Abstract

Increasingly, the heterogeneity of devices and software that comprise the Internet of Things (IoT) is impeding innovation. IoT deployments amalgamate compute, storage, networking capabilities provisioned at multiple resource scales, from low-cost, resource constrained microcontollers to resource rich public cloud servers. To support these different resource scales and capabilities, the operating systems (OSs) that manage them have also diverged significantly. Because the OS is the "API" for the hardware, this proliferation is causing a lack of portability across devices and systems, complicating development, deployment, management, and optimization of IoT applications.

To address these impediments, we investigate a new, "clean slate" OS design and implementation that hides this heterogeneity via a new set of abstractions specifically for supporting microservices as a universal application programming model in IoT contexts. The operating system, called *Ambience*, supports IoT applications structured as microservices and facilitates their portability, isolation, and deployment time optimization. We discuss the design and implementation of Ambience, evaluate its performance, and demonstrate its portability using both microbenchmarks and end-to-end IoT deployments. Our results shows that Ambience can scale down to 64MHz microcontrollers and up to modern x86_64 servers, while providing similar or better performance than comparable commodity operating systems on the same range of hardware platforms.

Ambience: A Portable and Efficient Operating System for Microservices
UCSB Technical Report Number 2023-01

Fatih Bakir
University of California, Santa Barbara and Apple Computer
mfatihbakir@gmail.com
Sierra Wang
University of California, Santa Barbara and Stanford University
sierraw@stanford.edu
Tyler Ekaireb
University of California, Santa Barbara
tylerekaireb@ucsb.edu
Jack Pearson
University of California, Santa Barbara
jackpearson@ucsb.edu
Chandra Krintz
University of California, Santa Barbara
ckrintz@cs.ucsb.edu
and
Rich Wolski
University of California, Santa Barbara
rich@cs.ucsb.edu

March 24, 2023

1 Introduction

Today, applications and systems that amalgamate heterogeneous, resource-restricted, or embedded devices with traditional resource-rich compute resources (e.g. cloud-hosted virtual servers) cannot use a single, "universal" set of abstractions to execute on all hardware components. Specifically, in an "Internet of Things" (IoT) context, resource-constrained, small scale devices are programmed using special-purpose or embedded technologies [9, 33, 112,

51, 97] that then must interoperate with services programmed using popular and productivity-enhancing cloud technologies [69, 73, 98]. Embedded development often sacrifices the convenience and productivity enhancement accruing to cloud development in favor of the ability to optimize comprehensively throughout the software stack. Cloud technologies are too abstract to support low-level system optimizations, and low-level and often bespoke device programming technologies are too granular to support productive and sustainable cloud applications. For IoT, this bifurcation of the system software between high-level software stacks that enable rapid development of scalable cloud services and highly-optimizable "bare bones" operating systems targeting resource-constrained devices, creates reliability, maintainability, security and scalability challenges.

Microservices are a popular architecture for building scalable, distributed network services and applications [84]. The microservice architecture has seen wide adoption, with numerous supporting infrastructure projects [3, 71, 87, 61]. Applications structured as microservices are composed of many small and "simple" services (to promote code reuse and cohesion). Moreover, separately developed services can interoperate successfully despite their internal use of widely varying software technologies when they interact via well-defined, message-based interfaces. For these reasons, microservices are typically hosted within separate isolation domains to improve fault isolation and/or implement multi-layered trust and security policies. For uniformity, service requests and responses between microservices are commonly implemented using typed, Remote Procedure Call (RPC) interfaces [50, 8, 27] and web-service frameworks or middleware [75, 90, 25].

Furthermore, since microservice design promotes the proliferation of many different services in a single application, users and administrators of these applications often employ container orchestration technologies to implement and maintain application deployments [71, 41, 29]. Using these frameworks, developers describe the end-state of a service-mesh deployment using a declarative language, and the framework instantiates and maintains it by creating new instances and decommissioning stale ones [22]. Thus, while the microservice architecture depends upon the integration of heterogeneous software stacks, it also typically requires an additional runtime framework for orchestrating isolation containers [35, 106, 28, 93, 31, 109] as well.

Both the microservice software stacks associated with each individual service, and the container management systems for orchestrating them, depend on general purpose operating systems which are typically a Linux or Windows variant. This dependence poses two challenges that are becoming

increasingly difficult to overcome with respect to managing the proliferation of technology heterogeneity in IoT settings. The first is that the plethora of hardware platforms (e.g. embedded IoT devices, microcontrollers, special purpose processors, edge computers, security co-processors, etc.) do not support a common set of operating system abstractions, let alone a common operating system, either among themselves or in common with commodity servers. That is, while most commodity servers and virtualization environments support some form of Linux or Windows, neither of these general purpose operating systems can be supported on *all* devices in a distributed deployment that includes special purpose or embedded systems.

Cloud vendors have attempted to address part of this challenge by providing "serverless" computing support for IoT applications. Serverless, or FaaS (Functions as a Service) microservices [13, 49, 32, 87] provide programming environments which permit developers to write simple-event driven service "handlers" that are then uploaded to a runtime service responsible for deploying them automatically, dispatching service requests to them, and scaling them up and down in response to offered service request load. FaaS functionality was originally developed to support automatic scaling cloud-hosted web services as a way of reducing hosting costs. For IoT, many of the large cloud vendors have extended this FaaS functionality to include service deployment "at the edge" – on a machine not part of the cloud, but reachable from it via a network. AWS IoT Greengrass [12] for example, allows developers to deploy transparently their AWS Lambda FaaS functions to a Raspberry Pi or x86 single-board computer running outside of the cloud. However, cloud providers have yet to extend the FaaS model to microcontrollers, possibly due to efficiency and security challenges associated with doing so. Thus, even with FaaS technologies that require no direct operating system interactions, the state-of-the-art is that the edge and the cloud can be programmed with a uniform "Function-as-a-Service" model, but microcontrollers must be programmed using different technologies (e.g., MQTT [82], FreeRTOS [43, 51], and IoT SDKs [14]). As a result, applications, even when adopting microservices in this context must correctly compose an increasingly vast array of disparate protocols and separately-developed technologies to achieve functionality.

The second challenge is that the cloud model of performance scaling does not translate feasibly to an IoT context. While "scale out" – the addition of separate vitualized hosts to a cloud-hosted web service (e.g. via a FaaS platform in response to increasing request load) – has proved economical and effective in a cloud context [45, 81], it is less effective or infeasible for deployments that include heterogeneous collections of low-resource

or resource-restricted devices and processors. When scale-out is infeasible, the alternative is to "scale up" by migrating or co-hosting services within larger, more resource-rich machines. For IoT, where geographic and siting limitations make the resource deployment topology heterogeneous, it is often not possible to find or site a more resource-rich machine to effect scale up. Further, even in relatively homogeneous cloud-hosted deployments, as our results presented herein indicate, the generality of commodity operating systems introduces a per-node performance penalty when executing microservices, thereby limiting the effectiveness of scaling up.

Our thesis is that for IoT applications to take advantage of cloud, edge and device infrastructure and technologies, they require a new and unifying software environment based on a common set of efficient abstractions that can be implemented at all resource scales. Further, to take maximal advantage of the technological success accruing to cloud computing, these applications are best structured as microservices. We describe *Ambience – a new operating system specifically designed to support IoT applications structured as microservices in heterogeneous distributed settings that include device and resource capabilities spanning a range of resource scales* [1]. Ambience is not a general purpose operating system. Its single set of common abstractions specifically supporting microservices across all resource scales, providing event driven, scalable, efficient systems at cloud and edge scales, while, at the same time satisfying the resource restrictions and requirements of embedded IoT devices. Ambience is also not an amalgamation of pre-existing technologies originally developed separately for cloud deployment and embedded systems (respectively) and then adapted to support IoT. It posits that the abstractions microservices require can be implemented as "native" operating system abstractions and by doing so, can be made space and performance efficient enough to be effective at all resource scales.

Thus, our research with Ambience postulates that it is possible to design an operating system which is both efficient enough and high-level enough to support microservices as a universal programming and deployment model. It does so by defining optimizeable, high level collection of abstractions that includes isolation groups, coroutine-based asynchrony, typed interfaces and deployment specification, among others. These abstractions expose more information that is specific to microservice implementation and deployment than their general purpose operating system alternatives.

Ambience makes use of these abstractions not only to ease programming

[1] Ambience is available as open source from `https://github.com/MAYHEM-Lab/ambience`

across heterogeneous systems, but also to introspect and automatically specialize the microservices it hosts. Rather than a single kernel image shared across all nodes of a deployment, Ambience generates individual kernel images, each specialized and optimized to run the microservices that are to be hosted on a specific target node in a deployment. That is, Ambience includes deployment information in the form of a "deployment manifest" that it uses to generate optimized and customized operating system images for each device or server targeted in a deployment.

We show that these optimization features allow Ambience to achieve throughputs on the order of hundreds of thousands of requests per second across isolation domains on a single x86 core. Furthermore, the same microservices can be transparently deployed on microcontrollers and single board computers, x86 hypervisors (KVM [68], Firecracker [4] and VirtualBox [108]) with virtio [95] support, and embedded within Linux systems (to facilitate incremental transition to Ambience), without modification.

Because Ambience is a complete system we, refine its exposition to an enumeration of the design choices and features that differentiate it from other research and commodity operating systems. These differentiating characteristics include the following.

- *Deployment-time determination of isolation boundaries* – Ambience delays the decision of how to implement isolation between microservices until the services are deployed. In particular, microservices can be conjoined within the same isolation domain without recoding while avoiding unnecessary messaging overhead (cf. Subsection 3.2).

- *Asynchronous Computational Model* – The default computational model for Ambience is stackless coroutines [62], although fibers [66] are also supported. This choice (described in Subsection 3.5) combined with single, queue-based Application Binary Interface (or ABI – described in Subsection 3.4) make Ambience space and time efficient enough to comprise all resource scales in an IoT deployment.

- *Typed System Calls and Compile-time Optimization* – Ambience requires that applications make requests for operating service using typed interfaces. It uses this information both to ensure system integrity and to perform compile-time optimizations (cf. Subsection 3.7). In this way, Ambience can comingle application code and operating system code into a single, optimized system image (cf. Section 4).

- *Automatic Network Overlay Generation* – Because Ambience generates a set of system images for a single deployment of a microservice

mesh, it can also automatically generate application-level message forwarding services (cf. Subsection 3.12) and (using the mechanisms describe in Subsection 3.3) include these services in the kernel of each system image.

- *Capability-based Access Control* – To implement end-to-end access control across a deployed service mesh, Ambience uses an enhanced version of CAPlets [16] – an efficient capability system designed for multiscale distributed systems (cf. Subsection 3.13).

In addition to these design features, Ambience includes a number of automated code synthesis capabilities (e.g. it autogenerates and inserts RPC code from cross-node communication) and optimization techniques (e.g. it makes heavy use of zero-copy communication whenever possible). While many of these features are inspired by features implemented in previous systems (cf. Section 2), Ambience uniquely aggregates them into a single operating system to create a common set of abstractions that can be implemented across resource scales in an IoT deployment. This unification, combined with secure network transparency, make novel research contributions in the operating systems, distributed systems, and IoT research domains.

We demonstrate Ambience's flexibility and portability empirically using a distributed IoT application that implements wildlife tracking in remote geographic areas using a combination of remotely sited devices, edge single-board computers, and the cloud. We evaluate its performance through detailed microbenchmarks. We also compare Ambience's key characteristics qualitatively both to Linux and and to Azure's IoT platform [83] with respect to IoT application development and deployment. In the sections that follow, we contextualize these contributions in terms of previous and related work and through an exposition of the Ambience abstractions, automated optimizations, and deployment support.

2 Related Work

Microservice frameworks are typically designed to use Linux containers to provide both isolation between conflicting software dependencies that individual service stacks may have and also runtime isolation for security purposes [65, 89, 6, 99]. The proliferation of container images, runtime configurations, and operational lifecycles among separately developed microservices (often within the same application) created the need for runtime and orchestration technologies that automate provisioning, scheduling, and

deployment of microservices [3, 71, 72, 29]. Kubernetes [71] has received wide-spread adoption from users and service providers alike. Kubernetes requires developers to specify their entire deployment in declarative files instead of manual provisioning. This makes the creation and migration of entire multi-node clusters a trivial operation. Ambience integrates these mechanisms at the operating system level and leverages a similar declarative approach for deployment specification.

Serverless computing and Functions-as-a-Service (FaaS) constitute an alterative to deploying and managing microservices using cloud platforms [58, 18, 56, 87, 13]. FaaS platforms are cloud-hosted service venues that accept service request handlers and trigger them when specific requests are forwaded to them via either a network facing request dispatcher or some other cloud-based service. Because users of FaaS platforms only provide handler code (and do not provision servers or other resources necessary to dispatch and execute the hander code), the term FaaS is often synonymous with the terms "serverless" or "serverless computing." These FaaS or serverless systems provide high availability, fault tolerance, dynamic elasticity via automated, event-driven provisioning, containerized execution, and management of the underlying infrastructure. Philosophically, Ambience shares the view of microservices (implemented using FaaS) as an "omniplatform" for IoT with [110] but it goes on to illustrate that miniaturizable FaaS functionality, by itself, is not sufficiently performant in terms of memory footprint and execution efficiency. Also, the authors of [110] specify no model for device I/O – a key feature in an IoT development context. The authors of [61] exploit locality across serverless microservices to replace RPC with IPC primitives to increase throughput and lower latency. The authors conclude that there remain many individual overheads. By co-designing the entire stack for deployment and performance, Ambience eliminates a significant number of these overheads.

Ambience integrates abstractions (lightweight isolation, asynchronous interfaces between trust domains, queues, groups, etc.) and tooling (deployment IDL, compilation support, deployment/code specialization) that are also found in other systems [71, 41, 57, 104]. The authors of [100, 101] introduce the implementation of asynchronous system calls in Linux by designating pages of memory as a buffer that is polled by kernel threads. However, to achieve adequate concurrency and performance, a large number of kernel threads are required, which causes memory pressure. The io_uring [11] effort is a recent approach to implementing an alternative asynchronous (async) system call for Linux [26]. However, at the time of this writing, it does not support all system calls, and does not support kernel-to-user requests. A

similar queue design is used in virtio's [95] interface where a guest operating system communicates with the host through *virtqueues*, similar to the earlier Xen [19]. While they too do not support host-to-guest requests, unlike io_uring, they use a unified pool of queue elements, so the guest can issue more work with the same amount of memory without VM-exits. Unlike these approaches, Ambience supports bidirectional asynchronous communication with low kernel resource consumption over its queue interface.

Kernel bypass systems [20, 91, 42, 60] try to eliminate kernel overheads related to network processing and context switching. Ambience, alternatively, attempts to eliminate these overheads by specializing the kernels it generates to support the user space microservices they host.

In [54, 74], the authors explore the use of memory protection units on microcontrollers to improve reliability and enable the execution of untrusted code. However, these approaches do not support server or edge class machines. Authors of [17, 110, 46] show that a lightweight serverless architecture implementation running in both Linux user space and on microcontroller systems, even without memory protection, is a viable architecture for building distributed IoT systems. Ambience is distinct from these efforts in that it is a comprehensive operating system approach that supports microservices running at all resource scales.

Unikernels [79, 24, 86] reduce operating system overheads by merging the kernel and the application, and by eliminating kernel protection. The motivation behind removing kernel protection is that because virtual machines implement isolation between applications, kernel corruption can only affect the application using it. However, their lack of IPC primitives prevents them from exploiting locality. Ambience supports multiple isolated services running in the same VM with efficient communication among them (including zero-copy IPC similar to that originally described in [94]). For deployment settings in which isolation is not desired, Ambience also supports transparent placement of services inside kernel space.

Using language typing to ensure operating system integrity is a feature of [21] and, more recently, [59], both of which use strong types to enforce isolation of user provided programs inside privileged domains. Through safe user code inside the kernel, such systems allow the dynamic introduction of efficient abstractions. However, for both systems, the type system is only available in special programming languages, and does not extend to untrusted programs written in arbitrary languages. Further, the type information is not used for performance optimizations, and mainly exists to statically enforce safety. Alternatively, [92] embeds a JIT (Just-In-Time) compiler in the kernel to automatically and dynamically create optimized code-

paths, facilitating specialization. However, performing this specialization dynamically precludes [92] from running on resource constrained devices. Ambience performs specialization using a variety of information available statically, making it possible to run fully optimized images across resource tiers.

3 Ambience

In this section, we overview our key design choices, their trade-offs, and our implementation approach for Ambience. Note that because Ambience is designed specifically for hosting microservices, it omits many abstractions found in general purpose operating systems. This specificity is not an impediment to generality, however, since most microservices do not make operating system calls directly. For example, an inspection of the Deathstar microservice benchmarks [44, 34] shows that the microservice code makes only the Linux `exit` and `signal` system call – all other system calls are implemented by the high-level web service frameworks within which the services operate.

3.1 Definitions & Abstractions

The primary abstraction of Ambience is a *service*. A service is a collection of procedures, each with a strongly typed interface, operating on a common, ephemeral state. The procedures act as entry points which can be concurrently executed. A *service interface* is a nominal abstract type consisting of procedure interfaces, defined in an interface definition language (IDL). Ambience includes its own IDL for generating service interfaces called *lidl*.

A *node* is an abstract entity that can host Ambience services. They can be physical machines (e.g. servers, single board computers or microcontrollers) or they can be virtual (e.g. cloud virtual machines, Linux processes, a webpage running webassembly [53], etc.). Ambience provides different levels of service on different host types since it does not have the same level of control in all physical and virtual devices. A *cluster* consists of a set of nodes and *network*s that connect those nodes.

All Ambience runtime abstractions are deployed via declarative *manifests*. Manifests direct the Ambience to "compile" images (one for each node in a deployment) that instantiate services (including their dependencies), describe network topologies, define security isolation groups, etc. Ambience manifests are written in a Domain Specific Language (DSL) embedded in Python. Ambience manifests encapsulate more information than existing

declarative approaches [71, 41]. Specifically, they include service interface types and dependencies, which Ambience uses to synthesize efficient code for communication, and security isolation. Listing 1 shows a deployment manifest excerpt.

Listing 1: A sample Ambience deployment manifest

```
%\begin{minted}{python}
# Services file
instance(name="detection",
         service=tflite_detection)
instance(name="camera",
    service=dcmi_camera,
    dependencies={"frame_handler": "detection"})
export(service="camera",
       networks={"udp-internet": 4898})
# Deployment file
group(name="camera_group",
      services=["detection", "camera"])
deploy(node="camera_microcontroller",
       groups=["camera_group"])
}
```

Ambience injects service dependencies using information within manifests during image construction, thereby precluding the need by each service to perform service discovery. That is, the service mesh topology associated with a specific deployment is "compiled into" the images that make up the topology. Ambience enforces type-safety in the manifests and synthesizes code that brings up all services in the correct order and passes dependencies to each service.

3.2 Service Groups

Microservice design advocates for the proliferation of small, simple, isolated services. In existing systems, the decision about whether to execute two or more services in the same isolation domain is often binary and irreversible – two services are either separate entities deployed in isolation, or they are part of the same service. The need to decide whether two services will be isolated or comingled poses an early design challenge in the service development engineering cycle. Developers must make design decisions about service isolation that are difficult and costly to reverse or change once development begins, and becomes more difficult to change as development

matures. Further, the performance of the resulting service mesh is not typically known until relatively late in the development lifecycle and, often, isolation design decisions must be revisited, and implementations recoded, to enhance performance.

Microservice design can also pose a deployment challenge in resource restricted settings. Each isolated runtime entity (e.g. a process) consumes system resources: page tables, thread structures, kernel entries, communication costs etc. Tying the allocation of these resources to each service reduces deployment flexibility and portability. For instance, deploying two related services in different address spaces may be desirable on a cloud server but not on a microcontroller, especially when the microcontroller logically constitutes a single protection domain (i.e. it has one owner or one user). Further, a developer may simply wish to improve performance when all services *can* run in the same trust domain, by removing the isolation boundaries.

Existing commodity operating systems do not support such flexibility directly: a program becomes a process when executed and a process is not meant to be occupied by multiple distinct, separately developed programs. Moreover, each process has global state associated with it (file descriptors, signal handlers, file system root, quotas, etc.) that are space-expensive to replicate in resource-restricted execution environments.

To overcome these challenges, Ambience eliminates all global state associated with a "process." Instead, it defines protected regions of address space that can be occupied by separate microservices. To enable this lighter-weight form of isolation we introduce *groups* as the unit of runtime execution and deployment.

Microservices assigned to the same group share address space and are not isolated from one another. Microservices assigned to separate groups, but hosted on the same node, are isolated and must communicate using fast Ambience local interprocess communication (IPC) as described in Section 3.3. Microservices executing on separate nodes must be in separate groups and communicate using RPC. Importantly, assignment of services to groups is *not* a design-time or development-time decision with Ambience, but rather a *deployment-time decision*. That is, the developer or operational manager (in a "DevOps" [36] context) can decide what assignment of services to groups is most appropriate for each deployment, based on site-specific trust policies, security policies, performance requirements, etc., without design or code modifications to the services or duplication.

When microservices are assigned to separate groups in a deployment, Ambience automatically incorporates IPC to facilitate communication between groups. It emits direct function calls to optimize communication

within a group. Note that it is not possible to make a similar decision of whether to include two service components in the same Linux process or different Linux processes at deployment time without having two separate versions of the code: one for conjoined deployment and the other for separate deployment.

Services within a group share runtime resources: the queues as explained in Section 3.3, an event loop and associated system threads, heap and page tables. By default, Ambience allocates a group per service. A developer is allowed to create explicit groups and include the services they wish to couple.

Note that under the Ambience group resource abstraction, services do not receive implicit resources and ambient privileges. For example, there is no global file system inherited by each group in Ambience: if a microservice requires file system access, the developer can explicitly assign a dedicated file system service to it or if two services are meant to share a file system, the developer can assign both of them to use a single file system service explicitly (either within the same group, separate groups, or in any combination.)

This flexibility is designed to support severely resource restricted devices as well as more resource-rich servers. For example, on microcontrollers with a few kB of memory, all services in a node can be placed in a single group, eliminating most of the Ambience runtime isolation memory footprint. Key to this approach, is that services need no changes when they are assigned to the same or different groups and the ability of Ambience automatically to insert appropriate communication primitive based on how the services are to be deployed.

3.3 User Space Design

Microservices (particularly those that employ a FaaS design structure) make use of event driven and asynchronous programming, whereas traditional systems mainly provide a synchronous programming environment, and the user space code is expected to implement asynchrony [78, 10] on top of synchronous abstractions provided by a kernel. Most kernels are themselves designed to implement these synchronous abstractions for user space programs using an event-driven and asynchronous model to interact with the hardware. For microservices, this translation from an asynchronous hardware interface, to a synchronous system call interface, and then back to an asynchronous model within the microservices themselves creates inefficiencies that Ambience attempts to avoid.

To do so, Ambience exposes the asynchronous abstractions used by the

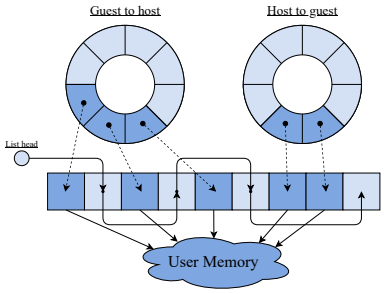

Figure 1: Isolated services communicate with the kernel via an asynchronous, bidirectional queue interface. The user space never performs serialization and merely sets up pointers. These rings form the exclusive interface to the kernel and other services, allowing Ambience to reconfigure deployment topologies without recompilation. For instance, a service binary can be placed inside the kernel space or a separate user space, since Ambience programs do not use system calls. The List head pointer is the head of the atomic free list.

kernel to microservice components running in user space. Ambience provides and manages an event loop at the kernel level for each user space. This event loop shares code with, and is almost identical to, the one used inside the kernel to handle hardware events. The kernel issues user-space procedure calls directly, instead of having the user space poll and route requests. This optimization reduces the workload on each service, and provides centralized, dynamic configuration parameters such as concurrency limits and a unified tracing and observability infrastructure.

Computations executing in separate groups use bidirectional asynchronous queues for communication with the kernel. Specifically, groups implement dedicated lock-free queues for both the kernel-to-user and user-to-kernel communication. Both queues index into a per-group, shared array of queue elements. The allocation of these elements is lock-free. Lock freedom here is necessary since multiple user or kernel threads may attempt to allocate an element concurrently. Unlike existing ring or queue based interfaces [95], Ambience allows both ends of the interface to make *and serve* requests.

3.4 Kernel "Styles"

As such, Ambience's kernel does not feature an ABI (Application Binary Interface) that includes a fixed set of system calls. Instead, its ABI is an interface that permits communication via the asynchronous ring data struc-

tures. Further, all core operating system functionality is exposed and deployed as services that are accessed just like user provided services, meaning such services can be individually omitted or deployed inside the kernel or a user space. Such flexibility is unique to Ambience and allows individual deployments (not service implementations) to be configured using different operating system kernel "styles" such as a unikernel [79], where every service is deployed inside the kernel group; a monolithic kernel [107, 5] where device services are deployed inside the kernel whereas user services are deployed in user space groups, a microkernel [48, 64] where supported device services as well as user services are deployed in user space groups, or an entirely new class, with no changes to the base system.

For example, an Ambience file system service exposes a file system interface and depends on a block device service. Since Ambience's ABI does not include system calls, the file system can be transparently deployed inside or outside the kernel, with different security, performance and reliability tradeoffs, and accessed via typed interface like any other Ambience microservice.

Because user-space microservices use the same ring interface to communicate with each other as they do to request service from the kernel, user-level microservices can be "moved" into the kernel transparently (say for performance reasons). This design feature also allows Ambience programs to be potentially portable to other operating systems, provided the ring interface and necessary core services are re-implemented. Ambience makes use of this feature for debugging Ambience services using gdb on Linux, albeit with reduced performance, since it does not yet have such a sophisticated native debugger.

3.5 Asynchronous Programming Model

One common design pattern for asynchronous runtime systems requires the programmer to use "callbacks." [78, 77, 85, 10]. Callbacks add program complexity and programmer burden as they require the creation of multiple related functions to implement a single request handler. In addition, in languages which lack garbage collection, the lifetimes of the shared variables among callbacks must be carefully managed by the programmer at the risk of memory corruption.

User-mode threads such as fibers [66] provide a compromise between callbacks and system threads. However, fibers have sub-optimal memory requirements: each fiber must allocate and retain enough stack memory for the worst case memory usage and/or call depth. Practically, the worst case stack size use is not statically known, and each fiber almost assuredly over

allocates its stack. This lack of memory efficiency can cause significant memory pressure in highly concurrent settings. For example, the authors of [62] report stackful coroutines are up to 93% slower than stackless coroutines on Windows.

Stackless coroutines provide the efficiency characteristics of callbacks while providing the benefits of synchronous programming abstractions. At any time, a coroutine retains only enough memory to store its working set of local variables. The disadvantage is they require compiler support to transform the coroutines into resumable functions. However, most programming languages now support coroutines. Indeed, 11 out of the 13 most popular programming languages support them [102], with the exceptions being C and Java.

Listing 2: Ambience's concurrency design allows the API for writing data to be decoupled from the model of asynchronous concurrency used by the microservice using the API. In this code, the write() creates a job that can be bound and completed by a coroutine, a thread, a fiber resumer, or a callback.

```
struct write_job {
    bytes data;
};
// Takes a bytes object and constructs an asynchronous job for writing t
write_job write(bytes);

bytes some_data = ...;
// Construct a job from some_data. Note that the job did not start yet,
// start once it is bound.
auto job = write(some_data);

co_await job; // Use coroutines
sync_wait(job); // Use threads
fiber_wait(job, this_fiber); // Use fibers
auto state = bind(job, [](auto res) {}); // Use callbacks
}
```

Due to their superior efficiencies, wide spread availability, and ease of programming, Ambience uses coroutines as its default computational model. It also supports fibers for compatibility with existing libraries expecting to be able to block in a deep call stack. Ambience's low level ring interface facilitates callback-style programming as well. We compare the performance

fibers and coroutines in Ambience in Subsection 5.5.

Implemented naively, support for multiple concurrency models together requires the the implementation of some functionality to be duplicated. To avoid this duplication, Ambience defines work units, called *jobs* that can then be bound to a particular completion handler associated with one of several different concurrency models. That is, the completion handler can be a callback, coroutine, fiber resumer, or a thread. This decoupling of the computational work specification from the concurrency model allows a single asynchronous API (for example, a network packet transmission API), to be used with different concurrency models employed by individual microservices. Listing 2 shows the developer facing API for writing a byte object, and Listing 3 shows the implementation of `sync_wait` which is used to complete the `write` job using a thread stack. Note that only `sync_wait` is coded to use threading, and the implementation of `write` can be used with any of the other concurrency models efficiently. Switching to this decoupled design from the naive version reduced Ambience's driver sizes by a factor of 3 in the current version.

Listing 3: The Ambience code for implementing job completion as a thread.

```
template <AsyncJob JobType>
auto sync_wait(JobType&& job) {
    // Allocate stack space to store the async result.
    late_initialized<result_type<JobType>> result;
    // sync_wait uses a semaphore to block the current thread
    semaphore sem{0};
    // bind launches an async job with a callback
    // bind returns an object that must remain alive until
    // the callback is invoked, which is trivially done in sync_wait
    auto state = bind(job, [&](auto&& res) {
        result.emplace(std::move(res));
        sem.up();
    });
    // Block until the semaphore is signaled by the callback
    // As we are blocking here until the callback is invoked, the semaph
    // and the temporary storage for the result and bind state will reme
    // valid as long as the async job is alive.
    sem.down();
    return std::move(result).get();
}
```

3.6 Immutability and Specialization

Microservice deployment orchestration frameworks [71, 41] often employ immutable and declarative languages to describe a deployment (i.e. how services are mapped to machines and their network interconnection topology). These frameworks use this specification to install, start, and maintain services across the nodes in a deployment. In practice, microservice orchestration frameworks deploy services in Linux containers, where each container is assigned to a node in a deployment. When a change is made to a service, the framework stops the container or containers running the service and starts one or more replacements with the updated image. Thus, the containers are immutable.

Ambience also uses a declarative language model to describe each deployment, but in a different way. Instead of instantiating a deployment strictly at runtime (as most container orchestration frameworks do) it builds a potentially unique kernel image for each node in a deployment, optimized for the microservice workload the node will run. Specifically, it carries relevant type and deployment information specified in an IDL and manifest files that are used by an image builder to create optimized images for each node in a deployment.

Ambience also can, opportunistically, preallocate some resources for services at build time, to reduce cold start times and to detect insufficient resources ahead of time. Currently, Ambience can preallocate the following for each group: isolation structures, system thread stacks and control blocks, queues, in-kernel group descriptors. It also preallocates networking structures, for instance UDP control blocks, for services that communicate across nodes. When possible, these resources are initialized at compile time using `constexpr` [30] data structures and algorithms. `constexpr` allows some stateless C++ code to execute at compile time. Ambience also supports dynamic provisioning of these resources at runtime. For example, pages can still be allocated and mapped dynamically, threads can be created and destroyed, and sockets can be created at runtime, only with higher runtime cost and the possibility of runtime failure.

3.7 Use of User-defined Types in the Kernel

Microservices in Ambience communicate over statically typed interfaces defined in an IDL which also conveys type information to the kernel. Further, because the microservice code and the kernel are "compiled" together when each node image is constructed, this information is used to optimize the

user-space kernel interactions. Ambience also uses this type information to auto-generate any serialization code that is needed to facilitate message-based communication (e.g. when services deployed to separate nodes communicate).

Note that by contrast, commodity general purpose operating systems typically implement "typeless" system call interfaces. That is, when a `write` or `read` system call is invoked, any data passed to or from them is an untyped collection of bytes. Communicating clients and servers recover the type information via serialization and deserialization. For example, a client-server application using a modern IDL such as gRPC [50] contains type information which is used by the user-space code implementing the microservice interaction for correctness, and as a programming aid. However, once a request or a response needs to be sent between the client and the server, the microservice stack will eventually make a call to the POSIX `write` system call, or a socket `send` call on the sending side and, conversely, a `read` or `recv` system call on the receiving side. These system calls only view the information as untyped collections of bytes.

In contrast, Ambience maintains interface type information for as long as possible. Service interfaces are typed at deployment time and this type information is available and used when the kernel is constructed. Type erasure is only performed when an Ambience invocation crosses a network boundary.

Ambience makes extensive use of this information to implement efficient communication, enable compiler optimizations, gain observability, and to introduce additional functionality. Ambience queues are strongly typed: when a user space microservice component makes a request, it does not perform serialization. Instead, it packs pointers to its arguments in a typed data structure designed to facilitate a zero-copy transfer that is generated by the IDL. If the request is to a service running in the the local kernel (recall that Ambience can support in-kernel microservices as described in Subsection 3.3) it is handled via this zero-copy mechanism. If the request is to a component running in another Ambience group on the node, Ambience synthesizes specialized code using the static type information to implement efficient parameter passing between the groups. If the request is to be handled off-node, Ambience performs serialization and communication. Critically, it is the Ambience image builder (and not the programmer) that automatically generates and inserts what ever code is needed to facilitate the communication efficiently, based on the IDL types and on the deployment manifest.

Finally, Ambience also uses this information to synthesize a broad range

of higher level functionality. For example, because Ambience can make sense of the bytes in service requests it can auto-generate externally accessible REST [1] end-points to be consumed by web applications, and automatically inject sophisticated authorization code by inspecting parameters for correctness.

3.8 Memory Management

Unlike unikernels, Ambience supports multiple address spaces natively. However, services running in isolated address spaces (i.e. separate Ambience groups) cannot communicate via direct function calls the way that services within the same address space can and, thus must involve the kernel to facilitate efficient passing and returning of necessary information between address spaces. For communication between microservice components mapped to separate nodes, this communication is implemented by automatically inserted serialization/deserialization and RPC communication primitives. However, for cross-group communication within the same node, Ambience makes heavy use of any memory protection features that are available from the node where the kernel is executing.

Ambience's memory management system is designed both to work on hardware systems that include a full-featured MMU (implementing paged virtual memory) as well as low-level, embedded systems with memory "Memory Protection Units" (MPUs) that implement protection (but no address mapping) of physical memory segments. Lack of virtual addressing on machines with MPUs means that Ambience's design must include the ability for services to work with a single address space with protected segments of memory.

Ambience's memory subsystem supports a generalized "page" abstraction called an *address space fragment*. An address space fragment is a range of contiguous memory in one address space that can be zero-copy shared with another address space. Supportable fragment sizes and the ability to support multiple sizes are hardware dependent. On paged systems (i.e. ones with MMUs), a fragment corresponds to a page directly and page sizes cannot be changed. On a microcontroller system, however, the fragment sizes and alignments can change dynamically. For example, MPU and PMP (Physical Memory Protection) hardware found in the ARM [67] and RISC-V [47] architectures correlate protected segment size with alignment. Specifically, a segment of size n (where n is a power of 2) must be aligned on an address that is a multiple of n. Therefore, there is no single fragment layout in such systems and runtime calculations are required to determine a fragment given

a range of memory.

3.9 Memory Sharing

When Ambience provides transparent access to a service in another address space, the kernel will automatically map memory segments from the caller's address space to the callee's address space usually in a read-only fashion to achieve zero-copy calls. Memory for the return values are also supplied by the caller via the `message_builder` type. Fragments belonging to a `message_builder` are mapped with read-write privileges to the callee. Allocation of these regions can be managed by users, but Ambience provides a user space library for simplifying their management, as these regions must be well-aligned. All fragments related to a call are immediately unmapped as soon as the callee completes the request.

Ambience's support for transparent cross-address space mapping is a novel feature. Cross-address space mapping in many existing systems requires non-trivial coordination across the processes accessing the shared memory. On POSIX systems, for example, memory can be shared using `MAP_SHARE`d anonymous pages across a `fork` or using a shared file or shared memory objects. In all such cases, programmers of both the caller and the callee must explicitly setup the sharing and make sure all arguments are in the shared area. While the sharing can support zero-copy communication, it is difficult to automate and/or error prone to program.

In particular, one challenge Ambience's design addresses is that a caller process might supply the same fragments (due to memory space limitation concerns) for multiple concurrent requests to the same server. In this case, the operating system must ensure that the fragments must remain mapped until the last request is completed. Our prototype associates an atomic reference count with each fragment mapped to an address space, and the reference count is maintained at every call and return.

3.10 Efficient Cross-address Space Communication

Cross-address space interprocess communication (IPC) is implemented by Ambience using a combination of memory copies (for small values) and pointer sharing and memory mapping (for larger ones). Because all interfaces are typed, Ambience can generate and automatically insert optimized IPC when microservices deployed to separate address spaces on the same machine communicate.

An Ambience IPC consists of a typed structure on the sending side of the

communication that the operating system replicates on the receiving side. As the kernel has a priori knowledge about the contents of the structure, it can replicate arbitrarily complex data structures and use the most efficient primitives for performing this replication.

Since large objects are passed by pointer, the kernel will follow the pointers and ensure the same data structure is replicated from the sender side to the receiver side. Whether a "true" zero-copy transfer for a data structure referenced by a pointer is possible or not depends on the data structure's alignment in the sender. For example, if a user-space computation attempts to send a string containing 100 characters on a paged system with 4k pages, it is impossible to directly map the page because the string shares the page with other data structures that should not be sent. However, as another example, consider sending a string of size 8193 bytes, on a system with 4K pages, starting at address $4096 * k + 4095$ for some constant k. This means that except for 1 byte at the beginning, the whole string can be mapped directly from the sending address space to the receiving one. For this case, Ambience will allocate an anonymous page to the receiver, copy the single byte to the end and map this at address $4096 * k$ in the destination address space. The rest of the data will be mapped directly at $4096 * (k + 1)$. This partial-copy approach ensures no unintended data is sent from the sender to the receiver service while using as little data copying as possible for large objects.

Note that on MPU systems, partial copying is not possible if the sending data structure not well aligned and sized, and a total copy has to be made since it is impossible to supply different physical memory for the unaligned portions of the data structure without virtual memory support. Zero-copy is still supported for buffers that are well aligned and sized so that they do not share memory fragments with other data structures.

3.11 Interprocess Communication Implementation using C++

Ambience's IPC mechanism is implemented within the kernel and written in C++. Since the approach is type based, Ambience can make use of C++ templates to synthesize the necessary functions. The IPC interface requires that each fundamental type in a message "opt-in" by providing a specialization of the primary `sharer<T>` template. The sharer interface consists of two required static functions: `size_t compute_size(const T& arg);` and `T do_share(Share auto& share, const T& arg);`.

The `compute_size` function returns how many bytes of extra data would `arg` need in the destination address space. For example, for small scalars,

this function always returns 0 since such scalars are always stored within the structure itself. For a string, it would be the size of the string if a total-copy needs to be made, or 0 if the string can be memory mapped. `do_share` performs the actual share through copying or memory mapping. Notice that if `T` is a pointer, `do_share` returns a pointer as well.

Using these specializations, passing an entire structure can be achieved by copying the structure verbatim to the destination address space and transforming each member through the `sharer<T>::do_share` method. The resulting code for sharers is very concise, and the overall sharing code is easy to read and maintain. The sharing functions for all fundamental types are implemented in less than 250 lines of code, most of which are templates. For user defined interface types, the necessary functions are automatically synthesized using these templates since they are by definition a composition of the fundamental types and can be trivially synthesized. Also, because the IPC code is implemented using static polymorphism, Ambience is able to optimize complicated, multi-step shares to a use a single `memcpy` and even to a single SIMD store instruction for smaller parameter packs.

The overall effect of the Ambience memory management functionality is to allow it to implement highly efficient IPC between microservices that are co-located on the same machine, but do not share addresses spaces. The operating system code makes maximal use of memory mapping to implement zero-copy communication on both systems with an MMU and on those with only MPU support (although with greater restrictions for the latter). Finally, the goal of Ambience is to allow the mapping of microservices to protection domains to be a transparent deployment-time decision, and *without* sacrificing performance. In this this way, Ambience makes use of typed interfaces, C++ templates, and static polymorphism to automate IPC optimization.

3.12 Automatic Network Overlay Generation

Ambience services that communicate over a network (i.e. between nodes) do not use overt network communication abstractions. Instead, the Ambience kernel supports efficient message forwarding (using the mechanisms described in Subsection 3.8) and the Ambience image builder automatically synthesizes an application-level network overlay for each deployment using the lidl (Ambience's IDL) specifications for each microservice interface and the deployment manifest. By incorporating the network overlay as first-class operating system abstraction that is automatically constructed at deployment time, Ambience is able to map the same service mesh to different

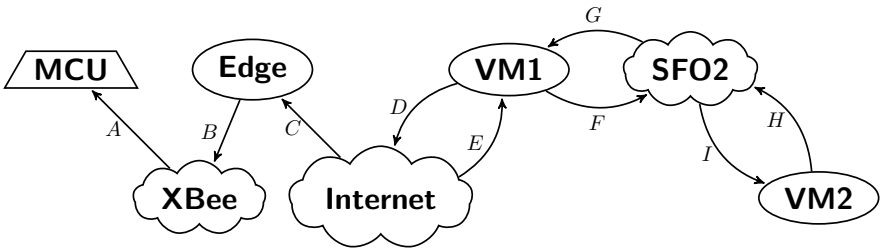

Figure 2: Ambience services import network interfaces when they need to accept off-node requests and export interface when they are required to send requests off-node. The build process generates an network overlay path by modeling the node mesh as a weighted directed graph and finding the shortest path from the desired service's node to the importing node. In this figure, the letters identify each edge and the MCU can import a service from VM2 through the path H, G, D, C, B, A. Note this automatic overlay generation feature allows Ambience to deploy the same service mesh over different heterogeneous, private and asymmetric networks.

heterogeneous network topologies without developer intervention.

From a deployment manifest, the Ambience image builder constructs a graph in both networks and nodes are both represented as vertices. In Figure 2, the networks are marked as "Xbee" represent a low-power Xbee radio network, "Internet" represents the common-carrier Internet, and "SFO2" represents an internal private network. Services that communicate off-node *import* from a network when they will perform requests to a node in that network and *export* to a network when they will serve requests to nodes in that network. An *import* is represented by a directed edge from the network vertex for the last "hop" in the graph to the node vertex where the service is to be hosted and, similarly, an *export* is a directed edge from the hosting node vertex to the network vertex that will be used for off-node that must traverse that network to reach their next hop. The edges are directed to allow for potential asymmetry in the connectivity (e.g. firewall rules that control connectivity) and weighted to allow deployment-time valuation of forwarding paths (e.g. by representing the relative bandwidth, latency, or reliability of alternative network choices).

Using this graph, Ambience constructs a high level communication overlay at the application layer (i.e. in terms of typed service requests rather than routing of untyped packets) as opposed to at the network layer as found on many existing systems [71]. The Ambience image builder consults

the deployment manifest and the graph to determine which services may send and receive requests that traverse more than one network vertex in the graph. Using the edge weights, it computes a least-weighted path between all pairs of communicating services. It then uses the lidl specification for any services that communicate across more than one network vertex to synthesize request forwarders for each node vertex along a a least-weighted path. These forwarders are then compiled into forwarding services and added to the deployment.

For example, using the graph shown in Figure 2, the image builder would create a request forwarder for each request from VM1 to the MCU and it would assign the forwarders to the node marked "Edge" in the figure. Finally, the image builder would set the destination address for any request from VM1 to the MCU to be the forwarder microservice on Edge.

Note that Ambience uses the same lidl specification for each service in a deployment to synthesize and inject IPC code and to generate the overlay forwarding microservices. Using this application-level overlay approach Ambience can transparently join public and private IP networks, low-power networks such as XBee [80, 111], as well as point-to-point links such as USB [7], SPI [37] and UART [52].

Note also that the generated application-level forwarding microservices take full advantage of the fast IPC mechanisms described in Subsection 3.8. Further, since they are synthesized by the image builder, Ambience deploys them within the kernel's isolation domain (as described in Subsection 3.4) for the best possible forwarding performance. In this way, Ambience builds application-level network overlays into the kernels of a specific deployment, making such overlays a first-class operating system abstraction.

Ambience need not deploy a common network software implementation to the radio links and wired networks alike. Returning to the example graph shown in Figure 2, XBee network communications are encrypted by default. Thus, Ambience need not deploy an additional encryption layer such as TLS [39] when messages are traversing edges A and B in the figure, while the communication from Edge to VM1 (traversing edges C and D in the figure) will require TLS, as IP networks are not encrypted by default. These specializations are automatically built into the individual kernel images, along with the required networking software, as dependencies needed to support microservices that are assigned to each node in a deployment.

3.13 End-to-end Access Control & Security

In typical microservice applications, network paths may be publicly accessible and (if software defined networking or network function virtualization is deployed) network traversal may require other services to be invoked. As a result, per "The End-to-end Argument" [96], the microservice mesh is expected to implement access control via an amalgamation of authorization mechanisms such as role- or attribute-based access control, access control lists, or decentralized, token-based authorization primitives [63, 23]. This approach introduces redundant work, precludes a separation of concerns, and limits the operating system's ability to specialize services within the same trust domain.

At the service level, access control takes the form of sanitization; it must answer the the question, "Can the current subject call this procedure with these arguments?" With conventional operating systems, the microservices are the "ends" with respect to end-to-end security. However, Ambience essentially convolves the microservices and operating system abstractions when it builds each image in a deployment. That is, the Ambience images are the "ends" in the terms of an End-to-end argument for security. As a result, the images can implement end-to-end security using code automatically inserted during image construction. Ambience combines the typed interfaces for each microservice with a with formal specification of predicates to ensure per procedure to synthesize access control code in each image of a deployment where it is needed.

To implement this support, Ambience incorporates CAPLets [16], an open source, capability-based authorization framework that runs on both microcontrollers and resource-rich machines. CAPLets requires policies to be defined as capabilities and constraints and written manually by developers. Ambience extends this approach to automatically generate capabilities and constraints from manifests, precluding the possibility of definition mismatch and reducing programmer burden. For requests that take place on the same machine, Ambience uses CAPLets policies directly, as synthesized and automatically injected code, when services communicate across groups. For off-node requests, Ambience automatically injects the CAPLets network protocol, which serializes the request, signs it, adds replay protection, and optionally encrypts it. Once received by the destination node, and the message is deserialized (again using injected code) the CAPLets policy mechanism is invoked.

The secret keys needed for network requests inside the deployment are automatically managed by Ambience through the CAPLets API with no

user visibility or involvement. However, if a service within a deployment must respond to externally generated requests (e.g. from a non-Ambience service) Ambience also generates capability tokens that can be shared (manually) with other parties. In this case Ambience also automatically generates ingress services to act as proxies for the target Ambience services to validate the tokens and implement the CAPLets policies.

This deployment-aware access control improves the pace of development by relieving programmers from implementing access control explicitly, reducing bugs through automatic synthesis of code, and improving runtime efficiency by optimizing away unnecessary checks. It also simplifies administration by providing a uniform authorization infrastructure at the operating system level.

3.14 Lack of POSIX Compatibility

To understand the technology adoption risk associated with Ambience, we analyzed the DeathStar microservices benchmark suite [44] to determine whether a POSIX compatibility layer was an essential feature. The Deathstar benchmarks make use of a test harness framework to implement networking, software dependencies, and platform configuration. We found that while microservices can be ported to Ambience, the test harness (which implements similar functionality to Ambience using conventional Linux system calls) cannot. Because it is not clear how to separate microservice performance from the performance of the test harness in the original Deathstar benchmark implementations, however, we chose not to use them to generate the performance evaluations described in Section 5.

From a code inspection, we found that the microservices within the benchmark suite all communicate with each other not through unstructured pipes or sockets, but over strongly typed interfaces, either via gRPC [50] or Thrift [8]. Further, none of the 33 services make direct use of Linux operating system calls. Even the test harness uses only the Linux `signal` and `exit` system calls, relying on lower-level libraries to interface to the operating system. This finding, coupled with our experience with microservice applications, leads us to believe that POSIX system call compliance is not a requirement in this domain.

The Deathstar suite also incorporates functionality not implemented using microservices (e.g. databases) that make operating system calls. To understand the universality of the Ambience design, we have developed Ambience microservices that provide equivalent functionality. Thus, while the Deathstar suite depends on external functionality that is not, itself,

implemented using microservices, the Ambience equivalents are complete microservice implementations.

These surprising observations have two important consequences for Ambience. First, as with any non-derivative operating system approach, users of Ambience must be concerned with software backwards compatibility, in this case, largely with the POSIX system call interface. The Deathstar benchmarks show that for microservices, this concern is potentially unfounded. Secondly, Ambience kernel need not include a POSIX compatibility layer that would increase its abstraction and implementation "footprints" and potentially degradate its performance. However, we note that Ambience does not have any limitation precluding a POSIX compatibility layer from being implemented in its user space.

4 Deployment Construction

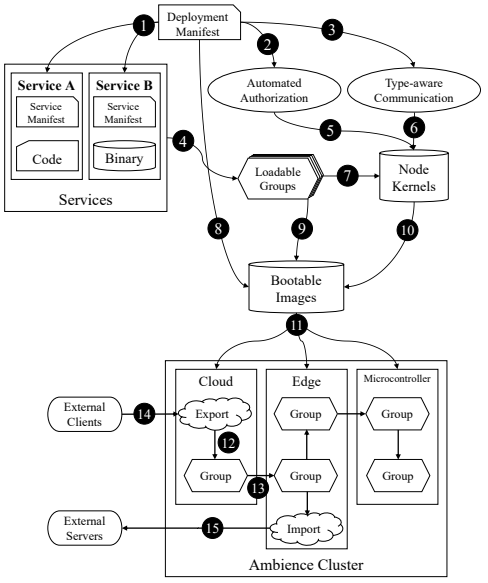

Figure 3: End-to-end overview of how Ambience constructs a deployment.

To achieve both the level of IPC optimization that Ambience enables and also to implement the automatic synthesis of application-level network overlays, Ambience constructs a set of customized images (once for each node) in a deployment. These images must then be installed on each node in a

node-dependent manner to form an Ambience Cluster. In terms of nomenclature, we refer to the entire collection of artifacts as a "deployment" and the set of installed and running images as a "cluster." Figure 3 presents an end-to-end overview of Ambience deployment construction.

A Deployment Manifest document enumerates the services that will be hosted in the cluster, each of which is described by its own Service Manifest. Each Service Manifest specifies the interface types for the service, as well as the interface types of all of its dependencies. The Deployment Manifest also specifies what nodes and networks will be included in the cluster (cf. ❶).

Using the Deployment Manifest to determine the node topology and the type information from the interfaces described in each Service Manifest, Ambience automatically synthesizes code to perform cross-group authorization (cf. ❷) and optimized interservice communication (cf. ❸).

Each Service Manifest also specifies an artifact (either binary or source) that Ambience compiles, if necessary, and links into loadable groups (cf. ❹). Loadable groups are binary modules that are combined with the Ambience-synthesized authorization code (cf. ❺), communication code (cf. ❻) to form node-specific kernel images. Ambience uses metadata associated with each loadable group to pre-allocate certain runtime resources, such as page tables and sockets (cf. ❼).

Ambience combines kernel and group artifacts associated with each specific node to create a bootable image for that node. For x86 or ARM-based fully-featured platforms Ambience generates a bootable ISO disk image and for microcontrollers, it generates a loadable image (cf. ❽, ❾, and ❿.) Using the Deployment Manifest, Ambience also generates memory layouts for all loadable groups. While a global memory layout is not necessary on hardware with paged virtual memory, microcontrollers operate directly on physical memory and each group must be loaded at a suitable location. At this stage, Ambience has the information necessary to allocate memory regions for nodes that do not support virtual memory. This build-time memory mapping relieves the microcontroller kernels from performing runtime memory relocations as well as avoids the use of position independent code to achieve maximum performance.

In the current implementation, bootable images generated by Ambience must be delivered to their respective nodes in a machine-dependent manner (cf. ⓫). For example, for cloud-hosted nodes, the images for each node must be uploaded to the cloud's image registery. Alternatively, for microcontrollers, the images must be installed via the serial interface or flashed to the microcontroller ROM.

Within a deployed cluster, services assigned to different groups that are co-located on the same node communicate using Ambience's synthesized, local Inter Service Communication mechanism (cf. 12). Within a node, Ambience passes arguments and return values between address spaces using its most optimal strategy (e.g. using memory mapping to deliver large amounts of data across protection domains).

To support off-node communication (determined from the Deployment Manifest) Ambience automatically includes the serialization and deserialization code necessary for communication to take place across a network, as well as access control code for protecting these interfaces (cf. 13). It is this component that enables application-level network transparency via automatically-generated network overlays.

To allow Ambience to service requests originating outside of the cluster, Deployment Manifests can specify explicit *Exports* so that Ambience services can support externally facing service interfaces (cf. 14). For example, Ambience supports the access of any internal service via an HTTP REST endpoint that is automatically generated and inserted in a node image with no involvement from the developer.

Similary, Ambience Deployment Manifests can also specify explicit *Imports* for cluster-external services so that Ambience services may issue requests to non-ambience services (cf. 15). However, an imported external service must provide an RPC-style interface so that it can be accessed transparently (i.e. as if it were an Ambience service) via automatically inserted communication code. Consequently, integrating with non-RPC services (e.g. a service supporting a streaming interface) requires the Ambience developer to perform a manual integration.

5 Evaluation

Evaluating the utility of an operating system with a novel system model is challenging. In particular, it is often difficult to make comparisons to existing systems that explain differences (e.g. performance improvements) analytically. Operating system functionality is often a convolution of architectural features that are difficult to study in isolation in a way that yields meaningful comparisons.

In light of this challenge, we focus our evaluation of Ambience on two of its key design goals:

- the ability to deploy end-to-end microservice meshes in different con-

figurations without modifications to the microservice code, and

- the effects of its aggressive build-time optimization strategies on per-node microservice performance.

We note that it is, in fact, Ambience's ability to achieve deployment reconfigurability without recoding *coupled* with aggressive node-level optimization that constitute the basis of its novelty and utility in an IoT setting. Also, we focus on node-level performance since cross-node performance is often dominated by network speed and the performance of the network protocol stack. Optimizations applied to either of these features benefit Ambience and any alternative operating systems equally.

To evaluate deployment reconfigurability, we have developed a motion-triggered "camera trap" application used in wildlife monitoring settings that captures digital images from a remote camera, processes them to perform classification of the images, and stores the classification results in a data repository. We use equivalent implementations for Ambience and the IoT software framework from Azure [83] and report both quantitative and qualitative (e.g. productivity) metrics associated with deploying each version in different configurations.

To evaluate the Ambience design decisions from a performance perspective, we use a set of microbenchmarks to provide isolated measurements of specific functionality. We also use other service-level benchmarks to expose the characteristics of different deployments, such as the effect of service call depth. Quantitatively, we focus on energy use, latency, portability, and scalability. In practical remote IoT settings, sensor and actuator nodes often use battery power (recharged during daylight hours using solar power) and operate on a duty cycle consisting of active periods and periods of low-power dormancy [76, 70, 38]. The minimum duration of the active periods is defined by execution speed and communication delay. Thus power consumption is often correlated with execution duration and, hence, reduced execution duration implies less energy consumption and the use of smaller batteries, a smaller solar array, more active periods per unit time, etc., for the same communication duration. Latency measures the duration of a specific operation or set of operations, and scalability plots the performance of a node as a function of the load it hosts.

The experimental testbed for these evaluations consists of four different computational resources (two microcontrollers, one small-board edge computer, and one cloud) and two different networking technologies. We name the computational platforms **Motion**, **Camera**, **Edge**, and **Cloud** respectively. Their resource configuration is as follows:

- **Motion** is a nRF52840s microcontroller with an ARM Cortex-M4 core running at 64MHz, 256KB of RAM and 1MB of flash memory with an attached motion sensor [55], and an Xbee radio network interface [80].
- **Camera** is an STM32F746 microcontroller with an ARM Cortex-M7 core running at 216MHz, 512KB of RAM, and 1MB of flash memory, an OV5640 CMOS image sensor [88], a motion sensor [55], and both an Xbee radio interface [80] and a 100-Mbit Ethernet interface.
- **Edge** is a single core x86_64 virtual machine with 1GB of RAM running under QEMU-KVM supported by a Linux Kernel 5.15.6 on an AMD 5950x processor running at 3.4GHz, on gigabit Ethernet network interface.
- **Cloud** are two DigitalOcean [40] single core cloud-hosted virtual machines with 1GB of RAM on Intel Skylake processors.

On the **Edge** and **Cloud** platforms, Ambience executes directly on the hypervisor as a stand-alone virtual machine with custom virtio [95] drivers (i.e. it is not "embedded" in another operating system). On the **Motion** and **Camera** microcontrollers, Ambience runs as the native operating system. Both motion sensors are polyelectric infrared (PIR [113]) sensors and all code is implemented using C++.

5.1 Wildlife Monitoring Application

As a motivating application and to demonstrate the flexibility that Ambience makes possible, we describe an end-to-end wildlife monitoring system designed for off-the-grid locations (e.g. research reserves). Physical sensors and cameras employ embedded microcontrollers. The application uses a version of Tensorflow [105] designed for mobile platforms (e.g. smart phones) to process images either on-camera, or off-camera (possibly traversing a network link in the process) on an x86_64 edge server device which then posts the analysis results to the cloud over an Internet connection.

In a typical deployment the motion detector nodes run completely on batteries, making battery life paramount. The camera nodes have solar power, but power usage is still important since the camera uses a battery during nighttime operation that is recharged during daylight hours. In the deployment we use, the edge servers also use batteries, but they are from a large battery complex with a large solar array located in an open space. The cameras communicate with the edge server via an Ethernet network, and the sensors communicate with the camera via low-power Xbee radios.

Using this testbed, we deploy the following service mesh. Note that all of the mesh components are implemented as microservices.

- *Motion Sensor* manages low level hardware events from the PIR motion sensors and forwards them to its event handler.
- *Camera Manager* manages the OV5640 camera using the STM32 Digital Camera Module Interface (DCMI) peripheral, capturing a full sized image every time it is triggered and passes the image data to the frame handler.
- *Detection* implements an animal detection service using Tensorflow on a sub-sampled image. If an animal is detected, the frame is passed to the recognition service. The model used in this service takes up around 320KB and is fully portable across the **Camera** and **Edge** nodes.
- *Recognition* implements an animal recognition service, using Tensorflow but on a higher resolution version of the frame, and classifies the subject. The classification result is passed to the database service.
- *Database* implements an append only log of classification events.

The service mesh deployment configuration shown in Figure 4 represents a typical deployment in a wildlife monitoring setting. The **Motion** device is located near a "stage" (e.g. a watering location) that is imaged by the **Camera** device from a clear vantage. Thus this service mesh configuration consists of four "tiers." **Motion** (tier 1) communicates with **Camera** (tier 2) via Xbee low power radio to trigger an image capture. **Camera** then communicates with **Edge** (tier 3) which has a public Internet connection that it uses to communicate with **Cloud** (tier 4). This four-tiered deployment requires 53 lines of Ambience configuration code (in addition to the code for the service mesh components), all of which is contained in Ambience Manifests.

In settings where the image stage is larger than what a single PIR sensor can cover, a different deployment configuration that uses the motion sensors on both **Motion** and **Camera** together is necessary.

Switching from using only the motion detector on **Motion** to using both motion detectors requires 5 lines of Ambience Manifest change, and no change to the service mesh code itself. Ambience synthesizes the necessary networking overlay code with no additional input from the user.

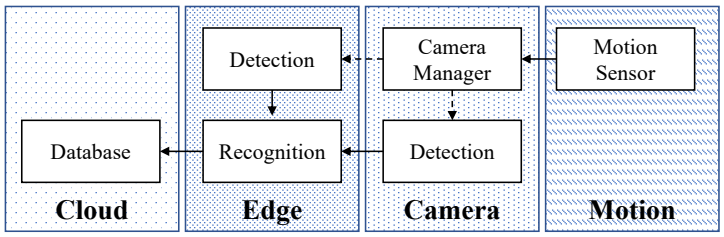

Figure 4: A typical service mesh deployment configuration for the Camera Trap application using the hardware testbed. For brevity, we omit the lower-level services such as logging, timers etc.

We deploy the Tensorflow *Detection* service on **Camera** by default. However if a camera is particularly active causing the **Detection** service to drain the battery at a rate that could threaten an overnight shut down, the service can be transparently offloaded to **Edge** (extending the battery charge duration of **Camera**) and then and moved back once the battery is recharged (to reduce computational load **Edge**). Offloading requires 2 Ambience Manifest configuration lines to change, and no change to the service code itself.

5.2 Microcontroller Latency Analysis

To understand the efficiency of the Ambience implementation on the microcontrollers, in Figure 5 we show a timeline of Ambience events that take place when the *Motion Sensor* microservice is executed on **Motion** and it makes

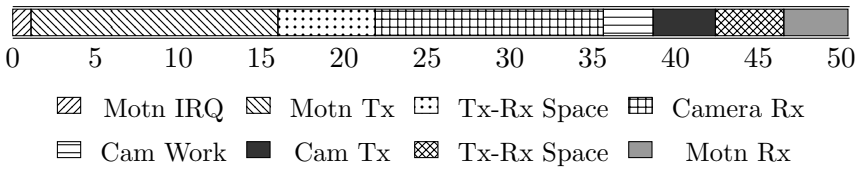

Figure 5: Timeline of the events captured using a logic analyzer when the *Detection* service is deployed on the Edge node and the Motion service is deployed on the Motion node. Units are in milliseconds.

a service request of the *Camera Manager* service on **Camera** to capture an image. The timeline units are milliseconds and the data was gathered with a logic analyzer attached to both microcontrollers. Our goal, with this study, is to understand the efficiency of the Ambience interaction between services hosted on microcontrollers in the camera trap application. In particular, we wanted to understand the prospective battery life of **Motion** since it does not have a solar array to recharge its battery.

On **Motion**, an IRQ (approximately 1 ms) triggers Ambience to start the process of sending a request to **Camera**. Next, approximately 15 ms (marked Motn Tx in the figure) are required to activate the Xbee radio through an on-board serial interface. Sending a message over the radio (marked Tx-Rx space in the figure) requires approximately 5 ms of communication latency during which time both the radios on **Motion** and **Camera** are active. To transfer the message from the Xbee radio through the serial interface on **Camera** requires approximately 15 ms (marked Camera Rx in the figure). The Ambience-induced workload necessary to receive the message and send a response (marked Cam Work) is approximately 2 ms, followed by 2 ms needed to transfer the short service response across the serial interface on **Camera** to the Xbee radio (marked Cam Tx in the figure). The 5 ms of network latency for the response is followed by 2 ms for the response to traverse the serial interface between the radio and the microcontroller (marked Motn Rx) on **Motion**.

Note from Figure 5 that approximately 40 of the 50 milliseconds are devoted to serial communication with radios on each microcontroller board. During these periods, the microcontroller processors are in low-power sleep and the transfers are made entirely using DMA hardware to conserve power. In particular, the **Motion** microcontroller is awake for less than 3% of the entire operation and spends less than 20 μJ for the entire event with the radio requiring 4.32 mJ. Using a battery cell with 13Wh capacity [2], this

energy consumption is sufficient for the device to service approximately 10 million events detected and transmitted over an XBee network. At reasonable event rates and quiescent current leakages, the battery could last multiple years, demonstrating Ambience's abstractions are efficient and effective enough to support low power applications as well as high performance ones.

5.3 Edge Platform Overheads

To facilitate reuse and scaling, individual microservices often implement very narrow functionalities, which are composed to form higher level services. Such services are deployed in separate trust domains (processes, address spaces) to achieve isolation. Services then use the IPC mechanisms implemented by the operating system to communicate between domains. Previous work [44, 61] notes that microservices can have quite large communication-to-computation ratios. From a performance perspective, cycles spent for computation is "useful work" and cycles spent for communication is "overhead" – a cost required to implement the useful work. Thus the ratio of communication cycles (cost) to computation cycles (benefit) is a simple representation of the cost/benefit ratio associated with a microservice deployment. We term this metric the *overhead ratio*.

To study the overhead imposed by the platforms, we implement a pair of *Camera Manager* and and *Detection* services 3 ways: natively on Ambience, using Azure IoT SDK, and using lidl on Linux over Unix Domain Sockets. To avoid differences in workload caused by the effects of numerical precision and Ambience's inlining of application and operating system code, we use parameterizable "mock" versions of both services that allow the the compute cycle counts for the microservice portion of the workloads to be set explicitly. We then compare the same services across three platforms in different configurations. Our mock *Detection* service allows us to specify an exact cycle count (regardless of architecture) to use to subsample an image in a given frame. We also experiment with multiple concurrent requests to try and amortize communication costs.

Figure 6 presents the overhead results. The y-axis in each graph is the overhead ratio computed as

$$overhead\_ratio = (total\_cycles - work\_cycles)/work\_cycles \qquad (1)$$

We set the number of work cycles explicitly in the benchmark and measure the total processor time using the real-time clock on **Edge**, dividing the overall time by the processor clock rate.

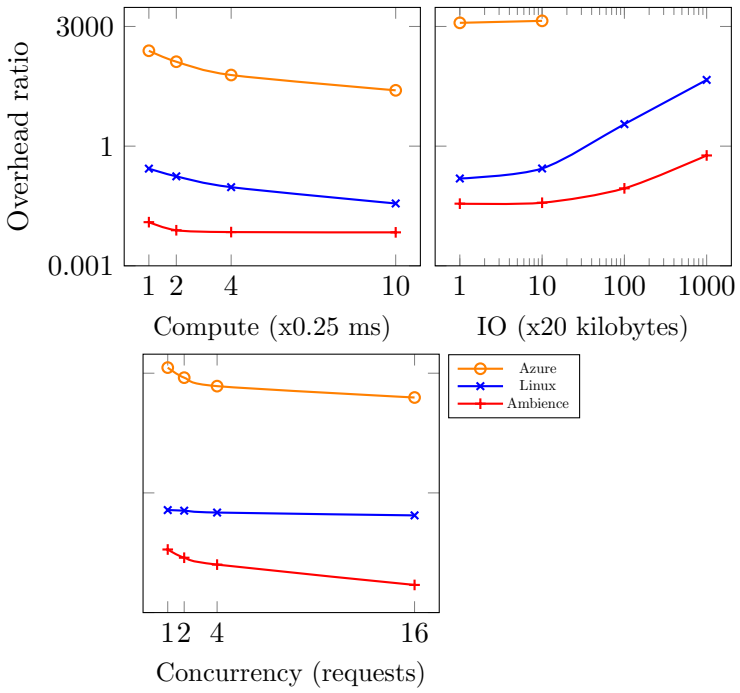

Figure 6: Overhead ratios for three benchmark regimes. The overhead ratio is computed as number of non-compute processor cycles to compute cycles processor cycles. In the Compute benchmark (left), the x-axis shows increasing compute workload summed over 16 concurrent requests and a fixed, 200KB message. For the IO benchmark (middle), the x-axis shows increasing request message size for a single request (concurrency 1) requiring 0.25 ms of compute cycles. For the Concurrency benchmark (right), the x-axis shows increasing concurrent requests, each requiring 0.25 ms of compute and a 200KB message size. Both the x-axis and the y-axis are shown on a log scale in each graph.

In Figure 6, we show the overhead ratio on the y-axis (on a log scale) for three different benchmark regimes using the "mock" *Detection* benchmark. The leftmost graph of Figure 6, shows results for a "Compute" benchmark. The x-axis corresponds to increasing compute workload for each of 16 concurrent requests that each require a 200 KB payload. The center graph shows results for increasing message payload along the x-axis for a single request requiring 0.25 ms. Finally, the rightmost graph shows increasing message concurrency along the x-axis, for 0.25 ms requests, each requiring a 200KB message payload. Note that Azure sets a 256KB limit on message payload and that the x-axis is depicted on a log scale in each graph. A ratio of greater than 1.0 gives the number "extra" non-work cycles necessary to accomplish a single cycle of useful work.

The evaluation shows that for low-compute, high-communication scenarios, Linux imposes significant overheads, with an communication-to-computation ratio of as much as 61. In contrast, the Azure overhead ratio of 1381 is almost two-orders of magnitude higher than that for the highest overhead native Linux implementation (large message sizes in the IO graph). By comparison, the overhead ratio for Ambience ranges from between 0.008 to 0.59. That is, Ambience is between two and four orders of magnitude more efficient than Linux, and between four and six orders of magnitude more efficient than Azure, in terms of communication-to-computation overhead ratio. Further, the overhead ratio for Ambience is never greater than 1.0 in these experiments indicating that the Ambience optimizations are able to amortize each overhead cycle against multiple work cycles in these benchmark regimes.

The predominant reason for this significant difference in efficiency is that the Linux and Azure IPC mechanisms require data to be copied when it traverses a protection domain. While this approach induces relatively little overhead when the data is small and the computations are lengthy, it creates significant overheads in a microservice context where each service performs a simple computation and the overall application is a large composition of such service invocations. For example, the *Detection* service requires the entire image to be passed between address spaces, but once it has been moved, the compute requirements are relatively small (since it subsamples the image). Thus the communication-to-computation ratio is potentially large when the entire image is copied into the protection domain hosting the *Detection* service.

Further, we note from the analysis of the DeathStar benchmark suite [44] discussed in Subsection 3.14 that simple computations are common to many of the microservice requests it embodies. For cloud-based microservices,

the resulting overheads may not be a serious impediment, but in an IoT context, where the overhead results in additional power consumption, this per-request efficiency is an important point of optimization.

Also of note is that all systems show improvements as concurrency increases, since certain costs such as i-cache and TLB misses and context switches amortize between concurrent requests, with Ambience improving the most while being the best overall. We expand our analysis of IPC overheads below using microbenchmarks. Note also that Azure imposes a hard limit of 256KB [15] on message size, preventing us from using it for comparative purposes with large messages. However, these results demonstrate that Ambience's aggressive type-aware, specialized IPC mechanism can dramatically reduce cross-domain overheads for microservices on a single node.

5.4 Portability of Cloud IoT SDKs

To compare the portability and network transparency features of Ambience to the state-of-the-art, we implemented as much of Camera Trap application as possible using Azure's SDK, since it, like Ambience, features the ability to run on the cloud, edge and microcontrollers. The embedded SDK for microcontrollers is not the same SDK as for the cloud or edge devices, meaning that moving software written for one to the other requires substantive code changes, testing to make sure that both implementations are equivalent, etc. This impediment is primarily due to divergent APIs, different programming models, and different deployment models needed to host a microservice in a specific tier. Specifically, on the cloud and edge, programmers must make use of Linux APIs, whereas on the microcontroller devices, the developer has the choice of FreeRTOS [43, 51], AzureRTOS [103] or "native" bare metal coding, each of which provides a unique and incompatible set of abstractions, making writing a piece of code that runs portably across the various platforms in a deployment labor-intensive and error prone.

Additionally, the communication infrastructure required by the Azure SDK makes use of MQTT for all interservice communication. Thus colocating two services on the same node incurs MQTT communication overhead even for the local communication. As a result, even if the operating systems abstractions could be unified across tiers in some future version of the Azure SDK, the communication overheads would remain comparatively high compared to Ambience. Finally, the hard limit on message sizes in the Azure SDK also makes certain service mesh decompositions impractical.

Ambience support for network overlay synthesis is a further aid to deployment portability. The available Azure software does not include support

for Xbee radio communication. Thus, to implement the communication between microcontrollers using the Azure SDK required we code a custom XBee driver for the Azure implementation that is based on the one automatically inserted by Ambience.

Finally, although the Azure software stack supports two microcontrollers models in the same "family" as the ones we had available for **Motion** and **Camera**, the specific models supported by Azure were unavailable due to supply-chain delays and, as a result, we could not obtain them to use with an Ambience port. Instead, we attempted unsuccessfully to port the Azure stack to **Motion** and **Camera** which share the same processor cores and peripherals with the Azure supported platforms. As a result, while we can make qualitative observations about microservice portability, the sensitivity of Azure to differences in platform-specific features made an end-to-end quantitative performance comparison of the Camera Trap application ultimately infeasible.

5.5 Microbenchmarks

While the wildlife camera trap application exemplifies the utility of Ambience in an end-to-end IoT context, its complexity makes isolating the effects of specific design choices challenging. To permit a more focused analysis, we detail the performance of individual design features using a combination of synthetic benchmarks and benchmarks extracted from more complex applications. Together, we refer to these as "microbenchmarks" since they each test a specific Ambience feature or subsystem.

Interface Benchmarking

To evaluate the effects of Ambience's integration of interface type information into the kernel, we created a variety of synthetic microservice interfaces designed to cover a representative set of results. Specifically, we constructed benchmarks with interfaces consisting of scalars of a uniform type, scalars of mixed types and relatively large strings and buffers. We executed the benchmarks while increasing the sizes of the arguments to identify any potentially hidden overheads. For each interface, we executed 10K requests using four implementation strategies and measured the average latency and overall throughput of each.

We compare four different interface strategies for implementing each interface. The **User** strategy represents typical interprocess communication (IPC) using byte wise copy between user spaces and the kernel (e.g. Linux

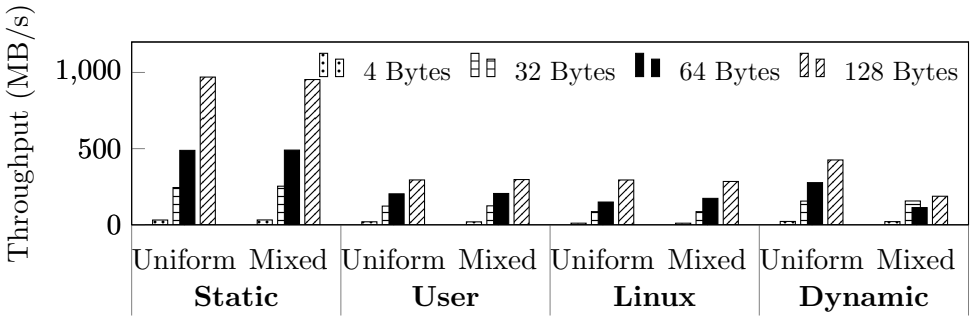

Figure 7: Average throughput for passing scalar arguments (of varying total size) between two address spaces. Uniform depicts the case where the scalars are all of the same type and mixed shows the effect of multiple scalar types in the argument payload. Each experiment is repeated 10,000 times and the units are megabytes/second.

pipes). All type erasure occurs in the user space, the kernel copies the bytes from the client to the server and the server deserializes the buffer. For **Linux**, we re-implemented the User strategy using pipes on Linux 5.15.6 and the same serialization/deserialization code in each comparative experiment.

In the **Dynamic** strategy, the user space code sets up a vector of pointers to arguments and tells the kernel the types of the pointers dynamically. The kernel then performs the sharing to the other address space, and creates a new vector of pointers to arguments the server address space can access. The advantage of this approach over the User strategy is the kernel can automatically map pages for large buffers (although it cannot precompile optimized sharing for each argument).

The **Static** strategy (the Ambience default strategy) is one in which the user space sets up a tuple of typed arguments and passes a pointer to this tuple to the kernel. Since the kernel has been compiled with type information from the interface for the system call it "knows" the structure of the data the tuple at compile time. It again creates the same structure on the server address space by either copying the arguments or mapping pages but the decision is "hard coded" into the kernel and optimized during the kernel image build.

Figure 7 shows the throughput achieved by the four strategies (Static, User, Linux, and Dynamic) with four different payload sizes (4 bytes, 32 bytes, 64 bytes, and 128 bytes). Each payload size consists of either a single

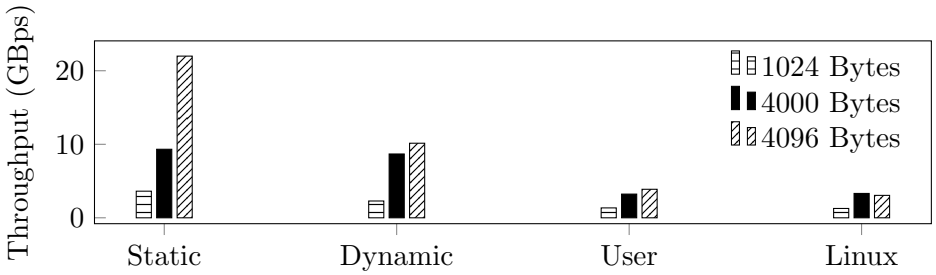

Figure 8: Average throughput for passing larger buffers across address spaces. User and Linux fall short as they always perform copies.

scalar type (denoted "Uniform" or a mixed set of scalars (denoted Mixed) where the sizes of the constituent scalars sum to the payload size. For example, the Uniform 32-byte payload consists of four 64-bit integers while the Mixed 32-byte payload comprises a 64-bit unsigned integer, a 64-bit signed integer, an 8-bit boolean, a signed 8-bit integer, a 32-bit floating-point scalar, a 16-bit signed integer, and a 64-bit floating point scalar.

From the figure, each strategy achieves approximately the same throughput performance for small payload sizes, except the Dynamic strategy which incurs a noticeable throughput penalty when the payload is Mixed. Note that the Static strategy (the Ambience default) achieves higher throughputs as the payload size increases, with little difference between the Uniform and Mixed payloads.

Further, comparing the Static and Dynamic strategies shows the effect of compile-time optimizations. While Dynamic uses the same primitives as Static in a program, for Dynamic, Ambience must traverse a list and make an indirect function call for each argument. The effects of this implementation is most apparent when there are many parameters of different types (i.e. Mixed workload) causing substantial branch mis-predictions and I-Cache invalidations. Static, User, and Linux on the other hand have no virtual function calls and there is no list to traverse: parameters are simply a packed, contiguous tuple. On top of the cache-friendliness, the static type information unlocks inlining opportunities for the compiler. For example, when passing 16 scalar arguments, the Static strategy emits a single large `memcpy` as opposed to 16 small ones. For passing few, well aligned large buffers (starting at around 100KB), Dynamic achieves similar results when its cache and inlining disadvantages are overshadowed by the efficiencies of page table manipulation.

Note also that because User and Linux need to perform multiple copies of large buffers (one for in-process serialization, another for IPC), they cannot achieve the high throughput afforded by direct page mapping. However, as they can make use of the static types in user space, they still outperform Dynamic for the mixed-type workload.

Figure 8 shows similar throughput results for larger buffers that are both page aligned and unaligned. Specifically the 1024 and 4000 are not page aligned sizes and thus must be copied for all cases. However, the 4096 byte buffer can be directly mapped for the strategies that can take advantage of page remapping.

Overall, the results show that the Static strategy is superior in both the small payload and large buffer experiments achieving 2.66x and 3.18x (respectively) higher average and 4.08x and 2.29x (respectively) higher maximum throughput. Further, the Linux and User results are almost identical results since they are implemented in a very similar manner.

Note that Linux user-space page-mapping support (i.e. *mmap*) cannot be used to automatically implement mapped arguments in system calls. Linux does not currently include support for mapping arbitrary pages from one address space into another, temporarily, with shared ownership. Using `mmap`, two processes could implement their own application-level emulation of the Ambience mechanisms however they would need to explicitly allocate memory in those pages, since `mmap` cannot map existing, anonymous pages to another address space. Further, if the same page is supplied as an argument in multiple concurrent requests, the page must be unmapped only when the last request completes (which would also need to be implemented as bespoke application code). Again, `mmap` does not support automatically mapping the same page multiple times and an unmap function that uses reference counting.

Scalability Benchmarking

Ambience supports stackless coroutines as its basic computational model. To explore the efficiency of this choice, particularly with respect to service request scalability, we implemented two different versions of a recursive and caching, DNS-like, name resolving service where clients make requests to resolve names to network addresses. One implementation uses Ambience coroutines while the other employs an implementation of fibers [66] for Ambience.

The experiment consists of two terminal resolvers (one implemented with coroutines and the other with fibers) each storing half of the known domains.

A single client of each resolver generates 10,000 requests for uniformly random selected domains (including some invalid domains). The requests are sent in batches where each request in a batch is serviced concurrently and the average request time is computed as the total time to complete all requests divided by 10,000.

For the fiber version, we use a stack size of 32KB which we note is moderately sized compared to the space (often tens of megabytes) allocated for stacks by other systems. In contrast, the coroutine version dynamically allocates a specifically-sized continuation frame of 627 bytes which is the minimum needed for each request.

Because the resolver is recursive, if the requested hostname is not cached, it will make a request to one or more of its upstream resolvers (we use a university campus DNS service as the most immediate upstream in these experiments) and wait. If the result is in the cache, it responds immediately. Once a request completes, all resources are freed. This means that if a request completes without any blocking, it consumes memory for only a very short time. Therefore, if the cache hit fraction is N, only $B * (1 - N)$ requests consume memory in a batch of requests having size B. We have tuned the request streams so that N is approximately 0.5 in the experiments we conduct.

Figure 9 compares the scalability of each approach in terms for four metrics: average request throughput, memory usage, TLB misses, and CPU processor cache misses, each as a function of increasing request concurrency. The throughput units are requests per second, memory usage is measured in bytes, TLB misses and cache misses are counts. Note that the cache-miss graph is on a log scale.

The results show that coroutines achieve approximately $2\times$ greater maximum throughput compared to fibers while supporting a maximum of $50\times$ the number of requests in the same memory footprint. This comparison illustrates both the runtime overheads and excess memory that a fiber implementation incurs, compared to coroutines, particularly when requests must block waiting for an upstream response.

The results also show that maximum throughput occurs when each batch of requests is size 64 (i.e. when $B = 64$) for both coroutines and fibers. To investigate this phenomenon in detail, we implemented kernel support for the Performance Monitor Counters for the AMD 5950x processor and gathered information on cache and TLB misses.

For fewer than 64 coroutines, the increase in throughput stems from a dramatic reduction in the number of context switches leading to fewer TLB misses and a relatively low cache miss rate as shown in the bottom

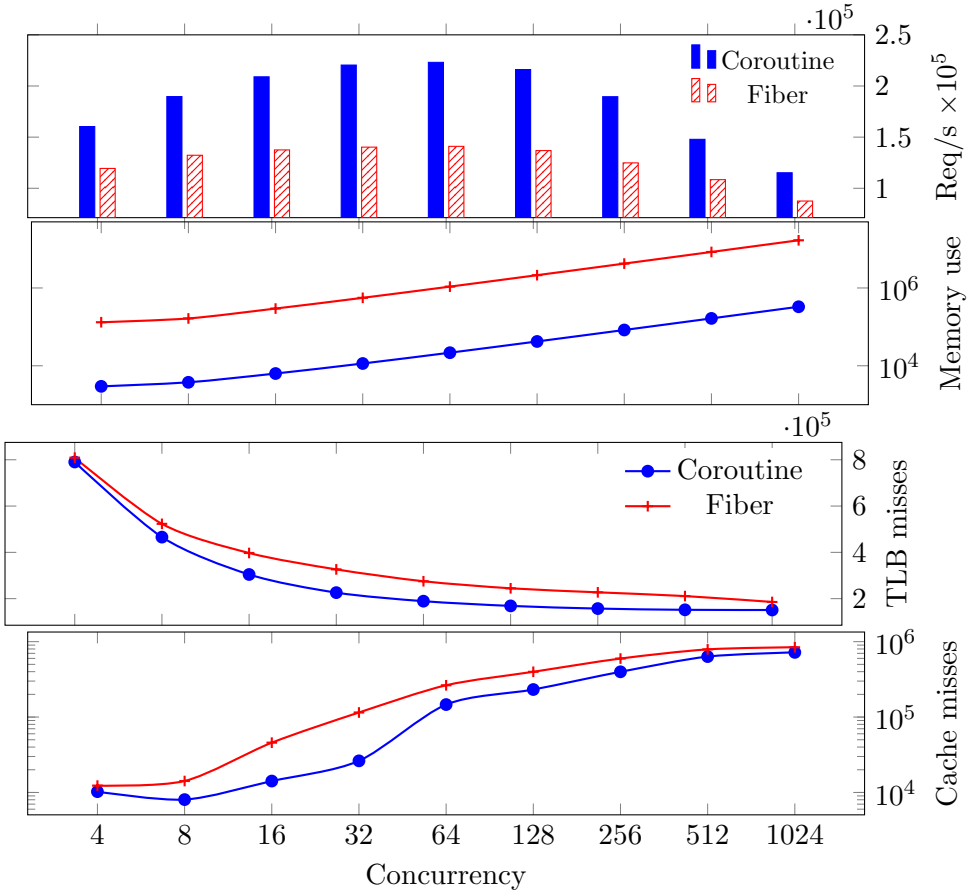

Figure 9: Comparing Ambience coroutines and fibers in terms of request throughput (requests per second $\times 10^5$), memory use (bytes), TLB misses ($\times 10^5$) and Cache misses (on a log scale) as concurrency increases

two graphs of Figure 9. However, every additional concurrent request grows the working set as individual, dedicated pages are created for each request and response. As concurrency increases beyond 64 requests, the TLB miss count continues to drop, but cache miss rate increases. The point at which these effects balance (yielding the highest throughput) is when the request concurrency is 64.

Cross Isolation Group Benchmarking

A key design feature of Ambience is the ability to change trust domain topology at deployment time without code modification. To explore the effect of this feature on performance, we compare a deployment scenario in which the client and the DNS resolver are trusted equally by the deployer to a scenario in which the deployer places them in separate trust domains. Note that in a Linux microservice context, this choice is not typically available – the deployer must use separate isolation domains regardless of the trust architecture associated with a specific deployment.

To evaluate this design feature, we placed the coroutine recursive resolver in the same group as the client and compare that performance to the performance shown in Figure 9 for the coroutine version where the client and resolver are in separate Ambience groups.

Figure 10 shows the comparative throughput in requests per second ($\times 10^6$). Note that the solid bars in the figure are generated from the data shown in Figure 9 which uses units an order of magnitude less than those in Figure 10. For example, in Figure 9, the average cross group throughput for concurrency level 64 is approximately 2×10^5 requests per second which is shown in Figure 10 as 0.2×10^6 requests per second. This change of scale is necessary because colocation of the client and the service within the same security group results in more than an order of magnitude increase in throughput. Critically, this benchmark comparison did not require code changes to either the client or resolver microservice code. Only the Ambience deployment manifest differs between the two deployment isolation topologies compared in Figure 10.

Benchmarking "Kernelized" Services

For deployments where the microservices and the Ambience kernel are equally trusted (e.g. on a device with a single owner who wishes to dedicated it to running one or more microservices), Ambience allows the microservices to share the kernel's address space. Note that this deployment choice, again,

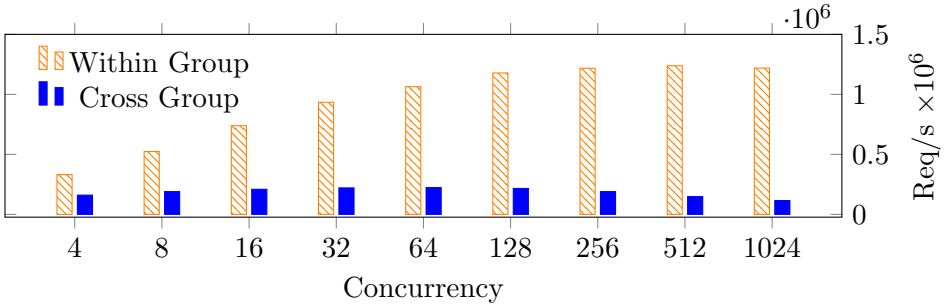

Figure 10: Comparing request throughput when the resolver and the client are deployed in the isolation group and separate isolation groups. The units are requests per second $\times 10^6$.

	User Space	Kernel
Edge PC	9.418	0.348
Motion MCU	58.765	7.265

Table 1: Average time to first service instruction from a hardware interrupt on **Edge** and **Motion** when the service is deployed in user space versus in-kernel. Units are microseconds.

only requires manifest declarations indicating kernel deployment with no code changes to the services themselves. Also, it is possible to "kernelize" multiple microservices with the same kernel, in contrast to a unikernel approach where each service may be comingled with its own, separate kernel.

Kernelized services (where they can be deployed according to the localized trust architecture) permit low-latency request responses because they avoid the context switching overhead necessary for user-space execution. Table 1 compares the average latency (measured in microseconds) between when an interrupt occurs and the first instruction of a microservice is executed on **Edge** and on **Motion**. Note that for **Edge** the timing is gathered within the virtual machine (i.e. after the interrupt has been vectored to the virtual machine by the hypervisor) in both cases.

In-kernel deployment reduces time to service latency by 27× for the x86_64-based **Edge** and by 8× for **Motion** microcontroller. The performance improvement is because when the service is in-kernel, Ambience can immediately schedule the service on the kernel job queue without initializing the memory protection data structures necessary for a full context switch.

6 Conclusion

We present Ambience, a new operating system for efficiently executing and deploying microservices. It does so via a novel combination of abstractions for isolation, asynchronous control flow, statically typed interfaces, automatic network overlays, capability-based access control, and separated declarative deployment orchestration. This combination makes it possible to optimize individual services and the kernels running those services and to reduce the overheads that hamper the use of general purpose operating systems on resource constrained machines and devices.

To achieve device portability across resource scales in an IoT setting, the Ambience design emphasizes memory parsimony and execution efficiency so that it is capable of implementing microservices on the most resource restricted devices. Its use of typed service interfaces and C++ compile-time optimizations allow it to scale these efficiencies "up" to more fully featured and resource-rich platforms (such as single board computers, edge systems, cloud computing instances, and Linux systems) without modifications to the microservices code or developer intervention.

The empirical evaluation of Ambience demonstrates both the ability to deploy unmodified microservices at different scales in an end-to-end IoT application and the performance opportunities and costs associated with different deployment configurations for the same set of microservices. This deployment flexibility is novel in that no other existing operating system or operating system style decouples IoT application development from deployment to this same extent. At the same time, this additional flexibility does not impose a performance penalty relative to the state of the art. Ambience is often between one and three orders of magnitude more efficient than commercial, multi-resource IoT frameworks.

To achieve these results, Ambience sacrifices "traditional" operating systems abstractions for more flexible isolation and control flow. In this respect, it is non-derivative and not backward compatible with other operating systems (although many of its features are inspired by and partially shared with other different systems). Part of the rationale for the "clean-slate" design approach stems from its focus on microservices which, at present, do not make heavy use of typical operating system abstractions directly.

For IoT, where the proliferation of devices, deployment requirements, and distributed security concerns span resource scales from small embedded systems to the cloud, Ambience postulates a unifying operating system that is designed to "tame" this heterogeneity. At the same time, it recognizes that for IoT, the *in situ* requirements defined by individual deployments

should allow the same set of microservices that comprise an application to be deployed in different configurations without the need for recoding or developer intervention.

References

[1] Representational state transfer, Jul 2021. https://en.wikipedia.org/w/index.php?title=Representational_state_transfer&oldid=1035705322.

[2] Panasonic 18650 battery datasheet. https://www.imrbatteries.com/content/panasonic_ncr18650b-2.pdf.

[3] Leila Abdollahi Vayghan, Mohamed Aymen Saied, Maria Toeroe, and Ferhat Khendek. Deploying microservice based applications with kubernetes: Experiments and lessons learned. In *International Conference on Cloud Computing (CLOUD)*, page 970–973, Jul 2018.

[4] Alexandru Agache, Marc Brooker, Alexandra Iordache, Anthony Liguori, Rolf Neugebauer, Phil Piwonka, and Diana-Maria Popa. Firecracker: Lightweight virtualization for serverless applications. In *17th {usenix} symposium on networked systems design and implementation ({nsdi} 20)*, pages 419–434, 2020.

[5] P Albertos, A Crespo, and José Simó. Control kernel: A key concept in embedded control systems. *IFAC Proceedings Volumes*, 39(16):330–335, 2006.

[6] Marcelo Amaral, Jorda Polo, David Carrera, Iqbal Mohomed, Merve Unuvar, and Malgorzata Steinder. Performance evaluation of microservices architectures using containers. In *2015 IEEE 14th International Symposium on Network Computing and Applications*, pages 27–34. IEEE, 2015.

[7] Don Anderson. *USB system architecture*. Addison-Wesley Professional, 1997.

[8] Apache thrift. https://thrift.apache.org/.

[9] Store Arduino Arduino. Arduino. *Arduino LLC*, 372, 2015.

[10] Boost.asio, 2021. https://www.boost.org/doc/libs/1_76_0/doc/html/boost_asio.html.

[11] Linux Kernel Authors. Efficient io with io_uring. *Linux Documentation*, page 16, 2021.

[12] AWS Greengrass. `https://aws.amazon.com/greengrass/` [Online; accessed 12-Sep-2019].

[13] Aws lambda – serverless compute - amazon web services. `https://aws.amazon.com/lambda/`.

[14] Aws iot - amazon web services. `https://aws.amazon.com/iot/`.

[15] Understand azure iot hub quotas and throttling — microsoft docs. `https://github.com/MicrosoftDocs/azure-docs/blob/4764ddf7bb4c71e33e58dfe67dbd7361ee0fb6fa/includes/iot-hub-limits.md`.

[16] Fatih Bakir, Chandra Krintz, and Rich Wolski. Caplets: Resource aware, capability-based access control for iot. In *2021 IEEE/ACM Symposium on Edge Computing (SEC)*, pages 106–120. IEEE, 2021.

[17] Fatih Bakir, Rich Wolski, Chandra Krintz, and Gowri Sankar Ramachandran. Devices-as-services: Rethinking scalable service architectures for the internet of things. In *USENIX HotEdge*, 2019.

[18] I. Baldini, P. Castro, K. Chang, P. Cheng, S. Fink, V. Ishakian, N. Mitchell, V. Muthusamy, R. M. Rabbah, A. Slominski, and P. Suter. Serverless computing: Current trends and open problems. *CoRR*, 2017.

[19] Paul Barham, Boris Dragovic, Keir Fraser, Steven Hand, Tim Harris, Alex Ho, Rolf Neugebauer, Ian Pratt, and Andrew Warfield. Xen and the art of virtualization. *ACM SIGOPS operating systems review*, 37(5):164–177, 2003.

[20] Adam Belay, George Prekas, Christos Kozyrakis, Ana Klimovic, Samuel Grossman, and Edouard Bugnion. Ix: A protected dataplane operating system for high throughput and low latency. In *USENIX Symposium on Operating Systems Design and Implementation*, 2014.

[21] Brian N Bershad, Craig Chambers, Susan Eggers, Chris Maeda, Dylan McNamee, Przemyslaw Pardyak, Stefan Savage, and Emin Gün Sirer. Spin: An extensible microkernel for application-specific operating system services. In *Proceedings of the 6th workshop on ACM SIGOPS*

European workshop: Matching operating systems to application needs, pages 68–71, 1994.

[22] R. Bias. The history of pets vs cattle and how to use the analogy properly, Sep 2016. http://cloudscaling.com/blog/cloud-computing/the-history-of-pets-vs-cattle/.

[23] Arnar Birgisson, Joe Gibbs Politz, Ulfar Erlingsson, Ankur Taly, Michael Vrable, and Mark Lentczner. Macaroons: Cookies with contextual caveats for decentralized authorization in the cloud. 2014.

[24] Alfred Bratterud, Alf-Andre Walla, Harek Haugerud, Paal E. Engelstad, and Kyrre Begnum. Includeos: A minimal, resource efficient unikernel for cloud services. In *IEEE International Conference on Cloud Computing Technology and Science (CloudCom)*, page 250–257, Nov 2015.

[25] Alan Brown, Simon Johnston, and Kevin Kelly. Using service-oriented architecture and component-based development to build web service applications. *Rational Software Corporation*, 6:1–16, 2002.

[26] Zach Brown. Asynchronous system calls. In *Proceedings of the Ottawa Linux Symposium (OLS)*, pages 81–85, 2007.

[27] Cap'nproto: Introduction. https://capnproto.org/.

[28] The centos project, 2021. https://www.centos.org/.

[29] Cloud application platform — heroku. https://www.heroku.com/.

[30] constexpr specifier - cppreference.com, 2021. https://en.cppreference.com/w/cpp/language/constexpr.

[31] Fedora coreos, 2021. https://getfedora.org/en/coreos?stream=stable.

[32] craigshoemaker. Azure functions overview. https://docs.microsoft.com/en-us/azure/azure-functions/functions-overview.

[33] Li Dan, He Erhu, Zhang Yanrong, and Zhang Yu. Teaching and practice innovation of embedded system design course based on proteus and keil-electronic stopwatch as an example. *International Journal of Smart Home*, 9(4):127–134, 2015.

[34] Deathstarbench code repository. https://github.com/delimitrou/DeathStarBench accessed December 29, 2023.

[35] Debian – the universal operating system, 2021. https://www.debian.org/.

[36] Devops, Aug 2021. https://en.wikipedia.org/w/index.php?title=DevOps&oldid=1038068265.

[37] Piyu Dhaker. Introduction to spi interface. *Analog Dialogue*, 52(3):49–53, 2018.

[38] Ruchi Dhall and Himanshu Agrawal. An improved energy efficient duty cycling algorithm for iot based precision agriculture. *Procedia Computer Science*, 141:135–142, 2018.

[39] Tim Dierks and Christopher Allen. Rfc2246: The tls protocol version 1.0, 1999.

[40] Digitalocean. https://digitalocean.com/.

[41] Docker compose, Aug 2021.

[42] Dpdk home. https://www.dpdk.org/.

[43] Freertos - market leading rtos, 2021. https://www.freertos.org/index.html.

[44] Yu Gan, Yanqi Zhang, Dailun Cheng, Ankitha Shetty, Priyal Rathi, Nayan Katarki, Ariana Bruno, Justin Hu, Brian Ritchken, Brendon Jackson, et al. An open-source benchmark suite for microservices and their hardware-software implications for cloud & edge systems. In *Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 3–18, 2019.

[45] Wolfgang Gentzsch, Lucio Grandinetti, and Gerhard Robert Joubert. *High speed and large scale scientific computing*, volume 18. IOS Press, 2009.

[46] Gareth George, Fatih Bakir, Rich Wolski, and Chandra Krintz. Nanolambda: Implementing functions as a service at all resource scales for the internet of things. In *2020 IEEE/ACM Symposium on Edge Computing (SEC)*, page 220–231, Nov 2020.

[47] Nicolas Gerlin, Endri Kaja, Monideep Bora, Keerthikumara Devarajegowda, Dominik Stoffel, Wolfgang Kunz, and Wolfgang Ecker. Design of a tightly-coupled risc-v physical memory protection unit for online error detection. In *2022 IFIP/IEEE 30th International Conference on Very Large Scale Integration (VLSI-SoC)*, pages 1–6. IEEE, 2022.

[48] David B Golub, Daniel P Julin, Richard F Rashid, Richard P Draves, Randall W Dean, Alessandro Forin, Joseph Barrera, Hideyuki Tokuda, Gerald Malan, and David Bohman. Microkernel operating system architecture and mach. In *In Proceedings of the USENIX Workshop on Micro-Kernels and Other Kernel Architectures*, pages 11–30, 1992.

[49] Cloud functions, 2021. `https://cloud.google.com/functions`.

[50] gRPC, 2021. `https://grpc.io/` [Online; accessed 1-Aug-2021].

[51] Fei Guan, Long Peng, Luc Perneel, and Martin Timmerman. Open source freertos as a case study in real-time operating system evolution. *Journal of Systems and Software*, 118:19–35, 2016.

[52] Aditya Gupta. Uart communication. In *The IoT Hacker's Handbook*, pages 59–80. Springer, 2019.

[53] Andreas Haas, Andreas Rossberg, Derek L. Schuff, Ben L. Titzer, Michael Holman, Dan Gohman, Luke Wagner, Alon Zakai, and Jf Bastien. Bringing the web up to speed with webassembly. In *Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation - PLDI 2017*, page 185–200. ACM Press, 2017.

[54] Taylor Hardin, Josiah Hester, Ryan Scott, Jacob Sorber, Patrick Proctor, and David Kotz. Application memory isolation on ultra-low-power mcus. In *USENIX Annual Technical Conference*, 2018.

[55] Hc-sr501 pir motion detector. `https://www.mpja.com/download/31227sc.pdf`.

[56] Joseph M. Hellerstein, Jose Faleiro, Joseph E. Gonzalez, Johann Schleier-Smith, Vikram Sreekanti, Alexey Tumanov, and Chenggang Wu. Serverless computing: One step forward, two steps back. In *Conference on Innovative Data Systems Research (CIDR)*, 2019.

[57] Helm. `https://helm.sh/`.

[58] Scott Hendrickson, Stephen Sturdevant, Tyler Harter, Venkateshwaran Venkataramani, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau. Serverless computation with openlambda. In *HotCloud*, 2016.

[59] Galen C Hunt and James R Larus. Singularity: rethinking the software stack. *ACM SIGOPS Operating Systems Review*, 41(2):37–49, 2007.

[60] EunYoung Jeong, Shinae Woo, Muhammad Jamshed, Haewon Jeong, Sunghwan Ihm, Dongsu Han, and KyoungSoo Park. mtcp: A highly scalable user-level tcp stack for multicore systems. In *USENIX Symposium on Networked Systems Design and Implementation*, 2014.

[61] Zhipeng Jia and Emmett Witchel. Nightcore: efficient and scalable serverless computing for latency-sensitive, interactive microservices. In *Proceedings of the 26th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*, page 152–166. ACM, Apr 2021.

[62] Christopher Jonathan, Umar Farooq Minhas, James Hunter, Justin Levandoski, and Gor Nishanov. Exploiting coroutines to attack the" killer nanoseconds". *Proceedings of the VLDB Endowment*, 11(11):1702–1714, 2018.

[63] Json web token (jwt). https://tools.ietf.org/html/rfc7519.

[64] Robert Kaiser and Stephan Wagner. Evolution of the pikeos microkernel. In *First International Workshop on Microkernels for Embedded Systems*, volume 50, 2007.

[65] Hui Kang, Michael Le, and Shu Tao. Container and microservice driven design for cloud infrastructure devops. In *2016 IEEE International Conference on Cloud Engineering (IC2E)*, pages 202–211. IEEE, 2016.

[66] Karl-Bridge-Microsoft. Fibers - win32 apps. `https://docs.microsoft.com/en-us/windows/win32/procthread/fibers`.

[67] Hee-Su Kim and Kyoung-Ho Han. Implementation of pmp using arm processor. In *Proceedings of the KIEE Conference*, pages 2138–2139. The Korean Institute of Electrical Engineers, 2006.

[68] Avi Kivity, Yaniv Kamay, Dor Laor, Uri Lublin, and Anthony Liguori. kvm: the linux virtual machine monitor. In *Proceedings of the Linux symposium*, number 8, pages 225–230. Dttawa, Dntorio, Canada, 2007.

[69] Scott Klein. *IoT Solutions in Microsoft's Azure IoT Suite*. Springer, 2017.

[70] Adam Kozłowski and Janusz Sosnowski. Energy efficiency trade-off between duty-cycling and wake-up radio techniques in iot networks. *Wireless Personal Communications*, 107(4):1951–1971, 2019.

[71] Kubernetes. `https://kubernetes.io/`.

[72] Simon Kuenzer, Vlad-Andrei Bădoiu, Hugo Lefeuvre, Sharan Santhanam, Alexander Jung, Gaulthier Gain, Cyril Soldani, Costin Lupu, Ştefan Teodorescu, Costi Răducanu, et al. Unikraft: fast, specialized unikernels the easy way. In *Proceedings of the Sixteenth European Conference on Computer Systems*, pages 376–394, 2021.

[73] Agus Kurniawan. *Learning AWS IoT: Effectively manage connected devices on the AWS cloud using services such as AWS Greengrass, AWS button, predictive analytics and machine learning*. Packt Publishing Ltd, 2018.

[74] Amit Levy, Bradford Campbell, Branden Ghena, Daniel B. Giffin, Pat Pannuto, Prabal Dutta, and Philip Levis. Multiprogramming a 64kb computer safely and efficiently. In *Symposium on Operating Systems Principles*, 2017.

[75] Hongjun Li. Restful web service frameworks in java. In *2011 IEEE International Conference on Signal Processing, Communications and Computing (ICSPCC)*, pages 1–4. IEEE, 2011.

[76] QianQian Li, Sarada Prasad Gochhayat, Mauro Conti, and FangAi Liu. Energiot: A solution to improve network lifetime of iot devices. *Pervasive and Mobile Computing*, 42:124–133, 2017.

[77] libevent, 2021. `https://libevent.org/`.

[78] Welcome to the libuv documentation — libuv documentation, 2021. `http://docs.libuv.org/en/v1.x/`.

[79] Anil Madhavapeddy, Richard Mortier, Charalampos Rotsos, David Scott, Balraj Singh, Thomas Gazagnaire, Steven Smith, Steven Hand,

and Jon Crowcroft. Unikernels: Library operating systems for the cloud. *ACM SIGARCH Computer Architecture News*, 41(1), 2013.

[80] Vachirapol Mayalarp, Narisorn Limpaswadpaisarn, Thanachai Poombansao, and Somsak Kittipiyakul. Wireless mesh networking with xbee. In *2nd ECTI-Conference on Application Research and Development (ECTI-CARD 2010), Pattaya, Chonburi, Thailand*, pages 10–12, 2010.

[81] Maged Michael, Jose E Moreira, Doron Shiloach, and Robert W Wisniewski. Scale-up x scale-out: A case study using nutch/lucene. In *2007 IEEE International Parallel and Distributed Processing Symposium*, pages 1–8. IEEE, 2007.

[82] Mqtt - the standard for iot messaging. `https://mqtt.org/`.

[83] Iot edge — microsoft azure. `https://azure.microsoft.com/en-us/services/iot-edge/`.

[84] Irakli Nadareishvili, Ronnie Mitra, Matt McLarty, and Mike Amundsen. *Microservice architecture: aligning principles, practices, and culture.* " O'Reilly Media, Inc.", 2016.

[85] Node.js. Node.js, 2021. `https://nodejs.org/en/`.

[86] Pierre Olivier, Daniel Chiba, Stefan Lankes, Changwoo Min, and Binoy Ravindran. A binary-compatible unikernel. In *ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments*, page 59–73, 2019.

[87] OpenFaaS. `https://www.openfaas.com` [Online; accessed 1-Sep-2020].

[88] Ov5640 datasheet. `https://cdn.sparkfun.com/datasheets/Sensors/LightImaging/OV5640_datasheet.pdf`.

[89] Claus Pahl, Pooyan Jamshidi, and Olaf Zimmermann. Microservices and containers. *Software Engineering 2020*, 2020.

[90] Srinath Perera, Chathura Herath, Jaliya Ekanayake, Eran Chinthaka, Ajith Ranabahu, Deepal Jayasinghe, Sanjiva Weerawarana, and Glen Daniels. Axis2, middleware for next generation web services. In *2006 IEEE International Conference on Web Services (ICWS'06)*, pages 833–840. IEEE, 2006.

[91] George Prekas, Marios Kogias, and Edouard Bugnion. Zygos: Achieving low tail latency for microsecond-scale networked tasks. In *Proceedings of the 26th Symposium on Operating Systems Principles*, page 325–341. ACM, Oct 2017.

[92] Calton Pu, Henry Massalin, and John Ioannidis. The synthesis kernel. *Computing Systems*, 1(1):11–32, 1988.

[93] Rancher os, 2021. `https://rancher.com/`.

[94] Richard Rashid, Robert Baron, Alessandro Forin, David Golub, Michael Jones, Douglas Orr, and Richard Sanzi. Mach: a foundation for open systems (operating systems). In *Proceedings of the Second Workshop on Workstation Operating Systems*, pages 109–113. IEEE, 1989.

[95] Rusty Russell. virtio: towards a de-facto standard for virtual i/o devices. *ACM SIGOPS Operating Systems Review*, 42(5):95–103, Jul 2008.

[96] Jerome H Saltzer, David P Reed, and David D Clark. End-to-end arguments in system design. *ACM Transactions on Computer Systems (TOCS)*, 2(4):277–288, 1984.

[97] Julio Sanchez and Maria P Canton. *Microcontroller Programming: The Microchip PIC®*. CRC press, 2018.

[98] Bhisham Sharma and Mohammad S Obaidat. Comparative analysis of iot based products, technology and integration of iot with cloud computing. *IET Networks*, 9(2):43–47, 2020.

[99] Vindeep Singh and Sateesh K Peddoju. Container-based microservice architecture for cloud applications. In *2017 International Conference on Computing, Communication and Automation (ICCCA)*, pages 847–852. IEEE, 2017.

[100] Livio Soares. Flexsc: Flexible system call scheduling with exception-less system calls. In *USENIX Symposium on Operating Systems Design and Implementation*, 2010.

[101] Livio Soares. Exception-less system calls for event-driven servers. In *USENIX Annual Technical Conference*, 2011.

[102] Stack overflow developer survey 2021.

[103] Robert Stackowiak. Azure iot solutions overview. In *Azure Internet of Things Revealed*, pages 29–54. Springer, 2019.

[104] Terraform by hashicorp. `https://www.terraform.io/`.

[105] Tensorflow lite — ml for mobile and edge devices. `https://www.tensorflow.org/lite/`.

[106] Ubuntu server - for scale out workloads, 2021. `https://ubuntu.com/server`.

[107] Steven Vercauteren, Bill Lin, and Hugo De Man. A strategy for real-time kernel support in application-specific hw/sw embedded architectures. In *Proceedings of the 33rd annual Design Automation Conference*, pages 678–683, 1996.

[108] Virtualbox. `https://www.virtualbox.org/`.

[109] Windows server 2019 — microsoft, 2021. `https://www.microsoft.com/en-us/windows-server`.

[110] Rich Wolski, Chandra Krintz, Fatih Bakir, Gareth George, and Wei-Tsung Lin. Cspot: portable, multi-scale functions-as-a-service for iot. In *Proceedings of the 4th ACM/IEEE Symposium on Edge Computing*, page 236–249. ACM, Nov 2019.

[111] Explore the digi xbee ecosystem, 2021. `https://www.digi.com/xbee`.

[112] Perry Xiao. *Designing Embedded Systems and the Internet of Things (IoT) with the ARM mbed*. John Wiley & Sons, 2018.

[113] Piero Zappi, Elisabetta Farella, and Luca Benini. Tracking motion direction and distance with pyroelectric ir sensors. *IEEE Sensors Journal*, 10(9):1486–1494, 2010.