NanoLambda: Implementing Functions as a Service at All Resource Scales for the Internet of Things.

Gareth George, Fatih Bakir, Rich Wolski, and Chandra Krintz
Computer Science Department
Univ. of California, Santa Barbara

***Abstract*—Internet of Things (IoT) devices are becoming increasingly prevalent in our environment, yet the process of programming these devices and processing the data they produce remains difficult. Typically, data is processed on device, involving arduous work in low level languages, or data is moved to the cloud, where abundant resources are available for Functions as a Service (FaaS) or other handlers. FaaS is an emerging category of flexible computing services, where developers deploy self-contained functions to be run in portable and secure containerized environments; however, at the moment, these functions are limited to running in the cloud or in some cases at the "edge" of the network using resource rich, Linux-based systems.**

In this paper, we present *NanoLambda*, a portable platform that brings FaaS, high-level language programming, and familiar cloud service APIs to non-Linux and microcontroller-based IoT devices. To enable this, *NanoLambda* couples a new, minimal Python runtime system that we have designed for the least capable end of the IoT device spectrum, with API compatibility for AWS Lambda and S3. *NanoLambda* transfers functions between IoT devices (sensors, edge, cloud), providing power and latency savings while retaining the programmer productivity benefits of high-level languages and FaaS. A key feature of *NanoLambda* is a scheduler that intelligently places function executions across multi-scale IoT deployments according to resource availability and power constraints. We evaluate a range of applications that use *NanoLambda* to run on devices as small as the ESP8266 with 64KB of ram and 512KB flash storage.

***Index Terms*—IoT, serverless, cloud functions, edge computing, microcontrollers, portability**

I. Introduction

The Internet of Things (IoT) provides a data collection and computing fabric for physical objects in our environment. As such, it enables developers to build new classes of applications that integrate with and act upon the world around us. However, with these applications come new data processing and software development challenges. Specifically, developers must either port their code to large numbers of highly resource-constrained, heterogeneous devices (e.g. sensors, actuators, and controllers embedded with microcontrollers), or move their data on volatile, high latency, and energy-consuming long-haul networks to perform computation on resource-rich cloud infrastructures.

Functions as a Service (FaaS) is a technology originally developed by cloud providers to ease the development of scalable cloud and web services [6], [9], [11], [21], [22], [26], [34]. Because of its simple, event-driven programming model and autoscaled deployment, FaaS is increasingly used for IoT applications [8], [12], [13], [23], [36]. With FaaS, developers decompose their applications into multiple functions, each responsible for some short-lived piece of processing. A function is typically a small, single-entry code package (50 megabytes or less) written in a high level language (e.g. Python, node.js, or Java). Thanks to their small size and self-contained design, these functions are easily invoked and scaled automatically by the FaaS platform in response to incoming events (data arrival, messages, web requests, storage updates, etc.). Many functions, potentially belonging to different tenants, execute on a single server, using Linux containers to provide secure isolation and a consistent execution environment. Together, these features allow the application developer to focus on the function logic using a single, familiar programming environment, without having to port code to heterogeneous architectures, to consider where functions execute, or to manage how and when to scale their applications.

To support IoT, some FaaS platforms are now able to move data from devices at the "edge" of the network to the cloud [8], [13], [23]. Given high latency and intermittent connectivity of these networks, other FaaS alternatives offer edge extensions that instead move the functions to the edge (which are typically much smaller in size than data) to consume and operate on data "near" where it is produced. Performing computations at the edge reduces application response latency and device power consumption, e.g. many IoT devices are battery powered and radio use consumes significant energy [16], [19], [20], [24], [33], [35], [36].

Although a step in the right direction, these edge-aware FaaS systems still impose a significant burden on IoT developers. Specifically, FaaS functionality in most cases is limited to Linux-based edge systems – precluding the use of a uniform programming methodology on non-Linux devices (i.e. microcontroller-based sensors and embedded systems). To integrate such devices, existing platforms use multiple technologies and protocols [10], [12], [13], [16], [23], [24]. For example, an IoT applications developer that uses AWS Lambda [9] and Greengrass (i.e. AWS Lambda at the edge) [24] for edge-cloud interoperation must configure MQTT [15], FreeRTOS, IoT SDK [25], AWS Lambda, AWS Greengrass, and a persistent storage service (typically S3 [4] or DynamoDB [3]), in addition to the function itself. Thus, IoT application development requires significant expertise in a myriad of programming technologies and styles that is further complicated by differing systems capabilities, resource restrictions, quality of service, and availability [10], [16].

CSPOT attempts to overcome these challenges via a low-level, distributed operating system that implements a FaaS execution environment for C-language functions across all tiers of scale in IoT [36]. *CSPOT* provides the same event-driven deployment and execution model across systems ranging from clouds to microcontrollers, to hide heterogeneity. It exports a distributed storage abstraction that functions use to persist and share state. *CSPOT* supports execution of functions written using high level programming frameworks and cloud APIs via Linux containers and C-language bindings. However, these latter capabilities (i.e. high-level language support, function isolation, and interoperability with other FaaS systems) does not extend to microcontrollers or any non-Linux system.

The key challenge with extending these capabilities to microcontroller-based systems, which all FaaS systems must face, is supporting the vast heterogeneity of such devices coupled with their limited computational resources. Sensors and embedded systems implement different hardware architectures, operating software, and peripherals, which make them difficult to program, to provide abstractions for, and to leverage using existing APIs and programming frameworks. Limited compute, storage, and battery resources severely constrain what programs run on these devices. Finally, most devices lack the physical hardware (e.g. memory protection units, sufficient volatile and non-volatile storage, etc.) that is required to run conventional FaaS software stacks.

In this paper, we overcome these challenges to extend FaaS uniformly to the least capable of IoT devices. To enable this, we design and implement a distributed FaaS platform called *NanoLambda* that runs over and integrates with *CSPOT*. By doing so, we leverage *CSPOT*'s FaaS programming and deployment model, its fault resilient distributed storage abstraction, and cloud and edge portability. *NanoLambda* raises the level of abstraction of *CSPOT* to support execution and isolation of Python functions on non-Linux systems. By doing so, *NanoLambda* makes Python FaaS functions ubiquitous and portable across all tiers of IoT (across sensors, edge systems, and clouds) and facilitates programmer productivity across these heterogeneous systems.

NanoLambda is also unique in that it is tailored to the FaaS programming model and mirrors the APIs of popular cloud-based FaaS systems. By optimizing resource use and power consumption, *NanoLambda* facilitates efficient execution of Python functions on severely resource constrained devices. By implementing popular FaaS and storage APIs from the de facto standard AWS Lambda [9], *NanoLambda* brings cloud-based FaaS familiarity to IoT programming, and facilitates reuse of existing FaaS applications in IoT settings.

The *NanoLambda* platform consists of Edge/Cloud components (called *NanoLambda Cloud/Edge*) and microcontroller (i.e. on-device) components (called *NanoLambda On Device*) that interoperate. *NanoLambda Cloud/Edge* runs on Linux-based systems, employs a containerized runtime, executes FaaS functions directly (those targeting the local host), and provides a translation service for functions that target neighboring, non-Linux devices. The translator transforms Python functions, emulating any AWS Lambda and Simple Storage Service (S3) API calls, using compact and efficient bytecode code packages that are executed by *NanoLambda On Device*. Our design of the translator is sufficiently small and efficient to run on even resource-constrained Linux-based edge systems (e.g. single board computers like the Raspberry Pi) .

NanoLambda On Device is designed without dependency on Linux interfaces or a memory protection unit for function isolation. To enable this, it integrates a from-scratch, novel redesign of a Python interpreter called *IoTPy*. *IoTPy* defines its own bytecode language (precluding the need for recompilation for different devices) and code packaging system (which the *NanoLambda Cloud/Edge* translator targets). *IoTPy* also implements FaaS- and device-aware optimizations that efficiently implement *CSPOT* capabilities while staying within the memory limitations of highly resource constrained IoT devices. Our design of *IoTPy* is motivated by our observation that existing Python interpreters, including MicroPython [30], are too resource intensive for increasingly ubiquitous IoT devices that we wish to target (e.g. the Espressif ESP8266). With the combination of these components, *NanoLambda* achieves AWS compatibility and efficient FaaS execution end-to-end.

In the sections that follow, we describe *NanoLambda* and the design and implementation of its components and features. With this hybrid edge-device service ensemble, this paper makes the following contributions:

- We explore the feasibility and performance of a FaaS service for running high level Python functions on highly resource restricted devices (e.g., microcontrollers). We evaluate the efficacy of our system using a "real world" implementation. We also provide microbenchmarks which delve into the performance trade-offs of running FaaS on-device and show that for a variety of suitable applications it can provide improved latency and power efficiency.
- We demonstrate portability and the possibility of ubiquity – executing the program on device, at the edge, or in the cloud – using *CSPOT* [36], a distributed runtime (designed to support FaaS) to implement a multi-scale compatibility service for FaaS that is API-compatible with AWS Lambda and S3.
- We present a simple scheduler that uses this portability to select locations for function execution based on latency constraints, and we show that the scheduler outperforms either remote-only execution or on-device-only execution across a range of problem sizes.
- We achieve isolation between untrusted FaaS functions on non-Linux IoT devices using a high-level language virtual machine for Python as an alternative to heavyweight containerization.

As a result, this work enables the use of a single FaaS programming model and set of APIs across *all* resource scales in an IoT deployment while leveraging the considerable existing code base and technological footprint of AWS Lambda.

II. Related Work

Functions as a Service (FaaS) is a cloud-computing platform with which developers write code without consideration for the hardware on which it runs. Applications are broken down into distinct units of functionality and packaged as functions that can be distributed and updated on the fly. Functions are executed in lightweight containers which provide portability, scalability, and security: functions are automatically scaled across multiple machines, while execution is isolated from other tenants of those instances. A number of major cloud providers such as AWS Lambda [9], Google Cloud Functions [22], and Microsoft Azure Functions [11] now offer hosted FaaS platforms.

Because of its automatic scaling and convenience, FaaS has seen increased adoption in IoT in recent years. AWS IoT, for example, leverages FaaS functions extensively in order to allow users of the platform to attach functions that handle IoT events [17]. Serverless [34], a leading industry toolchain for FaaS programming, estimates that IoT makes up 6% [1] of the use of FaaS in industry.

An emerging development in IoT data processing is the prospect of FaaS at the edge for data processing. Commercial offerings such as AWS GreenGrass [24] and open source Apache OpenWhisk [6] make it possible for users to host their own FaaS deployments on low-power devices. By running functions at the edge, the hope is that handlers can be run closer to where data is produced, making better use of locally-available processing and reducing latency penalties and transfer costs. The authors of [31] explore the concept of a serverless platform for real-time data analytics at the edge and discuss the benefits and challenges of the approach.

CSPOT is an open source, low-level distributed operating system for IoT that executes a uniform FaaS model across devices ranging from servers in the cloud to resource-constrained sensors and microcontrollers [36]. *CSPOT* exports a distributed storage abstraction called a WooF for persistent storage of FaaS function objects. WooFs are append-only logs with built-in event tracking and repair capabilities [28]. WooFs serve both the role of state storage and execution log – appending data to WooFs optionally can trigger the execution of an event handler that responds to the appended data. Functions (i.e. event handlers) are implemented as Linux executables allowing for flexibility of language choice and libraries. A pool of docker containers is maintained to run handlers – by running only one handler in a container at any given time, functions are isolated from one another. This enables development of complex applications, and provides the security, portability, and scalability expected of FaaS systems.

For Linux systems, *CSPOT* provides an execution environment and AWS API compatibility (for S3 and Lambda services) for Python functions. However, this support does not extend to microcontrollers (which we address with the work herein). To execute C-language functions on microcontrollers, *CSPOT* uses a threaded, multitasking system with direct memory addressing. This support is available for ARM, AVR, and Espressif microcontrollers. FaaS functions in *CSPOT* must be compiled and installed on a per-device basis. Our work enables Python functions (including Python AWS Lambda functions) to be executed over *CSPOT*, including on microcontrollers. We do so using a combination of automatic translation at the edge and interpretation on-device to enable both portability and efficiency.

When functions can be made highly portable, as is the case with FaaS, interesting possibilities for improving performance emerge, such as scheduled execution. The authors of [7] propose a framework for scheduling execution among multiple FaaS providers. The authors develop an extensible API for executing functions across multiple providers, as well as a metrics database that tracks execution statistics from these services. Once a model of function performance has been built for a given provider, this information can be provided to a user-defined scheduler, e.g., a Python function which selects the provider with the minimum latency. The authors find that by employing a scheduler, they can achieve a 200% faster average round-trip execution time compared to executing on any one service provider. This serves as a promising example of how the portable nature of FaaS functions can provide application developers with more flexible execution options and improved performance.

III. *NanoLambda* Design and Implementation

At present, writing applications for IoT means programming for APIs and libraries specific to each device, writing functions that are "locked in" to particular cloud provider APIs, and using only those devices that are supported by these platforms. It is our objective to support a common programming model everywhere, extending from IoT devices to the edge to the cloud. In service of this goal, we opted to design our system around *CSPOT*, a low-level framework for supporting IoT FaaS. *CSPOT* provides the foundation for our high-level FaaS runtime, with powerful cross-platform primitives for secure function (event handler) execution and data persistence.

However, existing cloud-based FaaS offerings that have seen adoption in industry use high level languages, such as Python, for their handlers. These high-level languages are particularly advantageous for portability and programmer productivity. The *CSPOT* service, while capable of running with minimal overhead on a wide range of severely resource-constrained devices, defines its own API. Thus while *CSPOT* is, itself, portable, it cannot execute FaaS functions written for existing cloud services such as AWS Lambda.

It is our view that, in order to provide the same convenience, compatibility, and security as FaaS handlers executing in the cloud, a FaaS system for resource-restricted, non-Linux devices should have the following properties:

1) Ease of development – we wish for *NanoLambda* to be familiar to developers already using FaaS, ideally allowing existing FaaS functions to run with little to no code changes.

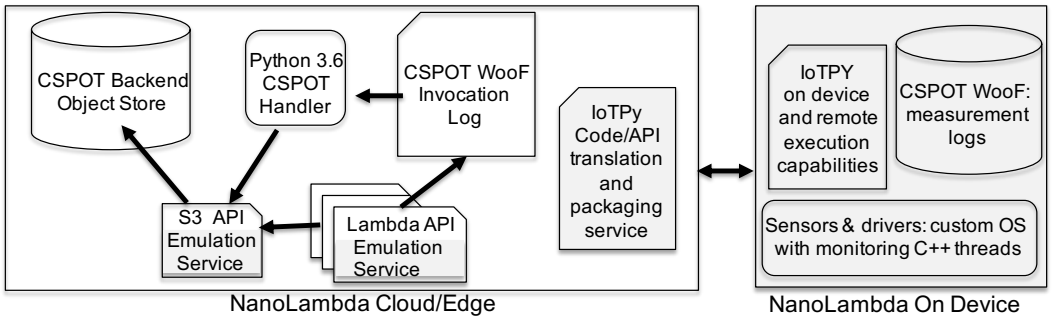

Fig. 1. Service diagram outlining the interactions of the *NanoLambda* components that deliver FaaS for IoT. Shaded components are those added to *CSPOT* by *NanoLambda*. *NanoLambda Cloud/Edge* runs on Linux hosts in the cloud or at the edge; *NanoLambda On Device* runs on microcontrollers and non-Linux systems.

2) Portability – we want portable application code so that the same implementation can run in the cloud or on the device with little to no additional developer effort.
3) Small code and memory footprint – *NanoLambda* should leave as much storage and memory space as possible available for libraries and C extensions.
4) Security – IoT devices can be particularly vulnerable to buffer overflow, stack overflow, and other classes of exploits [2] [29], which we must protect against.

Figure 1 shows the architecture of the system we have developed to meet these goals. We deliver on our goals with two core systems built on *CSPOT*:

- *NanoLambda Cloud/Edge*, shown on the left in the figure, provides FaaS handler execution to Linux capable devices as well as object storage. It doubles as a centralized repository from which IoT devices can request functions.
- *NanoLambda On Device*, shown on the right in the figure, provides on-device handler execution capabilities with *IoTPy*, a Python interpreter tailor-made for FaaS handler execution.

This separation of concerns is primarily driven by the needs to tailor our device implementation to best perform within the constraints of low power non-Linux IoT devices. This allows us to offload complex and processing-intensive tasks to *NanoLambda Cloud/Edge* while keeping *NanoLambda On Device* space- and power-efficient.

A. NanoLambda Cloud/Edge Service

Figure 1 illustrates the *NanoLambda Cloud/Edge* service on the left. The core of this service consists of two REST API servers, which export (i) an object store compatible with the Amazon Simple Storage Service (S3) [4], and (ii) a FaaS service that deploys FaaS functions written for AWS Lambda [9] over *CSPOT*– in the same way they are deployed over AWS.

To enable this, the servers leverage multiple *CSPOT* primitives. This includes an object store and event model in which handlers are triggered in response to object updates. *CSPOT* objects are fixed-size records in append-only logs. Handlers execute inside Linux containers allowing for concurrent execution of handlers as isolated processes. The *NanoLambda Cloud/Edge* API servers build upon these primitives and *CSPOT* support for a subset of the S3 and AWS Lambda APIs (for Linux systems), to unify deployment and execution of AWS Lambda Python functions across Linux and non-Linux systems.

Specifically, the S3 API Emulation Service provides a wrapper for the *CSPOT* append-only object store. It stores blobs to *CSPOT*'s fixed-size log records by first splitting them into 16KB chunks and then linking these chunks together with metadata records. It then stores a mapping between the blob's key and the location of the first chunk separately in an index log. Blobs are retrieved via a sequential scan of the index log to find their location. Once located they are read back in chunks and streamed to the requester. This service exports this functionality via the Amazon S3 REST API [5].

The Lambda API Emulation Service manages function deployment and facilitates function execution in *CSPOT*. The service stores the configuration and code package using the S3 API Emulation Service, making it a single store for all persistent data. By offloading state management to a centralized object store, these services make it possible to run load-balanced instances across multiple physical machines, to improve performance and scalability.

AWS Lambda compatibility is provided via a special *CSPOT* handler binary, which embeds a Python interpreter. When an function invocation request is received, *NanoLambda Cloud/Edge* appends an invocation record to *CSPOT*'s object store. This append creates an event, which invokes this handler in an isolated docker instance. The handler reads the invocation record and checks for the presence of the Python code for the handler. If necessary, the code is fetched from the object store. Once ready, the Python interpreter invokes the handler function, with the payload from the invocation record, and writes the result to a result log for return to the requester.

To support handler execution on device *NanoLambda Cloud/Edge* implements an *IoTPy* translator (cf Section III-C), which compiles and packages Python handlers for execution on neighboring microcontroller devices. This translation service enables IoT devices to fetch handler code from the same function repository used by the Lambda API Emulation

Service. The service compiles handlers on-demand into a compact binary representation that can be sent to the device for execution. Fetch requests check that the code size and memory requirements (specified in the function configuration) fit within the resource constraints of the requesting device. If rejected, it is possible for the cloud/edge device to execute the function on behalf of the IoT device.

B. NanoLambda On Device

NanoLambda On Device is runs Python handlers on non-Linux IoT devices. As shown in Figure 1 (on right), the service leverages *CSPOT* WooF logs on-device for persistent storage and event invocation. As for Linux systems, FaaS handlers are invoked in response to data appended to storage logs.

As seen in the figure, the typical model for *NanoLambda On Device* is that data is produced by threads monitoring sensors on the device and appending it to objects in the object store. It is also possible for data to be remotely delivered for processing over *CSPOT*'s network API. Each append to the object store is processed by *CSPOT*'s event model which runs a C-language handler function with the new data. Similar to the *NanoLambda Cloud/Edge*'s support for Python using a special Linux handler binary with an embedded Python interpreter, we extend *NanoLambda Cloud/Edge* with Python support by registering a special C-language handler function which interprets the handler's name to be a Python Lambda function deployed via the *NanoLambda Cloud/Edge* service. When this function is invoked, it triggers the handler with the payload provided by the newly appended object in an instance of the *IoTPy* VM. By leveraging *CSPOT*'s existing functionality, we are able to offer Python 3 Lambda support on device.

C. IoTPy

To ease development and portability, we choose Python as our FaaS programming language because its bytecode is easily portable and its simple virtual machine implementation can be easily ported to IoT devices. Likewise, we package FaaS functions using the package format defined for AWS Lambda for compatibility purposes.

To achieve a small code and memory footprint we opted to implement *IoTPy*: our own extremely lightweight Python interpreter built from the ground up with embedding in mind. We found existing solutions, such as MicroPython [30], to be unsuitable because they left little additional space for features such as our FaaS runtime and the drivers for our networking and sensors. On our most resource-constrained devices, such as the ESP8266, our binary must be able to fit in as little as 512KB of flash memory.

IoTPy is written to be as unintrusive as possible with a focus on allowing the Python interpreter to be treated like a library. To keep *IoTPy* small, we opted to omit the Python parser/lexer on the device. Instead, these tasks are offloaded to the *NanoLambda Cloud/Edge* service. This benefits us in a number of ways:

1) bytecode compilation as well as optimization steps can be performed by a full Python 3.6 implementation on *NanoLambda Cloud/Edge* service
2) memory use is reduced since we are only shipping compact bytecode representation to the device, and
3) a simpler virtual machine implementation can be used on device – by delegating compilation to the *NanoLambda Cloud/Edge* service, our *IoTPy* is able to limit its functionality to implementing the core language features of the Python virtual machine (VM).

Together, these capabilities mitigate the additional complexity of maintaining a simple Python bytecode VM. To enable these capabilities we make some tradeoffs with regard to program storage and architectural simplicity. *IoTPy* provides a convenient C/C++ interface for extending Python with native functions. We currently provide implementations for only the most common Python libraries and emulation of only the AWS S3 and Lambda deployment services. The libraries include common math functions, JSON operations, APIs for accessing device peripherals, basic networking, and interfaces for the APIs provided by *NanoLambda Cloud/Edge*. *IoTPy* cannot, however, load standard Python packages.

When comparing the size of the generated binary files for *IoTPy* and MicroPython, we find that the code for the *IoTPy* library uses 290KB of flash storage, whereas MicroPython requires 620KB of flash storage. Thus the *IoTPy* implementation leaves us with almost 300KB of additional storage available for programs, drivers, and the FaaS runtime.

D. Python VM Security and Isolation for FaaS Handlers

IoT devices are often so resource-constrained that they lack security features such as memory protection units to implement separate address spaces for isolated execution of user code. Additionally, conventional FaaS isolation, achieved by Linux features like container-based isolation, is simply not possible on devices like these. The authors of [2] and [29] find various attacks, many of which rely on out-of-range memory access, to exploit IoT devices and execute code or update their firmware with malicious payloads. Yet it is essential that IoT developers be able to reprogram and update their devices in the field.

By placing the majority of application logic in FaaS handlers, we reduce potential attack surface area. A secure Python virtual machine implementation can ensure that the bytecode running within it cannot access memory out of range or otherwise escape the permissions allowed to the handler by the built-in functions provided to it. Python handlers can then be isolated from one another by running in separate instances of the Python VM. This isolation is similar to traditional Linux containers.

E. Deploying and Running Functions

By leveraging *NanoLambda Cloud/Edge* and its compatibility layer for Amazon's AWS Lambda APIs, we minimize the effort to port functions to the *IoTPy* platform. In many cases, functions run directly on *IoTPy* without code changes.

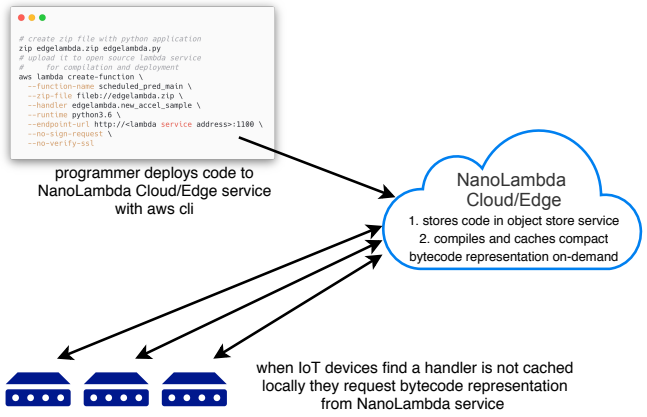

Fig. 2. Life cycle of a function deployment from the application developer's command line to the *NanoLambda Cloud/Edge* service to the IoT device running *NanoLambda On Device*

Function deployment is performed by developers using the aws-cli or the aws-sdk (using a *NanoLambda Cloud/Edge* device as the cloud target), meaning existing tools capable of deploying to AWS Lambda can also deploy to *NanoLambda Cloud/Edge* with few changes. Deploying a new version of an application is done with the "aws lambda create-function" command, which accepts the application code package as a zip file, as well as other configuration parameters including the function entry-point, its memory limit, and Python version (Python 3.6 is currently supported). The deployment process checks code size and resource limits (as is done in AWS Lambda) and also checks for use of unsupported AWS APIs in the code (e.g. only AWS REST S3 API is currently supported). The validated function is stored in the S3 API Emulation Service as described above.

Function execution on device is performed by *IoTPy*. Local invocations are accelerated by an interpreter cache, which allows the *IoTPy* interpreter for a given handler to be reused if it is still available from a previous invocation. This is done to avoid the relatively expensive process of reinitializing Python built-ins as well as functions and globals defined in the bytecode.

In Figure 2, we show the deployment and function download process mapped out for *NanoLambda On Device*. When the interpreter is still running (i.e. because it is already running a function), execution involves simply setting up a new Python frame with the call stack for the function and invoking the function in the cached interpreter. When no interpreter is running, the *NanoLambda* service opens a TCP connection to *NanoLambda Cloud/Edge* and serializes a request for the function using flatbuffers [18], a fast and memory efficient binary protocol for network serialization. When *NanoLambda Cloud/Edge* receives this request, the function is fetched from our object store and compiled on the fly, using the Linux implementation of Python 3.6 and the Python dis package to extract generated bytecode.

F. Execution Offloading Capabilities

Our *NanoLambda On Device* service provides direct handler code compatibility with the *NanoLambda Cloud/Edge* service. We achieve this by adopting the same AWS Lambda-compatible handler format.

NanoLambda On Device supports local execution of handlers on microcontrollers like the ESP8266 and the CC3220SF. *NanoLambda Cloud/Edge*, leveraging *CSPOT*'s tiered-cloud approach, extends these execution options to include edge cloud nodes like the Raspberry Pi Zero as well as in-cloud solutions like *CSPOT* running on AWS EC2. Lastly, because both services use a common function format which is code-compatible with AWS Lambda, our handlers can be easily transferred between the services, or even executed on AWS Lambda itself. Selecting to develop application logic in compartmentalized functions using a high-level language lends the developer immediate portability benefits, which grant her the power to choose where code should be run to best utilize resources or achieve desired performance.

IV. Evaluation

In this section, we evaluate *NanoLambda* on two example FaaS applications for sensor monitoring. We provide a performance analysis of *NanoLambda* using micro-benchmarks to expose a detailed analysis of its performance characteristics under its various configurations. Additionally, we show how *NanoLambda*'s portability can be leveraged to implement a latency-aware scheduler that migrates execution between the device and the edge cloud to improve performance across a range of problem sizes.

A. Automatic Light Control Application

We show how *NanoLambda* might be used in an automatic lighting control system – for example, to automatically turn on walkway lighting at night based on ambient light levels. The hardware setup for this experiment is a photoresistor connected to the analogue-to-digital converter (ADC) pin of an ESP8266 microcontroller. This processor is extremely resource-constrained, with 80KB of user data RAM and 512KB of flash storage for code. The only device-specific C code written for this experiment is a simple sampler thread, which measures the voltage on the ADC pin of the microcontroller and writes it out to an invocation log at a rate of 5 samples per second.

In response to each log append (analogous to publishing to an MQTT queue), an *IoTPy* handler is invoked to analyze the new data and send commands to the light system if necessary. The implementation of the handler for this automatic light control experiment is shown in Figure 3.

```
def new_light_lvl(payload, ctx):
    if payload[0] > 100:
        device.set_pin_high(2)
    else:
        device.set_pin_low(2)
```

Fig. 3. Handler function for the light control application

In Table I we show invocation latencies for various configurations of the *NanoLambda* runtime. The "No Server Cache" configuration represents the true "cold start" cost of invoking a new AWS Lambda function for the first time. This number includes both the time taken for the server to compile the Lambda function and to deliver it to the device. The Local Cache configuration shows us the typical amortized runtime of the application where frequently invoked Lambda functions will typically be already available on the device. We can see that the performance difference is substantial, requiring almost 32x as much time to invoke a function that is not in cache. This is largely due to the effects of both compilation time and the cost of network latency. Compilation and code transfer each take roughly 100 milliseconds on the network tested. We include "No Local Cache" to show the startup cost of a function shared by multiple devices. In this scenario, a function typically has already been accessed and compiled by the server but has not yet been delivered to a given device. We expect this to be representative of the startup cost for real-world deployments.

We include a C implementation of the same handler function as a baseline. It is worth noting that this is on the order of 20x faster than our configuration with local caching. This is a substantial overhead, but it is representative of what we would expect for the performance difference between an interpreted language without JIT and a compiled language like C [32]. Ultimately, it is up to the application developer to determine when the flexibility and ease of deployment with Python and the FaaS ecosystem outweigh its overhead.

Configuration	Latency in ms
NanoLambda Local Caching	6.7 ms
NanoLambda No Local Cache	119.5 ms
NanoLambda No Server Cache	220.4 ms
NanoLambda C Handler	0.3 ms

TABLE I
AVERAGE LATENCY FOR 100 ITERATIONS OF THE HANDLER FUNCTION

B. Predictive Maintenance Application

In this experiment, we look at the detailed performance of a *NanoLambda* application with more demanding processing requirements compared to the previous example. Predictive maintenance techniques use sensors to detect part failures or other maintenance requirements.

Our experiment considers motor maintenance by analyzing the vibrations off of a motor as measured by an accelerometer. It is possible to detect if the motor or a connected part has failed by monitoring for changes in the distribution of the magnitude of vibrations picked up by the accelerometer. Our application detects these changes with a Python implementation of the Kolmogorov–Smirnov (KS) test, used to compare a reference distribution against real time measurements from an accelerometer. The test provides the probability that the empirical distribution matches the reference distribution. A possible failure can be flagged if this probability drops below a tuned threshold.

```
# list containing a reference
# distribution from accelerometer
reference = [...]

def kstest(datalist1, datalist2):
# omitted, see implementation from
# gist.github.com/devries/11405101

def new_accel_sample(payload, ctx):
    global reference
    transformed = []
    for record in payload:
        transformed.append(
            magnitude(record)))
    prob = kstest(
        transformed,
        reference)
    return prob
```

Fig. 4. The implementation of the handler providing the KS test results

In Figure 4, we show the implementation of our KS test handler. The handler is provided with a JSON payload containing a list of recent accelerometer data as tuples of (x, y, z) acceleration. These records are normalized to a transformed array of floating point magnitudes and then passed to an "off-the-shelf" open-source implementation of the KS test by Christopher De Vries [27]. This KS test implementation is computationally intensive and scales linearly with the input problem size.

In Table II, we show the performance of a real-world installation. The hardware configuration is a CC3220SF processor with 1MB of program flash and 256KB of RAM. This processor is wired to an accelerometer monitored by a sampler thread (much like the photoresistor experiment), which collects data from the accelerometer 5 times per second and appends it to a data log. Every 32 samples, a handler is invoked to analyze recent data. The handler is always invoked on device and shows the performance of a more complex handler running on *IoTPy*. The more complex function requires longer execution time, $147ms$, as well as a longer startup time of $436ms$ to compile.

We found this handler to be sufficiently computationally intensive to be a good candidate for offloading to an edge cloud device. To this end, we next present a scheduler for offloading computationally-intensive tasks to remote FaaS providers.

C. Saving Power With Execution Offloading

In Figure 5 we show the power used by the CC3220SF microcontroller for KS problem sizes 20 and 80 over the

Configuration	Latency (ms)	Power Use (mJ)	Memory Use KB
NanoLambda Local Caching	209.4 ms	21.23 mJ	23.6KB
NanoLambda No Local Cache	729.0 ms	85.00 mJ	21.7KB

TABLE II
AVERAGE TIMES/POWER CONSUMPTION OVER 100 ITERATIONS OF THE HANDLER FUNCTION ON THE CC3220SF MICROCONTROLLER WITH A KS PROBLEM SIZE OF 32 IN EACH CONFIGURATION.

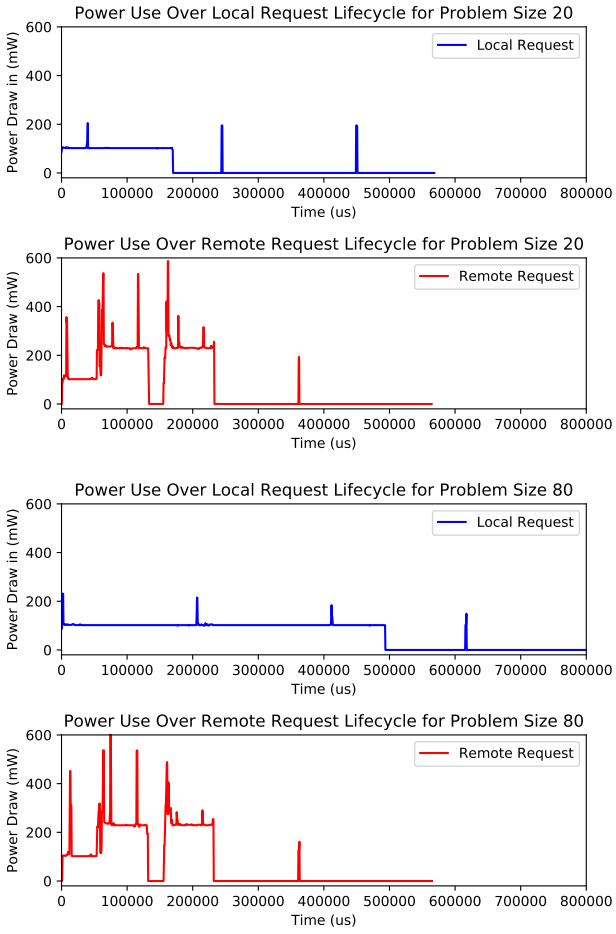

Fig. 5. Comparison of power draw over request lifecycle for various KS problem sizes using both remote and local strategies.

life cycle of a request. We observe here that, with the exception of some power spikes due to background activity on the microcontroller, the power used by the local request is continuous from the beginning of the invocation until the result is available, and so it grows with the size of the problem (plus some constant overhead from the FaaS runtime). With remote execution, we see just over twice the peak power draw of the local execution strategy, but we see a trough in the middle of the request life cycle, where power use drops to near zero. This is because the device is free to completely idle while waiting for results to return from the server. Therefore, we pay a large but short-lived power cost during transmission, but when enough computation can be performed in the cloud, we can save power overall relative to local execution.

D. Predictive Maintenance Application with a Naive Offloading Scheduler

In this application we take advantage of our code compatibility between *NanoLambda Cloud/Edge* and *NanoLambda On Device* to extend our predictive maintenance application running on device with the capability to dynamically offload computation to the edge.

Our scheduler offloads computation when its algorithm estimates that better latency can be achieved by transferring execution to the cloud. We model execution latency by checking the runtime of the last execution on *NanoLambda On Device* and comparing it against the round trip latency of the last invocation on *NanoLambda Cloud/Edge*. The service with the lowest total latency is selected to run the handler.

However, as conditions change, execution-time measurements can become stale. Much like the authors of [7], we periodically retest each possible execution strategy to allow the scheduler to sense when the relative performance has changed, e.g. as a result of evolving network conditions or changes in problem complexity. In this experiment, our scheduler is configured to retest the losing strategy 1 out of every 16 invocations.

To benchmark the scheduler, we measure the average invocation latency for a range of synthetic problems from 5 data points to 100 data points. We examine problem sizes at multiples of 5 with 1000 trials at each size.

Figure 6 shows average invocation latency as a function of problem size for the scheduler, a remote-only invocation strategy, and a local-only invocation strategy. Local invocations provide very consistent runtime, which scales linearly with the problem size. Remote invocation, on the other hand, is highly subject to network latency and congestion, as we can see from occasional spikes. The scheduled invocations appear to offer the best of both worlds, scaling linearly until the latency exceeds the average network latency, at which point the strategy switches to remote invocation.

Figure 7 shows average power per invocation as a function of problem size for the scheduler, the remote-only invocation strategy, and the local-only invocation strategy. We see similar trends as the previous the latency graph – remote execution uses relatively constant power while local execution power consumption scales with execution time. We see, again, that the scheduler offers the best of both strategies. Interestingly, however, even though the scheduler begins to switch to remote invocations at problem size 40, we do not observe power savings from this until the problem size reaches 60.

A more detailed breakdown of the choices made by the scheduler strategy are shown in Figure 8. The top of this chart shows the execution latency while the bottom shows the number of calculations run locally versus the number offloaded over the network. Initially all calculations run on device for small problem sizes; however, once the problem size exceeds 40, it becomes intermittently worthwhile to run over the network when latency is low. After the problem size exceeds 60, almost all invocations run remote, as the speed difference between on-device execution and remote execution outweighs the latency cost of transferring data over the network. We observe, however, that this naive scheduling algorithm leaves some room for improvement. First, with a fixed re-testing latency we allocate more cycles to re-testing than may be necessary. As we can see in this graph, the scheduler is always spending $60/1000$ cycles re-testing the losing strategy. When there is a large difference in performance, this can be

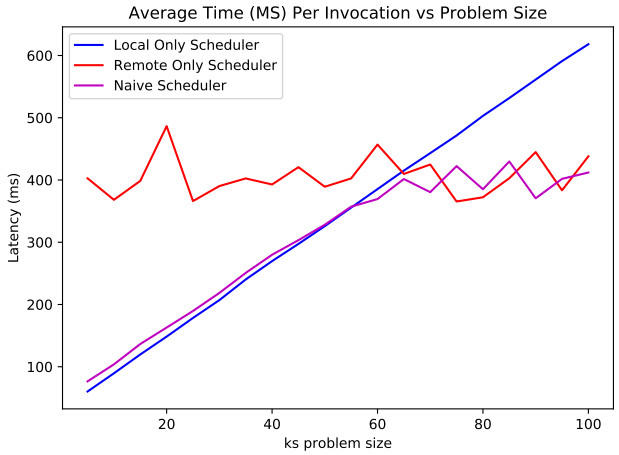

Fig. 6. Comparison of average invocation latency for local invocation strategy, remote invocation strategy, and offloading scheduler invocation strategy.

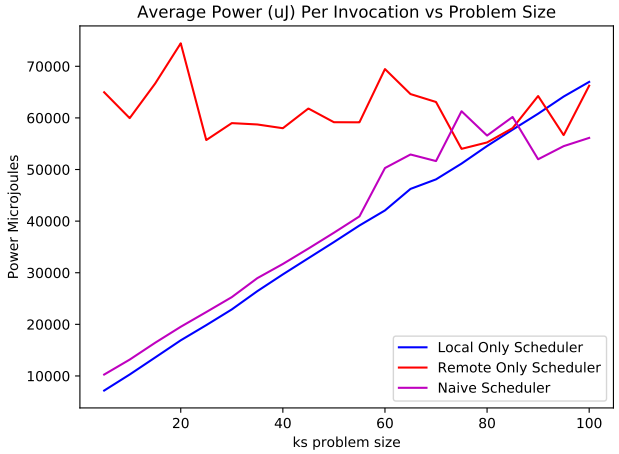

Fig. 7. Comparison of average invocation latency for local invocation strategy, remote invocation strategy, and offloading scheduler invocation strategy.

a substantial overhead. Second, the scheduler has difficulty switching to remote invocations when optimal, as it is subject to temporary network spikes.

E. An Improved Scheduler for Predictive Maintenance

Learning from the performance of our naive scheduler, we next construct a model adjusts to short-lived network spikes and that reduces the amount of time spent retesting. To this end, we devise a new scoring function that incorporates an exponential term to adjust, dynamically, the re-testing interval. In addition, we use the median latency of recent requests as a way of filtering outlier measurements.

We define the new score function to be

$$score = \text{median}(H) - (1 + r)^d$$

Where H is the invocation latency history for the strategy (local or remote) being scored, r is the decay rate which is set to $r = 1.2$, and d is the decay term – the number of

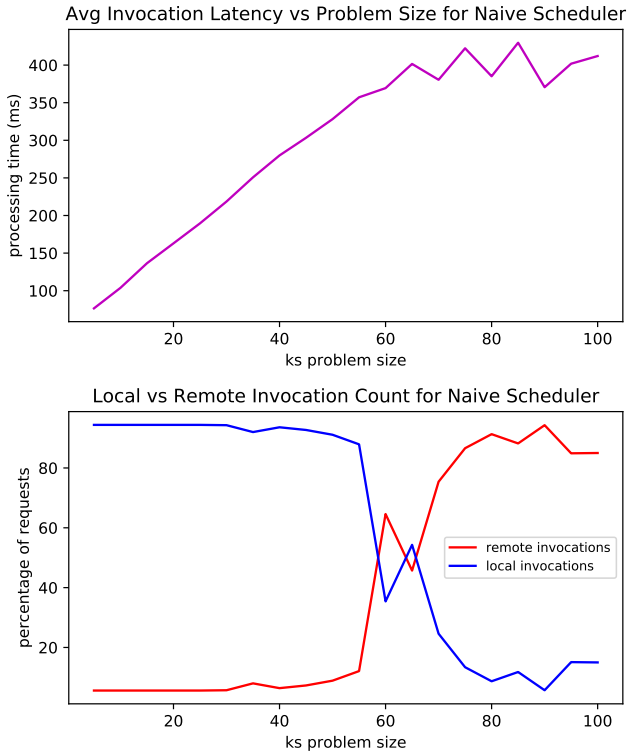

Fig. 8. Top: average invocation latency as a function of KS problem size for the offloading scheduler. Bottom: local invocation count vs remote invocation count selected by the offloading scheduler for each problem size.

invocations since this strategy was last selected. We take the median over a window of 3 samples which serves to filter out any one unusually fast or slow result allowing us to more easily recover from lost packets or other short-lived network spikes.

In Figure 9 we can see how this invocation strategy breaks down in terms of request distribution. When compared with the naive scheduler's strategy breakdown in Figure 8, we observe that the improved scheduler begins to switch scheduling strategies slightly earlier than the naive scheduler. It is interesting to see that the improved scheduler, with its noise filtering, transitions smoothly from local to remote invocations as the problem size gradually increases. When problem sizes are in the range from approximately 50 to 65, the scheduler mixes local and remote invocations as a result of variations in network latency, etc.

With the improved scheduler, the transition to remote execution is much smoother and completes much earlier than the switch-over seen in the naive strategy. The naive strategy fails to fully commit to remote execution until a problem size of 75 or 80, whereas the improved scheduler has mostly switched over by the time the problem size reaches 65. Even when the latency difference between remote execution is large, as is the case with problem sizes greater than 80, the naive scheduler appears to occasionally switch back to local execution. This is most likely a result of network timeouts which go unfiltered in

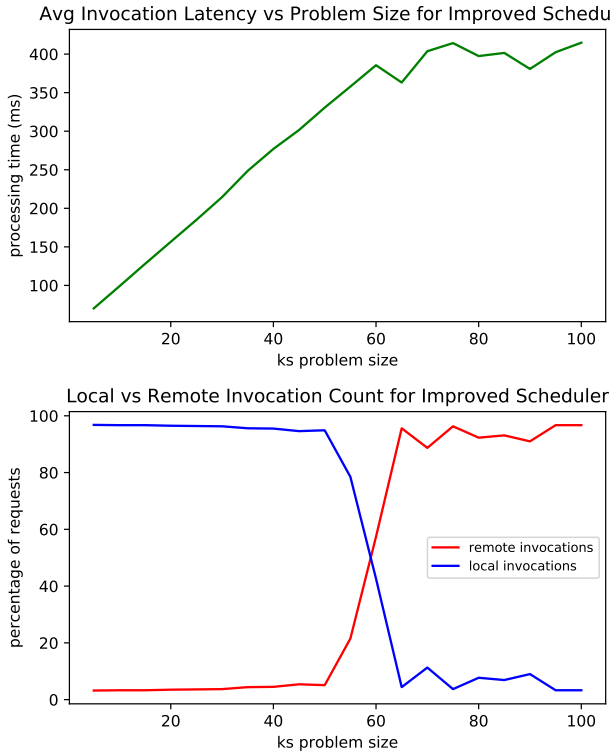

Fig. 9. Top: average invocation latency as a function of KS problem size for the improved scheduler. Bottom: local invocation count vs remote invocation count selected by the scheduler for each problem size.

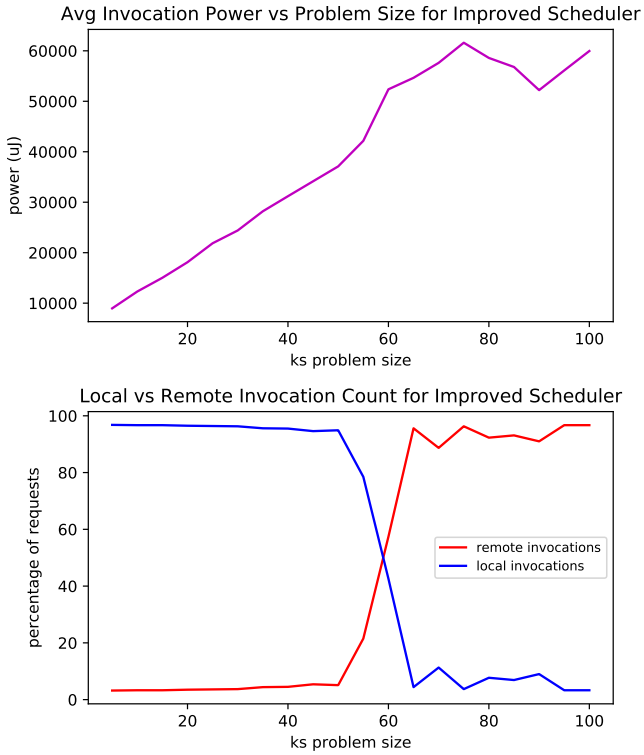

Fig. 10. Top: average invocation power draw as a function of KS problem size for the improved scheduler. Bottom: local invocation count vs remote invocation count selected by the scheduler for each problem size.

the naive model causing an unnecessary switch in execution mode compounding the performance penalty of the timeout.

In Figure 10 we show how the improved invocation strategy breaks down in terms of power consumption. We can see that, for the most part, optimizing for latency also optimizes power consumption. However, this rule appears to not always hold true. It is also interesting to see that, around problem size 50-60, where the scheduler begins to select remote execution, power consumption increases non-linearly before reaching resting power levels consistent with what we would expect from the results in Figure 7 for remote execution power draw. This non-linearity could likely be smoothed out with a model that has a deeper understanding of the relationship between latency and power draw for the two execution modes (local and remote).

In Figure 11, we show how this scheduling strategy compares to the strategies we have discussed so far. We consider naive, remote, and local scheduling.

In the local phase of execution (problem size less than 60) the advanced scheduler performs with the least overhead relative to local-only execution, though the difference is marginal. This is likely due to a slightly lower rate of re-testing with the exponential back off, which is supported by the the gap narrowing as the problem size (and thus the difference in latency) decreases, and so the retesting frequency increases for the advanced scheduler. During the remote phase of execution,

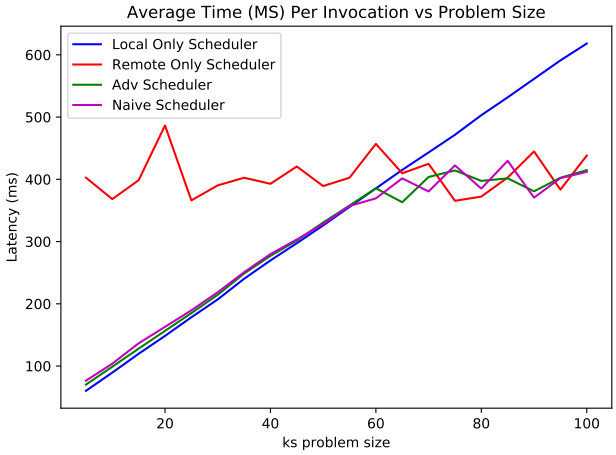

Fig. 11. In this chart is a latency comparison for all four schedulers (local, remote, naive, and adv)

the advanced scheduler's latency is generally less volatile and slightly lower than the naive scheduler. Both schedulers are, however, more consistent than the remote-only scheduler.

Figure 12 shows the average power use for each scheduling strategy. In this graph, the difference between the advanced scheduler and the naive scheduler is most pronounced during the local phase of execution – this is likely due to the higher power penalty for activating the network. Around a KS

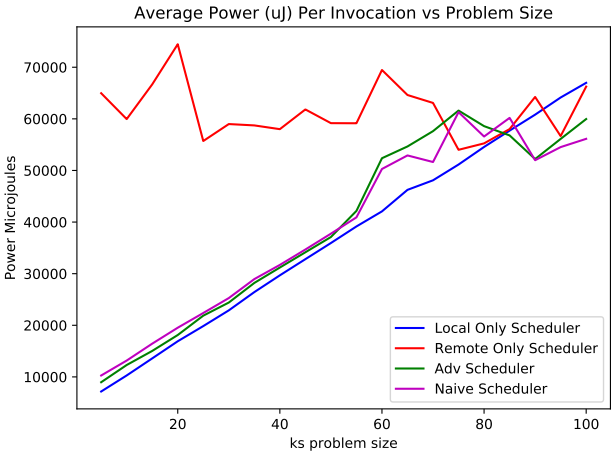

Fig. 12. In this chart is power comparison for all four schedulers discussed in this paper (local, remote, naive, and adv)

problem size of 50, we see a pronounced spike in power draw for both strategies, as was discussed in Figure 10. Interestingly, during this phase, the advanced scheduler appears to draw substantially more power – although it appears to minimize latency in Figure 11.

These graphs show that leveraging *NanoLambda*'s portability to schedule execution of Python handler functions enables optimization of latency and power use to achieve better performance than either remote-only or local-only execution. Additionally, these performance gains can be achieved with minimal overhead and no change to the handler implementation.

V. Limitations

Our results show that *NanoLambda* is capable of running Python AWS Lambda functions, with demanding processing loads, on-device as well as intelligently scheduling the execution of these function between the device and the edge or cloud. Despite providing a significant step toward raising the abstraction for and simplifying the programming of multi-scale, IoT devices end-to-end, multiple limitations remain that we hope to address in future work.

Currently, *NanoLambda* provides AWS Lambda compatibility through our AWS Lambda compatible code package format and *NanoLambda Cloud/Edge* service which emulates AWS's APIs for AWS Lambda and AWS S3. This feature simplifies code deployment and provides functions (running in the cloud, at the edge, or on a sensor) with persistent object storage capabilities. Handlers running on *NanoLambda Cloud/Edge* can directly access AWS Lambda and S3 or our emulated services running on top of *CSPOT*. Unfortunately, however, functions running on-device are limited to using wrapped interfaces that communicate only with our *NanoLambda Cloud/Edge* emulated services. Our goal is to extend *NanoLambda Cloud/Edge* with support for relaying requests directly to AWS for services that lack emulation.

Another limitation of *NanoLambda* is its support for Python packages. At the moment IoTPy's approach to security leveraging high-level Python VM's for isolation means that only Python bytecode can be delivered from the cloud for execution on device. As such, functions that depend on Python 3 C modules must reimplement the functionality of those libraries in pure Python or extend IoTPy with trusted C code running without memory isolation. One area of future work for possibly broadening the Python library base for *NanoLambda* may be cross compiling Python modules to web assembly for isolated execution on-device.

Finally, *NanoLambda*'s execution scheduler is simple and does not account for device capability, energy use, battery life or other factors in its function placement decisions. Similarly on the security front, *NanoLambda* does not authenticate edge devices or provide authorization or access control when translating or executing functions. Therefore, any device or client may acquire every available function in the server. Unfortunately, the mainstream access control mechanisms that employ public-key cryptography primitives are too resource intensive for most resource-constrained devices. To bring end-to-end security for functions deployed on device, we plan to leverage a promising new capability-based access control mechanism for IoT that uses HMACs [14].

VI. Conclusions

In this paper we introduce *NanoLambda*, a new Python runtime system and tool chain designed for the least-capable devices in IoT deployments. The use of Python makes function execution on heterogeneous devices portable and raises the level of abstraction to facilitate programmer productivity. To build this functionality, we adopt and mirror existing APIs as much as possible and thereby simplify the process of porting applications to it. To this end, we present *NanoLambda* to leverage an S3-compatible REST API for its code and configuration storage, as well as an AWS Lambda-compatible function package format, thus enabling many existing FaaS applications to run on *NanoLambda* with little modification. We detail the architecture of these services as well as our approach to on-device Python execution with *IoTPy*.

IoTPy is a new Python interpreter designed for micro-controllers, which defines its own bytecode language and code packaging format. *NanoLambda* offloads translation of *IoTPy* binaries to a *NanoLambda Cloud/Edge* service. *IoTPy* implements FaaS- and device-aware optimizations that efficiently implement *CSPOT* capabilities while staying within the memory limitations of highly resource constrained IoT devices.

We evaluate *NanoLambda* with a range of example applications for realistic use cases on the basis of invocation latency (important to control loop applications) as well as power draw. As part of our evaluation, we present an approach to scheduled edge cloud execution offloading as a way of winning back performance by leveraging our platform's ability to run the same code on-device and at the edge. We find that scheduled

execution allows our platform to outperform either remote-only execution or on-device-only execution across a range of problem sizes by rescheduling execution in response to a latency model for each execution strategy.

VII. Acknowledgments

This work has been supported in part by NSF (CNS-1703560, CCF-1539586, ACI-1541215), ONR NEEC (N00174-16-C-0020), and the AWS Cloud Credits for Research program. We thank the reviewers and our shepherds for their valuable feedback and suggestions.

References

[1] 2018 serverless community survey: huge growth in serverless usage. https://www.serverless.com/blog/2018-serverless-community-survey-huge-growth-usage/ [Online; accessed on 24-June-2020].

[2] T. Alladi, V. Chamola, B. Sikdar, and K. R. Choo. Consumer iot: Security vulnerability case studies and solutions. *IEEE Consumer Electronics Magazine*, 9(2):17–25, 2020.

[3] Amazon DynamoDB. https://aws.amazon.com/dynamodb/. [Online; accessed 15-Nov-2016].

[4] Amazon s3. "www.aws.amazon.com/s3" [Online; accessed on 24-June-2020].

[5] Amazon s3 rest api. "https://docs.aws.amazon.com/AmazonS3/latest/API/Type_API_Reference.html" [Online; accessed on 24-June-2020].

[6] Apache OpenWhisk. https://openwhisk.apache.org/. [Online; accessed 24-Jun-2020].

[7] A. Aske and X. Zhao. Supporting multi-provider serverless computing on the edge. In *International Conference on Parallel Processing*, 2018.

[8] AWS IoT Core. "https://aws.amazon.com/iot-core/" [Online; accessed 12-Sep-2017].

[9] AWS Lambda. https://aws.amazon.com/lambda/. [Online; accessed 24-Jun-2020].

[10] AWS Lambda IoT Reference Architecture. http://docs.aws.amazon.com/lambda/latest/dg/lambda-introduction.html [Online; accessed 12-Sep-2017].

[11] Azure Functions. https://azure.microsoft.com/en-us/services/functions/. [Online; accessed 24-Jun-2020].

[12] Azure IoT Edge. https://azure.microsoft.com/en-us/services/iot-edge/. [Online; accessed 22-Aug-2018].

[13] Azure IoT Hub. https://azure.microsoft.com/en-us/services/iot-hub/ [Online; accessed 22-Aug-2018].

[14] F. Bakir, R. Wolski, C. Krintz, and G. Sankar Ramachandran. Devices-as-services: Rethinking scalable service architectures for the internet of things. In *USENIX Workshop on Hot Topics in Edge Computing (HotEdge 19)*, July 2019.

[15] A. Banks and R. Gupta. Mqtt v3.1.1 protocol specification, 2014.

[16] Nicole Berdy. How to use Azure Functions with IoT Hub message routing, 2017. "https://azure.microsoft.com/en-us/blog/how-to-use-azure-functions-with-iot-hub-message-routing/".

[17] Creating a rule with a aws lambda action. https://docs.aws.amazon.com/iot/latest/developerguide/iot-lambda-rule.html [Online; accessed on 24-June-2020].

[18] Flatbuffers. https://google.github.io/flatbuffers/ [Online; accessed on 24-June-2020].

[19] D. Floyer. The Vital Role of Edge Computing inthe Internet of Things. "http://wikibon.com/the-vital-role-of-edge-computing-in-the-internet-of-things/" [Online; accessed 22-Aug-2016].

[20] Fog Data Services - Cisco. http://www.cisco.com/c/en/us/products/cloud-systems-management/fog-data-services/index.html. [Online; accessed 22-Aug-2016].

[21] Function as a Service. https://en.wikipedia.org/wiki/Function_as_a_Service [Online; accessed 12-Sep-2017].

[22] Google Cloud Functions. https://cloud.google.com/functions/docs/. [Online; accessed 24-Jun-2020].

[23] Google IoT Core. https://cloud.google.com/iot-core/. [Online; accessed 12-Sep-2019].

[24] GreenGrass and IoT Core - Amazon Web Services. https://aws.amazon.com/iot-core,greengrass/. [Online; accessed 2-Mar-2019].

[25] Internet of Things - Amazon Web Services. https://aws.amazon.com/iot/. [Online; accessed 22-Aug-2016].

[26] E. Jonas, J. Schleier-Smith, V. Sreekanti, C. Tsai, A. Khandelwal, Q. Pu, V. Shankar, J. Carreira, K. Krauth, N. Yadwadkar, J. Gonzalez, R. Popa, I. Stoica, and D. Patterson. Cloud Programming Simplified: A Berkeley View on Serverless Computing, Feb 2019.

[27] kstest.py. https://gist.github.com/devries/11405101 [Online; accessed on 24-June-2020].

[28] W-T. Lin, F. Bakir, C. Krintz, R. Wolski, and M. Mock. Data repair for Distributed, Event-based IoT Applications. In *ACM International Conference On Distributed and Event-Based Systems*, 2019.

[29] Z. Ling, J. Luo, Y. Xu, C. Gao, K. Wu, and X. Fu. Security vulnerabilities of internet of things: A case study of the smart plug system. *IEEE Internet of Things Journal*, 4(6):1899–1909, 2017.

[30] Micropython. https://microPython.org [Online; accessed on 24-June-2020].

[31] 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*, 21:64–71, 01 2017.

[32] Python 3 versus C gcc fastest programs. https://benchmarksgame-team.pages.debian.net/benchmarksgame/fastest/Python3-gcc.html [Online; accessed 24-June-2020].

[33] M. Satyanarayanan, P. Bahl, R. Caceres, and N. Davies. The Case for VM-based Cloudlets in Mobile Computing. *IEEE Pervasive Computing*, 8(4), 2009.

[34] Serverless Platform. [Online; accessed 10-Feb-2019] www.serverless.com.

[35] T. Verbelen, P. Simoens, F. De Turck, and B. Dhoedt. Cloudlets: bringing the cloud to the mobile user. In *ACM Workshop on Mobile Cloud Computing and Services*. ACM, 2012.

[36] R. Wolski, C. Krintz, F. Bakir, G. George, and W-T. Lin. CSPOT: Portable, Multi-Scale Functions-as-a-Service for IoT. In *ACM/IEEE Symposium on Edge Computing*, 2019.