\sim$227ms. We tried different configurations, but `OpenWhisk` had trouble performing consistently on our resource-constrained `RPi 4`. The issue seems to be related to container instantiation being sporadically extremely slow. This makes `OpenWhisk` a poor choice for applications where decent and consistent response latency matters.

C. Memory Footprint

Mohanty et al. [38] recommend the use of simple workload to understand the performance of serverless platform. The FDC workload was selected to test `UniFaaS` due to its simplicity. We increase the workload as to force the system to spawn more execution engines (from 1 to 10) and measure the total memory consumption of the system (including `UniFaaS` itself and all other background task on the system). We average the results over 10 runs. The results are presented in Fig. 5. We can clearly see that the impact of additional execution engine is minimal on the memory footprint (this is not surprising as a unikernel has only a size of a few MB). Higher variance in memory consumption on `x86-64 machine` is due to the higher number of background task on our `x86` server.

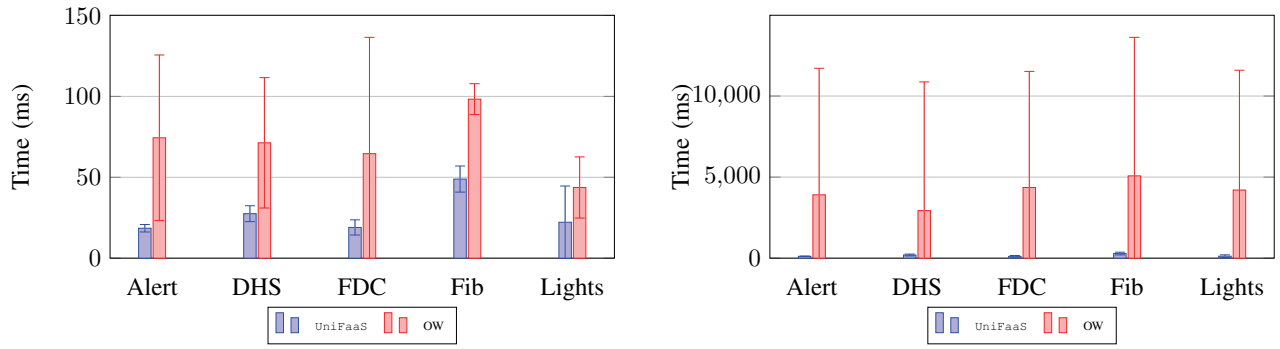
D. CPU Footprint

We proceed similarly to measure the CPU footprint of `UniFaaS`. The results are presented in Fig. 6. We can see a small linear growth as the number of execution engine increase from 1 to 10. `UniFaaS` is efficient and does not impose high CPU load, which is ideal for resource-constrained environments.

E. Boot Latency

In this experiment, we measure the time between the orchestration engine instantiating a new execution unit, and when the execution unit becomes available to process data. For `OpenWhisk` we report both cold and warm boot.

The results in Fig. 7 show that the unikernels used by `UniFaaS` are *three* times faster to boot than warm containers used by `OpenWhisk`. Furthermore, on average `UniFaaS` can boot execution units in $11ms$, which is an order of magnitude faster than cold containers which take $135ms$ to boot.



(a) x86-64 machine (b) RPi 4

Fig. 3. Execution times for the Evaluation Workloads.

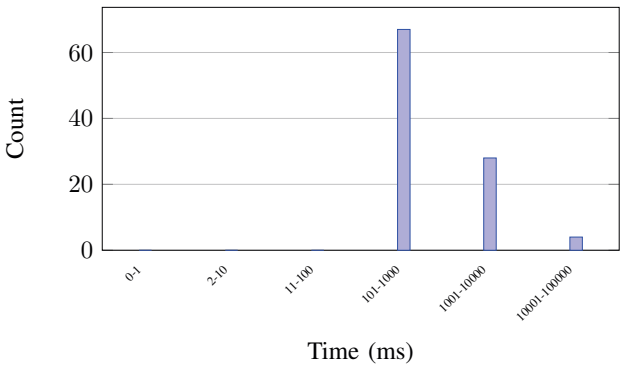


Fig. 4. Execution time distribution for OpenWhisk on RPi 4 for the execution time for the DHS workload.

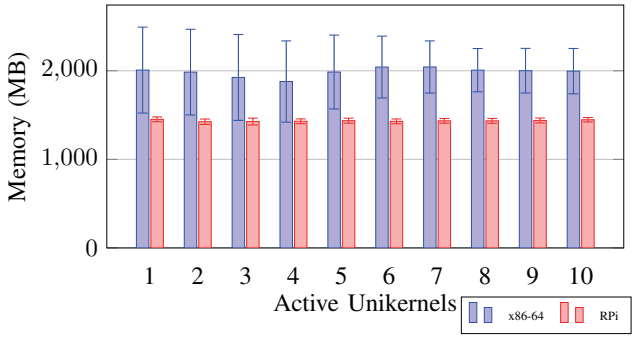


Fig. 5. Memory Usage of UniFaaS when running FDC workload on x86-64 machine and RPi 4.

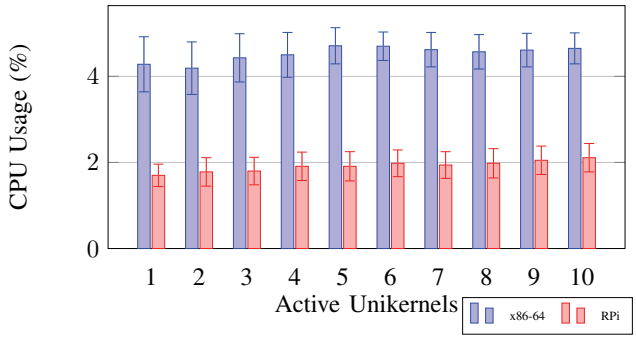


Fig. 6. Average CPU usage of UniFaaS when running the FDC workload on RPi 4 4 and x86-64 machine

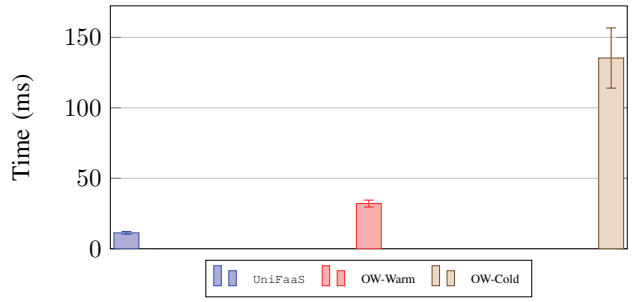


Fig. 7. Average times taken to boot a function on UniFaaS and OpenWhisk on the x86-64 machine

V. Use-Case

To demonstrate the practicality of the proposed solution, we deployed UniFaaS in our indoor pollution monitoring system.

The Smart Citizen Kit (SCK) [13, 14] is a result of the iSCAPE (Improving the Smart Control of Air Pollution in Europe) research and innovation project. The iSCAPE project explored problems surrounding air quality and carbon emissions in relation to climate change. SCK measures air temperature, relative humidity, noise level, ambient light, barometric pressure, and particulate matter (PM). Technology enthusiasts and environmentalists utilise SCK in a variety of ways, for example, to understand indoor/outdoor air pollution and correlation with health issues or to resolve noise issues due to nightlife activities [39].

In our usecase, we monitor the value of carbon dioxide (CO2); if it is around 1000 - 2000 ppm, a person might

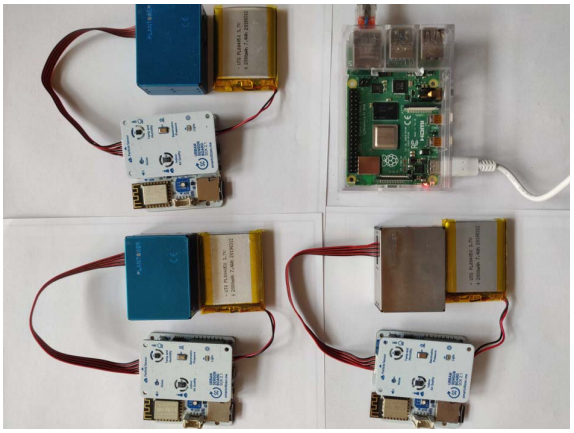


Fig. 8. Our RPi gateway and our Smart Citizen Kits.

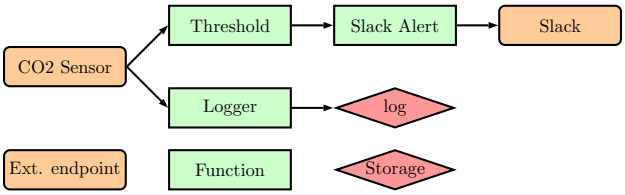


Fig. 9. Our CO2 monitoring workflow.

complain of drowsiness and poor air. CO2 can be decreased by opening the window for cross-ventilation. Our usecase demonstrates `UniFaaS` functionality by processing data generated by SCK on the `RPi 4` and sending alerts to users (in our office environment, we decided to send the alert to our group's `Slack`). The devices used for our experiment are shown in Fig. 8.

We illustrate the workflow associated with this scenario in Fig. 9. The data captured by the SCK is received through an `MQTT` front-end and pushed to an event queue (see § III). Consequently, `UniFaaS` spawns (if needed) a new execution unit to execute the `Threshold` function which retrieves events from the event queue, one-by-one, parses each event's message and extracts the sensor value, compares the value to a predefined threshold and pushes an event to another event queue. The `Slack Alert` function, is associated with this queue. This function, on reception of events, parses each event's message and sends an alert to the Slack organization/channel pair specified in the event. The `Slack function` can be shared across multiple workflows as required. The `Logger` function is used to store sensor readings for future analysis, for example, as part of a research project on indoor pollution.

We evaluated the performance of this usecase workflow. It takes on average 192.64 ms for `UniFaaS` to process an event, while OpenWhisk takes 534.26 ms. Additionally, due to `UniFaaS`'s very low boot overhead (see § IV-E) the variance between execution is only 20.79ms while OpenWhisk exhibits a 227.83ms variance. These results are consistence with our evaluation in § IV.

VI. Limitations and Future Work

Our solution is an early academic prototype and has been evaluated in a limited setting. We are planning to integrate the proposed unikernel-based approach into an orchestration framework such as Kubernetes. The orchestration will help evaluate our solution in a more complex and realistic IoT testbed.

Further, re-implementing existing business logic to run on MirageOS requires to rewrite applications in OCaml. It may be represents a significant engineering effort preventing wider adoption of our proposed solution. We are exploring recent advance in unikernel technology, where legacy OS such as Linux are leveraged to produce unikernels compatible with existing code base (e.g., Lupine Linux [19]). An alternative approach would be to explore the use of transpilation techniques [40] to convert existing code-base. We plan to explore solutions to convert existing container-based FaaS applications to unikernel ones with minimal user involvement.

VII. Related Work

A. Unikernels vs Containers

Several papers compared unikernels with containers [11, 41–44]. Early comparisons [41, 44] used OSv [17], a POSIX-like unikernel. However, supporting POSIX compatibility comes at a cost. Indeed, OSv unikernels achieve approximately half of the memory throughput compared to containers, VMs and native [44].

However, recent comparisons between OSv unikernels and containers offer significantly different results. Goethals et al. [11] performed benchmark tests in different languages (Go, Python, Java). They found that OSv unikernels are 38% faster than containers when running Go applications and 16% faster when running Java applications. Our work confirms that unikernels can significantly outperform container-based solutions and are a prime technology choice.

B. Edge serverless platforms

Several works combine cloud and edge resources [3, 4, 45]. For example, Aske et al. [45] monitors the status of serverless platforms and makes an informed choice of where to execute the function: locally or on a remote cloud service. However, this requires the local serverless system to be compatible with the serverless industry platform. Consequently, they implement OpenWhisk as the local serverless platform and outsource to IBM Bluemix and AWS Lambda. Their results show that their system consistently executes work on the best provider, which is often locally. Furthermore, when the local platform experiences a significantly heavy workload, their system easily outsources the work to the next-best available platform. Such an approach can be seen as complementing our current proposal, either by supporting unikernel-implemented functions on the cloud platforms (we showed superior performance to `OpenWhisk` on X86 hardware § IV) or by leveraging POSIX-like unikernels [17–19] as local execution units.

Hall et al. [7] propose to use WebAssembly rather than containers. Hall et al. observe similar poor performance of

container-based solutions, as containers need to be shut on and off due to limited memory (see § IV-B). Hall et al. rely on Google V8 as their source of isolation. This provides much weaker guarantees than the hypervisor-based techniques that `UniFaaS` leverages. We also note that javascript applications can be used to build unikernels (see https://ops.city/ as an example).

VIII. Conclusion

In this paper we demonstrated that unikernels are a viable alternative to containers when developing serverless platforms aimed at edge devices. Our `UniFaaS` platform outperforms `OpenWhisk` in processing speed, memory consumption and boot-time. We deployed this platform with our Smart Citizen Kit as a use case, thus demonstrating its practicality as a tool for building IoT applications.

Acknowledgement

We thank the anonymous reviewers who helped improve the paper. We thank our colleagues Prof. Jean Bacon from the University of Cambridge and David Eyers from the University of Otago for their feedback.

Availability Statement

We made the implementation available online at https://github.com/cm16161/EdgeKernel under MIT license.

References

[1] J. Lin, W. Yu, N. Zhang, X. Yang, H. Zhang, and W. Zhao, "A survey on internet of things: Architecture, enabling technologies, security and privacy, and applications," *IEEE Internet of Things Journal*, vol. 4, no. 5, pp. 1125–1142, 2017.

[2] G. McGrath and P. R. Brenner, "Serverless computing: Design, implementation, and performance," in *International Conference on Distributed Computing Systems Workshops (ICDCSW)*. IEEE, 2017, pp. 405–410.

[3] S. Nastic, T. Rausch, O. Scekic, S. Dustdar, M. Gusev, B. Koteska, M. Kostoska, B. Jakimovski, S. Ristov, and R. Prodan, "A serverless real-time data analytics platform for edge computing," *IEEE Internet Computing*, vol. 21, no. 4, pp. 64–71, 2017.

[4] A. Glikson, S. Nastic, and S. Dustdar, "Deviceless edge computing: extending serverless computing to the edge of the network," in *International Systems and Storage Conference*. ACM, 2017.

[5] T. Rausch, W. Hummer, V. Muthusamy, A. Rashed, and S. Dustdar, "Towards a serverless platform for edge ai," in *Workshop on Hot Topics in Edge Computing (HotEdge 19)*. USENIX, 2019.

[6] T. Pfandzelter and D. Bermbach, "tinyfaas: A lightweight faas platform for edge environments," in *International Conference on Fog Computing (ICFC)*. IEEE, 2020, pp. 17–24.

[7] A. Hall and U. Ramachandran, "An execution model for serverless functions at the edge," in *Proceedings of the International Conference on Internet of Things Design and Implementation (IoTDI)*. ACM/IEE, 2019, pp. 225–236.

[8] A. Bratterud, A.-A. Walla, H. Haugerud, P. E. Engelstad, and K. Begnum, "Includeos: A minimal, resource efficient unikernel for cloud services," in *International Conference on Cloud Computing Technology and Science (CloudCom)*. IEEE, 2015, pp. 250–257.

[9] A. Madhavapeddy, R. Mortier, C. Rotsos, D. Scott, B. Singh, T. Gazagnaire, S. Smith, S. Hand, and J. Crowcroft, "Unikernels: Library operating systems for the cloud," *ACM SIGARCH Computer Architecture News*, vol. 41, no. 1, pp. 461–472, 2013.

[10] J. Martins, M. Ahmed, C. Raiciu, V. Olteanu, M. Honda, R. Bifulco, and F. Huici, "Clickos and the art of network function virtualization," in *Symposium on Networked Systems Design and Implementation (NSDI)*. USENIX, 2014, pp. 459–473.

[11] T. Goethals, M. Sebrechts, A. Atrey, B. Volckaert, and F. De Turck, "Unikernels vs containers: An in-depth benchmarking study in the context of microservice applications," in *International Symposium on Cloud and Service Computing (SC2)*. IEEE, 2018, pp. 1–8.

[12] X. Gao, Z. Gu, M. Kayaalp, D. Pendarakis, and H. Wang, "Containerleaks: Emerging security threats of information leakages in container clouds," in *International Conference on Dependable Systems and Networks (DSN)*. IEEE, 2017, pp. 237–248.

[13] G. Camprodon, Ó. González, V. Barberán, M. Pérez, V. Smári, M. Á. de Heras, and A. Bizzotto, "Smart citizen kit and station: An open environmental monitoring system for citizen participation and scientific experimentation," *HardwareX*, vol. 6, p. e00070, 2019.

[14] Smart citizen kits. https://smartcitizen.me/.

[15] S. Hendrickson, S. Sturdevant, T. Harter, V. Venkataramani, A. C. Arpaci-Dusseau, and R. H. Arpaci-Dusseau, "Serverless computation with openlambda," in *Workshop on Hot Topics in Cloud Computing (HotCloud)*. USENIX, 2016.

[16] K. Stengel, F. Schmaus, and R. Kapitza, "Esseos: Haskell-based tailored services for the cloud," in *International Workshop on Adaptive and Reflective Middleware*. ACM/USENIX/IFIP, 2013, pp. 1–6.

[17] A. Kivity, D. Laor, G. Costa, P. Enberg, N. Har'El, D. Marti, and V. Zolotarov, "OSv—optimizing the operating system for virtual machines," in *Annual Technical Conference (ATC)*. USENIX, 2014, pp. 61–72.

[18] P. Olivier, D. Chiba, S. Lankes, C. Min, and B. Ravindran, "A binary-compatible unikernel," in *International Conference on Virtual Execution Environments*. ACM, 2019, pp. 59–73.

[19] H.-C. Kuo, D. Williams, R. Koller, and S. Mohan, "A linux in unikernel clothing," in *European Conference on Computer Systems (EuroSys)*, 2020.

[20] D. Williams, R. Koller, M. Lucina, and N. Prakash, "Unikernels as processes," in *Symposium on Cloud Computing (SoCC)*. ACM, 2018, pp. 199–211.

[21] A sandboxed execution environment for unikernels. https://github.com/Solo5.

[22] T. Kim and N. Zeldovich, "Practical and effective sandboxing for non-root users," in *Annual Technical Conference (ATC)*. USENIX, 2013, pp. 139–144.

[23] G. McGrath and P. R. Brenner, "Serverless computing: Design, implementation, and performance," in *International Conference on Distributed Computing Systems Workshops (ICDCSW)*. IEEE, 2017, pp. 405–410.

[24] B. D. Goodman, F. Jania, K. C. Lagarde, C. Shu, and M. Van Der Meulen, "Pub/sub message invoking a subscribers client application program," Feb. 15 2011, uS Patent 7,890,572.

[25] J. M. Hellerstein, J. Faleiro, J. E. Gonzalez, J. Schleier-Smith, V. Sreekanti, A. Tumanov, and C. Wu, "Serverless computing: One step forward, two steps back," *arXiv preprint arXiv:1812.03651*, 2018.

[26] I. Baldini, P. Castro, K. Chang, P. Cheng, S. Fink, V. Ishakian, N. Mitchell, V. Muthusamy, R. Rabbah, A. Slominski *et al.*, "Serverless computing: Current trends and open problems," in *Research Advances in Cloud Computing*. Springer, 2017, pp. 1–20.

[27] L. Wang, M. Li, Y. Zhang, T. Ristenpart, and M. Swift, "Peeking behind the curtains of serverless platforms," in *Annual Technical Conference (ATC)*. USENIX, 2018, pp. 133–146.

[28] E. Jonas, J. Schleier-Smith, V. Sreekanti, C.-C. Tsai, A. Khandelwal, Q. Pu, V. Shankar, J. Carreira, K. Krauth, N. Yadwadkar *et al.*, "Cloud programming simplified: A berkeley view on serverless computing," *arXiv preprint arXiv:1902.03383*, 2019.

[29] T. Lorido-Botrán, J. Miguel-Alonso, and J. A. Lozano, "Auto-scaling techniques for elastic applications in cloud environments," *Department of Computer Architecture and Technology, University of Basque Country, Tech. Rep. EHU-KAT-IK-09*, vol. 12, p. 2012, 2012.

[30] J. Manner, S. Kolb, and G. Wirtz, "Troubleshooting serverless functions: a combined monitoring and debugging approach," *SICS Software-Intensive Cyber-Physical Systems*, vol. 34, no. 2-3, pp. 99–104, 2019.

[31] Online discussion: Sending/receiving https requests with mirageos. https://discuss.ocaml.org/t/sending-receiving-https-requests-with-mirageos/5590/2.

[32] D. Breitgand and P. Kravchenco. Apache openwhisk meets raspberry pi. https://medium.com/openwhisk/apache-openwhisk-meets-raspberry-pi-e346e555b56a.

[33] M. Shahrad, R. Fonseca, Í. Goiri, G. Chaudhry, P. Batum, J. Cooke, E. Laureano, C. Tresness, M. Russinovich, and R. Bianchini, "Serverless

in the wild: Characterizing and optimizing the serverless workload at a large cloud provider," *arXiv preprint arXiv:2003.03423*, 2020.

[34] V. Sivaraman, H. H. Gharakheili, A. Vishwanath, R. Boreli, and O. Mehani, "Network-level security and privacy control for smart-home iot devices," in *International Conference on Wireless and Mobile Computing, Networking and Communications (WiMob)*. IEEE, 2015, pp. 163–167.

[35] J. Kuhlenkamp, S. Werner, M. C. Borges, K. El Tal, and S. Tai, "An Evaluation of FaaS Platforms as a Foundation for Serverless Big Data Processing," in *International Conference on Utility and Cloud Computing*. IEEE/ACM, 2019, pp. 1–9.

[36] T. Nolet. Aws lambda go vs. node.js performance benchmark: updated. https://medium.com/hackernoon/aws-lambda-go-vs-node-js-performance-benchmark-1c8898341982.

[37] J. Jackson. Benchmarking serverless: Ibm scientists devise a test suite to quantify performance. https://thenewstack.io/ibm-scientists-set-quantify-serverless-performance/.

[38] S. K. Mohanty, G. Premsankar, M. Di Francesco *et al.*, "An evaluation of open source serverless computing frameworks." in *CloudCom*, 2018, pp. 115–120.

[39] S. Coulson, M. Woods, M. Scott, D. Hemment, and M. Balestrini, "Stop the noise! enhancing meaningfulness in participatory sensing with community level indicators," in *Designing Interactive Systems Conference (DIS)*. ACM, 2018, p. 1183–1192.

[40] T. Pasquier, D. Eyers, and J. Bacon, "Php2uni: Building unikernels using scripting language transpilation," in *IEEE International Conference on Cloud Engineering (IC2E'17)*. IEEE, 2017, pp. 197–203.

[41] P. Enberg, "A performance evaluation of hypervisor, unikernel, and container network i/o virtualization," Ph.D. dissertation, MS thesis, Faculty Sci., Univ. Helsinki, Helsinki, Finland, 2016.

[42] T. Bui, "Analysis of docker security," *arXiv preprint arXiv:1501.02967*, 2015.

[43] F. Manco, C. Lupu, F. Schmidt, J. Mendes, S. Kuenzer, S. Sati, K. Yasukata, C. Raiciu, and F. Huici, "My VM is Lighter (and Safer) than your Container," in *Symposium on Operating Systems Principles (SOSP)*. ACM, 2017, pp. 218–233.

[44] B. Xavier, T. Ferreto, and L. Jersak, "Time provisioning evaluation of kvm, docker and unikernels in a cloud platform," in *International Symposium on Cluster, Cloud and Grid Computing (CCGrid)*. IEEE/ACM, 2016, pp. 277–280.

[45] A. Aske and X. Zhao, "Supporting multi-provider serverless computing on the edge," in *International Conference on Parallel Processing Companion*. ACM, 2018, pp. 1–6.