MSDBench: Understanding the Performance Impact of Isolation Domains on Microservice-based IoT Deployments

Sierra Wang, Fatih Bakir, Tyler Ekaireb, Jack Pearson, Chandra Krintz, and Rich Wolski

Computer Science Dept.
Univ. of California, Santa Barbara
sierrawang@ucsb.edu

Abstract. We present MSDBench – a set of benchmarks designed to illuminate the effects of deployment choices and operating system abstractions on microservices performance in IoT settings. The microservices architecture has emerged as a mainstay set of design principles for cloud-hosted, network-facing applications. Their utility as a design pattern for "The Internet of Things" (IoT) is less well understood.
We use MSDBench to show the performance impacts of different deployment choices and isolation domain assignments for Linux and Ambience, an experimental operating system specifically designed to support microservices for IoT. These results indicate that deployment choices can have a dramatic impact on microservices performance, and thus, MSDBench is a useful tool for developers and researchers in this space.

1 Introduction

As web service technologies have improved in performance and usability, the design of web/cloud service applications (often user-facing web venues) has evolved to make use of internal purpose-built web services as composable application components. This approach is often termed a *microservices* design or architecture, and the internal services themselves are called microservices.

Software architects find microservices attractive from a software engineering perspective because they promote software reuse [12,13,33], they naturally admit heterogeneous software languages and runtimes [12,16,35], and they improve the performance of software quality assurance mechanisms such as unit testing [12, 40]. They also enhance software robustness and facilitate distributed placement flexibility by incorporating modularity and service isolation into the internal design of the overall application [31, 33, 44].

The cost associated with these benefits, compared with monolithic application design in which the internal functionality is not a decomposition of microservices, is execution performance. Performance, in this context, refers (i) to the latency a user of the application observes when making individual requests to the application, (ii) to the computational and storage capacity that is necessary to support the application's functionality, and (iii) to the communication overhead of sending and processing network requests to/from other microservices and across isolation boundaries. As such, microservice designs tend to in-

crease user-experienced request latency and application capacity requirements compared with their monolithic counterparts [16, 41].

These costs are especially acute for applications designed to implement the "Internet of Things" (IoT). Microservices, as a fundamental design principle, is endemic in large-scale application hosting (e.g. cloud computing) contexts where web service technologies are well supported both from a performance and also a security perspective. Furthermore, many IoT applications use cloud-based services for scalable analysis, visualization, and user interactions. Thus, microservices have become a key architectural approach to building IoT applications due largely to the facility with which they can be deployed in the cloud.

However, the latency and capacity requirements for IoT applications differ considerably from other web service applications (e.g. e-commerce, social networking, web-content delivery, etc.). IoT applications almost always include data acquisition deadlines that arise from sensor duty cycles (e.g. a sensor produces a measurement with a periodicity measured in milliseconds to seconds) and sometimes include near real-time response deadlines (e.g. to operate an appliance as an automated response to analysis of sensor data). Thus, a careful understanding of application response latency is important to IoT application design, particularly when the design is microservice based.

For these reasons, IoT deployments are increasingly incorporating "edge" computing capabilities that augment cloud-based processing. By processing IoT data *in situ*, before traversing a long-haul network to a cloud, IoT applications can reduce response latency, decrease the needed long-haul bandwidth (e.g., by performing data aggregations at the edge), and improve scale. The edge resources, however, are typically not full-scale cloud resources but smaller, more resource restricted, single board computers or microcontrollers that can be inexpensively deployed near the "Things" in the Internet of Things. Therefore, the capacity requirements of microservices located at the edge must be considered.

The task of determining what each microservice does in an application (i.e., the service decomposition "boundaries") is typically a manual process that falls to the software architect. As such, the choices are design-time choices, and not deployment-time choices. In a cloud context, where computational, storage, network, and security topologies can be understood to be relatively static, design-time decomposition is effective. In an IoT context, *the same application may be deployed to many different infrastructures*, each with its own unique set of performance and security characteristics. Thus, it is critical for the designer to be able to anticipate the costs associated with service decomposition decisions for different IoT deployments.

To enable this, we present MSDBench – a set of microservice benchmarks specifically designed to capture the relevant performance characteristics for IoT applications. Our work is distinct from previous microservice benchmarking efforts [16, 26, 41] in that we focus specifically on the impact of using different isolation alternatives and placement decisions that consider devices, "the edge", and the cloud as possible execution sites. In particular, our benchmarks do not assume that the edge and device resources can run a common commodity oper-

ating system (e.g. Linux or Windows) since IoT deployments often incorporate devices requiring lightweight or real-time operating systems.

The benchmarks, described in Section 3, comprise a set of microbenchmarks that exercise cross-domain functionality and an end-to-end application benchmark based on the popular publish-subscribe IoT design pattern. To illustrate the diagnostic power of the benchmark suite, in Section 4, we compare the performance of the benchmarks using Linux as a host operating system to Ambience, an operating system specifically designed to support IoT microservices [3]. These results show the importance of different deployment decisions with respect to isolation domains and network connectivity. We also show (using Ambience) how the choice of isolation domain decomposition affects performance on devices that include only microcontrollers.

2 Related Work

Microservices is an application architecture that composes loosely coupled components that communicate using inter-/remote procedure calls or other REST APIs. Their loose coupling facilitates fault tolerance, scaling, and automatic orchestration [6, 11, 29] which enables independent development and enhanced software engineering benefits. As a result, microservices are widely used for development of web/cloud applications [8, 14, 29, 32], and more recently for applications deployed across the cloud-edge continuum [24, 28].

Given this widespread use, multiple benchmarking systems have emerged to help developers understand and reason about the performance implications of Linux-based microservices applications. DeathStarBench is a suite designed to explore how well the cloud system stack supports microservices, from the hardware to the application implementation [16]. The DeathStarBench applications were designed to be representative of large, language- and library-heterogeneous, end to end microservices applications that run primarily in the cloud. Their work compares microservices applications against monolithic applications and analyzes how well the cloud platform supports each application type.

Several benchmarking suites analyze the resource demands of specific application types, including scale out workloads, latency critical applications, and online data intensive microservices applications [15, 20, 25, 26, 34, 41, 43, 45, 48]. Ppbench examines how different languages, containerization, and a software defined networking affect microservices performance [26]. Other work explores techniques for benchmarking microservices and how to use this information to inform deployment decisions [1, 17, 18, 21, 23, 47]. While several other benchmarking efforts describe IoT benchmarking suites for evaluating IoT architectures, IoT Gateway systems, IoT hardware devices, IoT database systems, IoT sensor and analytics platforms, and distributed stream processing platforms [5, 19, 27, 30, 36, 38], these latter suites are not designed for or tailored to the microservices architecture.

MSDBench differs from this prior work in both its focus and its content. It is unique in that it targets how runtime systems support microservices applications with regards to deployment options common to IoT settings (placement,

isolation, cross-service optimization). To show its utility, we use the suite to evaluate and empirically compare the impact of different operating systems, RPC frameworks, hardware, and isolation domains across deployments that span the IoT cloud-edge continuum.

3 Benchmark Design

Microservices are useful for IoT because

- Microservices can be sized/decomposed to match the heterogeneous set of computing capacities in a target IoT deployment (e.g. one consisting of a resource constrained microcontroller, capacity limited edge device or edge cloud virtual machine (VM), or resource rich VM in a public or private cloud interconnected by low-power radio networks, WiFi, and wired networking).
- Microservices can be assigned to separate isolation domains (e.g. process- or service-level virtualization technologies) to implement site-specific security policies and to improve fault isolation.
- Microservices are decoupled from the operating system and other microservices, enabling independent development, distributed deployment, and use of a wide range of isolation options (e.g. IPC/RPC communication, process virtualization, or system virtualization); and
- Operating and build systems that are microservices-aware can exploit static information associated with deployments (e.g., co-location, service dependencies) to automatically optimize away various overheads associated with isolation and decoupling [3, 22].

These features facilitate portability, rapid development, improved performance, and low maintenance. Moreover, this design enables horizontal scaling with little involvement from programmers, since a particular dependency of a service can be transparently replicated.

Developers must also face a number of new challenges when using microservices in distributed and heterogeneous IoT settings. In particular, service proximity can have a significant impact on overall application performance. For example, co-locating microservices on a single node (e.g. as a Kubernetes "pod" [29]) can enhance the inter-service communication, but also introduce security and/or fault isolation vulnerabilities. Further, in an IoT context where some of the services implement data acquisition, moving a microservice away from the data acquisition site to improve its inter-service messaging performance may degrade data acquisition latency. Moreover, these performance-impacting factors can be *deployment specific* meaning that the developer must code the microservices without knowing how they will ultimately be deployed.

To address these challenges, we have developed *MSDBench*, a pair of benchmark suites for exploring the performance implications of different operating system, isolation, and placement alternatives for microservices applications deployed across the multi-tier IoT resources (microcontrollers, edge systems, and public/private clouds). MSDBench is unique in that it facilitates the study of

different operating systems, devices, system-level virtualization, and isolation domains in combination. As described in the previous section, existing microservice benchmarking approaches [16, 26, 41, 43] focus on resource-rich, relatively homogeneous cloud deployments and devices that run Linux. By addressing this gap, MSDBench enables developers to reason about the performance of emerging IoT deployments end-to-end and to interrogate the performance impact from using different isolation domains and operating systems.

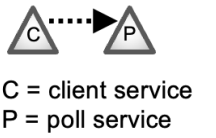

Fig. 1: MSDBench microbenchmark structure.

MSDBench consists of a microbenchmark suite and end-to-end application suite. Both suites separate the client from the rest of the application (which we refer to as the server-side services) to allow for separate performance analysis (end-to-end versus server-side). We write microservices in C++ for device portability and implement them to be as efficient as possible. The microbenchmark suite consists of an application with two microservices depicted as triangles in Figure 1. The client service makes requests to the poll service (dashed arrow in the figure). The poll service simply returns. The call benchmark includes a request payload, the size of which is parameterizable. This suite enables us to understand the overhead associated with calls and returns (local or remote) and any use of payload serialization.

The end-to-end application suite is a "Best Effort Pub/Sub (BEPS)" application consisting of six unique microservices. We provide a graphic of BEPS in Figure 2; The microservices are triangles, and their dependencies are arrows. The dashed arrow is used by the client for requests to the BEPS entry point. The client service (C) makes requests to the server-side services which comprise the suite's benchmarks. The server-side services consist of a load balancer microservice (LB), 1+ workers (W), a user database service (DB), a payload database service (PDB), and a read endpoint (R). The client makes `create_user`, `subscribe`, `publish`, `unsubscribe`, and `delete_user` requests to the load balancer. Each request type benchmarks a different aggregate functionality from the microservice mesh. The load balancer distributes requests using round-robin among the workers. Each worker uses the payload database and the user database to service each request and publishes updates to the read endpoint as necessary. The number of workers is parameterizable; we use five in the evaluation herein.

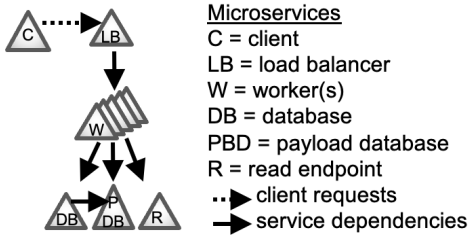

Fig. 2: MSDBench end-to-end application structure.

We designed BEPS to represent several common microservices design patterns [2, 9, 46]. The load balancer is an "API Gateway (or Proxy)" as it provides a uniform interface to make requests to different services. Each worker is an "Aggregator" since it combines information from both databases to update a user's feed in the read endpoint.

BEPS employs the "Data Sharing" pattern since all of the workers share the same two database instances. When the OS supports asynchrony (e.g. Ambience does so via coroutines), all communication implements the "Asynchronous messaging" pattern.

In this paper, we use MSDBench with Linux and Ambience [3]; the latter is an experimental operating system specifically designed for IoT microservices. The benchmark suites are coded as generically as possible to facilitate their porting to other operating systems and software ecosystems. We choose these two examples to illustrate how the benchmarks allow a developer to assess the trade-off between performance and technology risk. The MSDBench Linux benchmarks use Thrift [39, 42] for RPC and argument serialization. They use Docker containers [10] for process-level isolation. MSDBench can use KVM VMs or physical hosts for these deployments.

Ambience uses a "group" abstraction to isolate and co-locate microservices. Microservices in the same group share an address space, are not isolated from each other, and can be optimized together. Microservices in different groups are isolated via protected address space regions. Ambience uses lidl, an interface description language (IDL), to describe inter-service communication. `lidl` transparently specializes these interfaces as direct function calls, zero-copy shared memory for calls across address spaces, or serialization for cross-machine calls. Ambience is also more resource-scale independent than Linux. It is possible to run Ambience on resource-restricted microcontrollers that do not have sufficient functionality (e.g. an MMU) or resource capacity to run Linux. At the same time, Ambience runs natively on the x86 and ARM architectures and on the KVM hypervisor. Thus, it is possible to run Ambience as a single operating system on microcontrollers, single board computers at the edge, and cloud-based VMs in a tiered IoT deployment. It is, however, highly experimental and supports a unique and potentially unfamiliar set of operating system abstractions specifically for optimized microservices.

MSDBench is also unique in that it decouples the mapping of microservices to isolation domain from the mapping of isolation domains to hosts. We distinguish the two because developers and operators typically have control over the former (isolation domain assignment, i.e. containerization). The infrastructure provider (e.g. cloud vendor) may demand an additional level of isolation to facilitate resource apportionment, decommissioning, and sharing of resources. Thus, MSDBench allows different combinations of these two mappings to be explored empirically. We refer to *isolation domains* when discussing operating-system implemented protection domains and *deployments* when discussing the assignment of microservices to isolation domains and the assignment of isolation domains to hosts. That is, a developer or application operator may decide on the assignment of microservices to isolation domains, and those isolation domains may either be implemented natively or placed in infrastructure-provided containers.

Mapping Microservices to Isolation Domains – We depict the three isolation domain configurations (*co-located*, *tiered*, and *isolated*) that we consider in our evaluation using the BEPS suite in Figure 3 (the microbenchmark suite is sim-

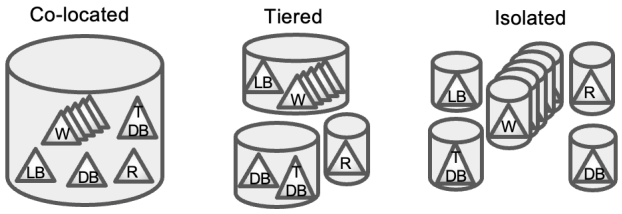

Fig. 3: BEPS Mapping of Microservices to Isolation Domains. An isolation domain provides process-level isolation (cylinders) for microservices (triangles). The letters in each triangle identify the BEPS microservice. For a Linux OS, the isolation domain is a Linux or Docker container. Ambience uses lightweight groups. MSDBench enables empirical evaluation and comparison for alternative isolation domain configurations, including those shown here: all co-located, tiered isolation (grouped), and full isolation.

ilar). An isolation domain for the Linux OS is a Linux or Docker container; for Ambience, it is an Ambience group. In the co-located configuration, the load balancer, workers, databases, and read endpoint are all in the same isolation domain. In the tiered configuration, the load balancer and the workers are in an isolation domain, the databases are in an isolation domain, and the read endpoint is in an isolation domain. In the isolated configuration, every microservice is in its own isolation domain.

Deployments: Mapping Isolation Domains to Hosts – We depict the five deployments for the BEPS suite that we consider in our evaluation in Figure 4 (we use the same deployments for the microbenchmark suite). For each deployment, we will evaluate the three isolation domain configurations above (co-located, tiered, and isolated). We represent these in the figures as a cloud icon marked "server-side." In deployment 1, we place the client within the load balancer's (LB's) isolation domain. All isolation domains in this deployment are co-located within a single VM on the same physical host. In deployment 2, we place the client in its own isolation domain, and co-locate that domain within the same VM and and on the same physical host. In deployment 3, we modify deployment 2 so that the client and its isolation domain are in a separate VM but on the same physical host as the server-side VM. In deployment 4, we modify deployment 3 so that the client VM is on a separate physical host from that of the server-side microservices. Deployment 5 is the same as deployment 4 only the client service is executed directly on the physical host instead of in a VM.

MSDBench Configuration – MSDBench configuration uses a combination of scripting and deployment manifests to implement its benchmark deployments. Deployment configuration however, is currently manual (we are working on automation as part of future work). Linux VMs can be configured and deployed using any one of the many configuration management tools to automate server provisioning (e.g. puppet [37], ansible [4], chef [7], etc.). Moreover, Ambience

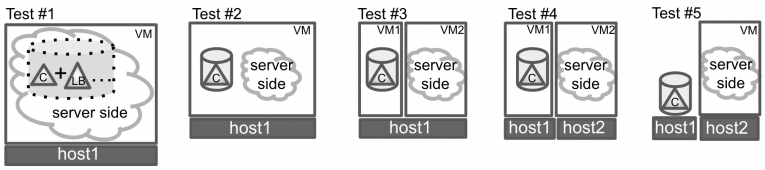

Fig. 4: BEPS Mapping of Isolation Domains to Hosts (MSDBench deployment alternatives). MSDBench enables evaluation of different deployment options by mapping isolation domain configurations to infrastructure options (e.g. sensors, edge, cloud). We consider 5 common deployments (shown here) in our empirical evaluation. The server side configurations that we consider are shown in Fig. 3.

has deployment support based on deployment manifests in which service interfaces and implementations are specified. It combines these with service manifests which specify service dependencies and hosts, to create an application deployment. We use qemu to instantiate virtual machines for both Linux and Ambience VM on KVM systems. For the microcontroller, we manually flash the devices with the Ambience images once they have been built. MSDBench leverages these tools for basic benchmark deployment scripting.

4 Empirical Evaluation

To generate informative results with minimal external noise, we run our experiments in a controlled environment. Our IoT setting consists of microcontrollers, edge devices, and a private cloud. In this study, we use the Nordic Semiconductor nRF52840 which has a 64MHz Arm Cortex-M4 CPI with FPU. It has 1MB of flash memory plus 256KB of RAM. It communicates via Bluetooth 5.3 and zigbee (IEEE 802.15.4). The multi-host microcontroller deployments use zigbee for communication. Our edge and cloud servers are Intel NUC8i7HNK systems (NUCs) with 8 Core i7 CPUs (3.1 GHz), 32GB of Memory, and 1TB of disk. The multi-host edge/cloud experiments use a dedicated, isolated Ethernet network between hosts for communication. All devices run Ambience v1.0 [3] which runs on all devices that we consider herein. All devices except the microcontrollers are capable of supporting Linux. We use Fedora 35 and Fedora 36, KVM for virtualization, and Thrift for RPC on the Linux systems. Ambience integrates virtualization internally (running directly on KVM or within a Linux process/container) and uses lidl for IPC/RPC.

We refer to the deployments that use the Intel NUCs as "edge/cloud" deployments, and the deployments that use the ARM devices as the "microcontroller" deployments in the evaluation that follows. Note that the nRF52840 microcontroller is not a Linux-capable single board computer (e.g. a device similar to a Raspberry Pi which also uses an ARM processor) but is a severely resource-restricted embedded device without an MMU.

For the microbenchmark edge/cloud deployments, the client makes 10,000 requests to the poll service per experiment. Our experiments evaluate different

request payload sizes (0, 512, 1024, 2048, 4096, and 8192 bytes). Each request returns a 64-bit response payload. For the microcontroller deployments (due to resource constraints), the client makes 100 requests and we experiment with request payloads of 0 and 64 bytes.

For the edge/cloud deployments of the end-to-end benchmarks, we use MSDBench to measure the round trip request latency for BEPS by timing 10,000 `create_user`, 10,000 `subscribe`, 10,000 `publish`, 10,000 `unsubscribe`, and 10,000 `delete_user` requests from the client. The BEPS user names are 10 characters and the messages are 280 bytes. For the microcontroller deployments, we perform 10 requests each and use user names and messages of length 5 and 20 bytes, respectively. MSDBench can be used to measure both the internal (server-side) time and the end-to-end time experienced by the clients. We report the end-to-end times experienced by the client herein.

We use these benchmark suites to evaluate five deployments and three isolation domain configurations described in the previous section for our edge/cloud experiments (Section 3). We consider deployments 1, 2, and 4 and isolation domains co-located and tiered for the microcontroller deployments. All results, unless otherwise specified (e.g. for the throughput study), are in microseconds.

4.1 Microbenchmark Results

The MSDBench microbenchmark suite is useful for determining the performance impact associated with the microservice interface boundaries. Microservices typically communicate with each other through remote procedure call (RPC) or remote invocation mechanisms across their exported interfaces. The benchmark uses a single poll service that accepts a request via RPC and returns a timestamp to enable measurement of the RPC call and return performance. To evaluate the utility of this suite, we compare the overhead of RPC calls using different request payloads. Note that because RPC mechanisms are language level abstractions, they often convey typed data which must be serialized for transfer and then deserialized upon receipt. The benchmark includes serialization overhead.

For all experiments, the client and poll services are on the same machine and VM. Figure 5 and Figure 6 show the average inter-service latency in microseconds for different payload sizes when deployed on the edge/cloud; Figure 7 similarly shows the average inter-service latency when deployed on the microcontroller. We use MSDBench to explore the performance differences of the no-isolation (co-located) and fully isolated isolation domains.

For the edge/cloud study, co-location reveals the impact of any optimization performed by the OS and/or microservices hosting framework. Note that Ambience uses compile and link time optimizations to automatically remove the messaging and serialization/deserialization code when microservices are co-located. In Linux, microservices use the same serialization and messaging code regardless of co-location. However, when co-located, Linux uses a "fast-path" for local network communication.

Ambience co-located thus achieves 73x better performance than fully isolated versus 1.3x for Linux. Ambience's group abstraction enables 6x better call

performance (isolated configuration) compared to Linux because it is able to optimize across groups (using zero copy shared memory), a feature not available for Linux containers. Note that Ambience performs similarly regardless of the amount of data passed. This is because Ambience requires a deployment manifest that shows the location of microservices in a deployment so it can "compile-away" serialization and data copies when microservices share an address space. Each system runs an image that is compiled using the manifest and relocation of microservices requires new images to be created and deployed. For Linux, serialization (via Thrift) and messaging cause the microservices to slow as the payload size increases. However, Linux microservices do not need to be recompiled when they are moved between compatible architectures, and they may not need to be relinked (depending on the degree of software version compatibility between potential execution sites).

For the microcontroller, Ambience co-located outperforms isolated by 20x (versus 73x for edge/cloud). This is due to the slower clock rate (compared to the x86-based NUC) and the limited resources of the device. As noted previously, the microcontroller does not support Linux so we do not report results for Linux.

Microbenchmark Results

The following graphs show the average time for the client service to call the poll service (y-axis), with different payloads (x-axis), under different deployment configurations.

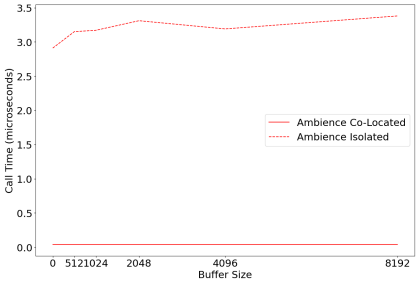

Fig. 5: Latency when isolating and co-locating Ambience services for edge/cloud.

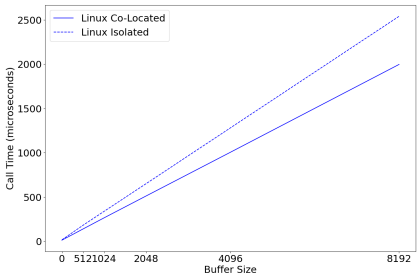

Fig. 6: Latency when isolating and co-locating Linux services for edge/cloud.

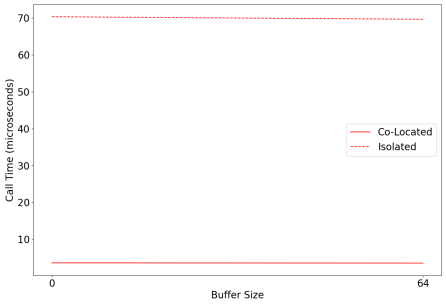

Fig. 7: Latency when isolating and co-locating the Ambience services for microcontrollers.

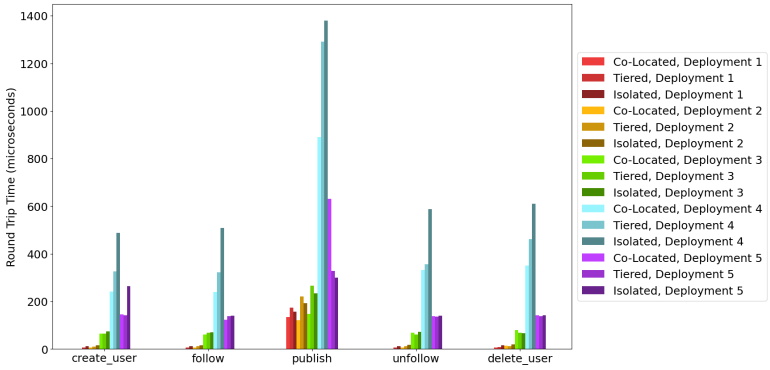

Fig. 8: End-to-End Benchmarking Results for Ambience on the Edge/Cloud deployments. The graph shows average round trip latency in microseconds for each request type, for each mapping of microservices to isolation domains (Co-Located, Tiered, and Isolated, see Fig. 3) and mapping of isolation domains to hosts (Deployments 1-5, see Fig. 4).

4.2 End to End Benchmark Results

We next use MSDBench to investigate a number of deployment related research questions using BEPS, the end-to-end benchmark suite. For these experiments, we consider the five deployments in Figure 4 and the three isolation (ISO) domain configurations (co-located, tiered, and isolated) shown in Figure 3. We benchmark both Ambience (Amb) and Linux (Lin) and report latency in microseconds observed by the client in terms of the average and standard deviation across 10,000 requests to each benchmark service function. The service functions are `create_user`, `subscribe`, `publish`, `unsubscribe`, and `delete_user`. Figure 8 shows the round trip times for each service function for all deployment configurations of Ambience. Figure 9 shows the corresponding results for Linux.

The data provides a number of different insights. First, the suite includes benchmarks with different resource requirements. For example, `publish` requires more server-side processing than the others, `unsubscribe` and `delete_user` are impacted by network overhead (e.g. for cross-VM and machine deployments). As a result, `publish` takes 11-15x longer on average than `create_user` on Linux when within the same VM but this difference is reduced to 50-70% when the client is placed on a different machine (because the networking and isolation overhead plays a much larger role). These differences enable developers to make informed decisions about workload mix, service replication, and placement.

Next, the data shows the potential for performance optimization for co-located microservices. In every case, both Linux and Ambience show significantly better performance for co-located versus tiered (approximately 30-70% slower for Ambience, and 20% slower for Linux) or isolated (approximately 20-70% slower for Ambience, and 10-40% slower for Linux). Third, it enables us to understand the performance differences between the use of a general and special purpose operating system. On average, Ambience is at least an order of magni-

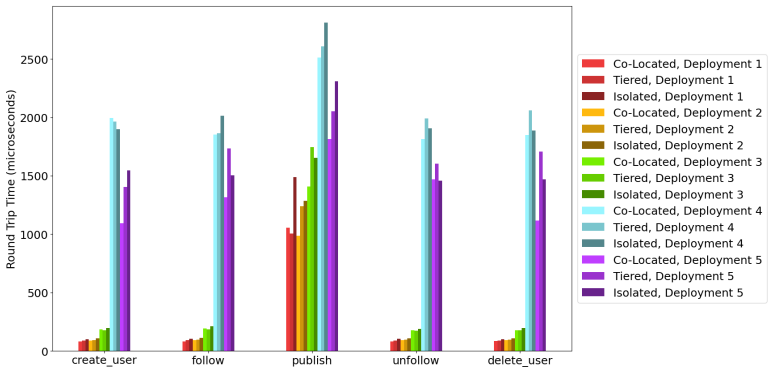

Fig. 9: End-to-End Benchmarking Results for Linux on the Edge/Cloud deployments. The graph shows average round trip latency in microseconds for each request type, for each mapping of microservices to isolation domains (Co-Located, Tiered, and Isolated, see Fig. 3) and mapping of isolation domains to hosts (Deployments 1-5, see Fig. 4).

tude faster than Linux for all equivalent deployments, and the slowest Ambience experiment across deployments 4 and 5 (which traverse a network connection) is faster than the fastest Linux experiment in deployments 4 and 5 *across all experiments*, regardless of isolation domain assignment and service request type. We were surprised by these results, given the relatively highly optimized nature of the Linux networking stack and the maturity of its isolation implementations.

The differences per deployment are also interesting. Deployment 1 enables us to remove client interaction. Although this would not be used in an actual deployment (clients are typically separated and isolated from the server-side services for fault resiliency), it allows us (as developers) to focus on the server side performance of our deployments. This deployment with co-located isolation is the configuration with the best possible performance because maximal optimization is possible and minimal overhead is introduced to provide limited isolation. The data across deployments shows that a large portion of the performance overhead end-to-end comes from separating the client from the server side.

Deployment 2 represents a more realistic edge case in which the microservices are co-located on the same device with the client isolated using only process-level virtualization (i.e. Linux containers or Ambience groups) and the server-side microservices isolated in various ways (all co-located, all isolated, or some combination (e.g. tiered)). Using deployment 2 as a baseline, Linux deployment 3 (isolating the client in its own VM) is 14-16x slower, and Linux deployment 4 (placing client and VM on a different host) is 65-68x slower. When we place the client on a different host without a VM (deployment 5), the end-to-end performance is only 45-56x slower. This latter result represents the overhead of system level virtualization (e.g. cloud use). For Ambience, deployment 3 and deployment 4 are 3-4x slower and 13-23x slower than deployment 2, respectively. The Ambience performance is also impacted by placing the client in a VM –

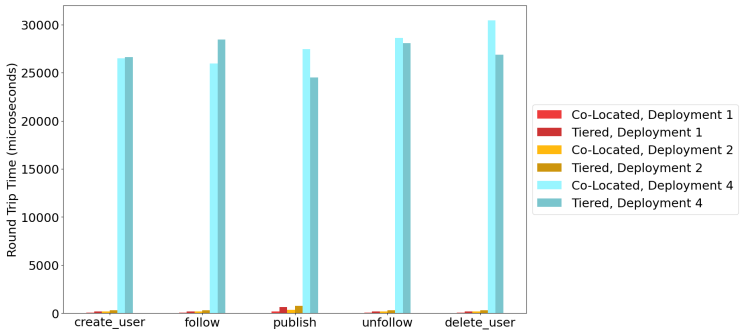

Fig. 10: End-to-End Benchmarking Results for Ambience on the microcontroller deployments. The graph shows round trip latency in microseconds for each mapping of microservices to isolation domains and a subset of the mappings of isolation domains to hosts (deployments 1, 2, and 4, see Fig. 4).

deployment 5 is only 6-8x slower when the client is placed on bare metal vs 13-23x slower in an VM (deployment 4).

Figure 10 shows the end-to-end results for running Ambience on microcontrollers. In these experiments, we use deployments 1, 2, and 4 and the co-located and tiered isolation domains. The trends are similar, however, the differences are less stark due to the slower clock speed and severe resource constraints of the devices. Separating out the client (deployment 2 vs 1) introduces about 2x overhead across benchmarks. The performance for co-located and tiered is similar when the client is separated. Using deployment 2 as a baseline, the total average time across all benchmarks is approximately 134x slower for deployment 4 when co-located. Another interesting aspect revealed by this benchmark suite is the relative performance between microcontroller and edge/cloud deployments. For example, due to the limited capability of the microcontrollers, microcontroller deployment 1 exhibits performance that is similar to that of edge/cloud deployment 3 (which adds VM-level isolation to the client) for co-located isolation.

4.3 Throughput Results

We next use MSDBench on the edge/cloud deployment 5 to test how the isolation domain configuration and platform supports different client workloads. In particular, we show how MSDBench can be used to support capacity planning for hosts in an IoT deployment. Capacity planning enables developers and deployment administrators to understand what the hosts in a deployment are capable of in terms of servicing microservice load.

To enable this, we use MSDBench to measure the performance of concurrent requests issued by multiple client processes simultaneously. For this study, we used an MSDBench client that is written in Python; Python simplifies the scripting of benchmark harnesses but adds considerable client-side latency (which is why we did not use it for the microbenchmark and end-to-end experiments). Our

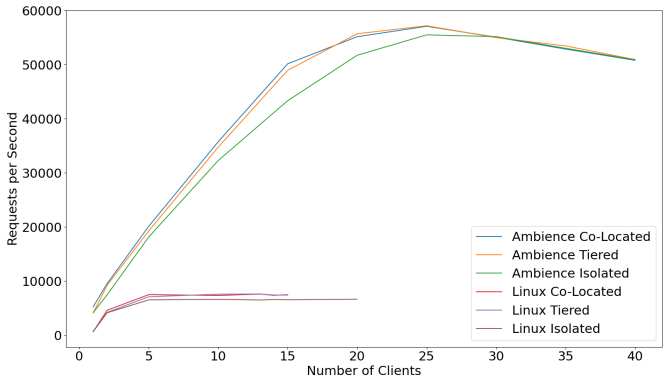

Fig. 11: MSDBench Throughput Experiments. This benchmark uses the edge/cloud deployment 5 to evaluate and compare three isolation domains co-located, tiered, and isolated for Ambience and Linux. The graphs show the average number of requests per second as the number of clients increases. The Linux system was unable to run workloads with more 15-20 clients for any configuration. Ambience achieves its peak throughput at 25 clients, Linux does so at 13. Such studies are a key component of capacity planning for IoT deployments.

Python client is the same for Linux and Ambience except in its use of Thrift versus lidl for the respective RPC implementations. We invoke the clients concurrently. Each client "warms" the application by executing 50 `create_user` and `delete_user` requests each. It then times 105,000 requests of each type (210,000 total requests), then computes and outputs the throughput number. We repeat the experiment for an increasing number of clients until the number of requests per second stops increasing, indicating the host's saturation point for this benchmark. The resulting throughput "curve" indicates how microservices consume capacity as a function of offered request load for a given mix of service requests. A similar throughput curve can be generated for any individual or combination of the MSDBench microservices and target device.

Figure 11 shows the throughput in requests per second (rps) for each OS and isolation domain configuration as the number of clients increases. We use this benchmark to compare the co-located, tiered, and isolated configurations and the two OS's we consider (Ambience and Linux). The Linux system consistently crashed (we were unable to determine why) for client counts higher than 15 for co-located and tiered, and 20 for isolated. The throughput of the Linux system achieves a maximum throughput of 7587 rps with 13 clients for co-located, 7594 rps with 13 clients for tiered, and 6613 rps with 10 clients for isolated. At 5 clients, Ambience achieves 2.7x more throughput than Linux. Ambience saturates the capacity of the server-side host at 25 clients achieving a maximum throughput of 57083 rps for co-located, 57193 rps for tiered, and 55508 rps for isolated.

Note that all of the throughput experiments are for deployment 5, where the clients are executed on a separate host and communicate with the microservices over a 1GB dedicated Ethernet network. Surprisingly, the throughput rate for Linux is not network dominated (it may be for Ambience, but we were unable to determine that it was conclusively). Indeed, the Linux networking stack is highly optimized compared to the nascent networking stack included in the Ambience runtime. Further, because requests are traversing the network, the Ambience requests include all serialization/deserialization and messaging overheads (the Ambience image compiler could not optimize these away). We expected that both Ambience and Linux would achieve the same saturation throughput (perhaps for different client counts) with the network as the performance bottleneck. This result illustrates both the impact of OS abstractions other than the networking abstractions on microservices as well as the relative capacity consumption of the two hosting operating systems.

5 Conclusion

We present MSDBench, a benchmarking suite for exploring the possibilities of deploying microservices in an IoT setting and understanding how deployment decisions impact microservices application performance. In our analysis, we study the effect of isolation domains, the assignment of isolation domains to hosts, operating systems abstractions, RPC Frameworks, and device types, revealing the strengths and weaknesses of each. We also investigate the performance associated with running microservices on resource-restricted devices (such as microcontrollers) that cannot host commodity service operating systems (e.g. Linux). The results indicate that the various deployment and operating system choices can have a dramatic effect of eventual application performance. This work enables us to understand how IoT technology supports microservices in terms of what is possible and what is optimal, informing future research and development on using microservices in an IoT setting.

References

1. Aderaldo, C.M., Mendonça, N.C., Pahl, C., Jamshidi, P.: Benchmark requirements for microservices architecture research. In: 2017 IEEE/ACM 1st International Workshop on Establishing the Community-Wide Infrastructure for Architecture-Based Software Engineering (ECASE). pp. 8–13. IEEE (2017)
2. Akbulut, A., Perros, H.G.: Performance analysis of microservice design patterns. In: IEEE Internet Computing, vol. 23, no. 6. pp. 19–27 (2019)
3. Ambience Microservices OS (May 2022), `https://github.com/MAYHEM-Lab/ambience` [Online; accessed 20-May-2022]
4. Ansible Configuration Management. `https://www.ansible.com`, [Online; accessed 20-July-2022]
5. Arlitt, M., Marwah, M., Bellala, G., Shah, A., Healey, J., Vandiver, B.: Iotabench: an internet of things analytics benchmark. In: Proceedings of the 6th ACM/SPEC International Conference on Performance Engineering. pp. 133–144 (2015)

6. AWS Elastic Container Service. https://aws.amazon.com/ecs/, [Online; accessed 20-July-2022]
7. Chef Configuration Management. https://www.chef.io, [Online; accessed 20-July-2022]
8. Decomposing Twitter: Adventures in Service-Oriented Architecture. https://www.slideshare.net/InfoQ/decomposing-twitter-adventures-in-serviceoriented-architecture, [Online; accessed 19-July-2022]
9. Everything You Need To Know About Microservices Design Patterns. https://www.edureka.co/blog/microservices-design-patterns, [Online; accessed 20-July-2022]
10. Docker, https://www.docker.com [Online; accessed 12-Sep-2017]
11. Docker Swarm. https://docs.docker.com/engine/swarm/, [Online; accessed 20-July-2022]
12. Dragoni, N., Giallorenzo, S., Lafuente, A.L., Mazzara, M., Montesi, F., Mustafin, R., Safina, L.: Microservices: yesterday, today, and tomorrow. Present and ulterior software engineering pp. 195–216 (2017)
13. Dragoni, N., Lanese, I., Larsen, S.T., Mazzara, M., Mustafin, R., Safina, L.: Microservices: How to make your application scale. In: International Andrei Ershov Memorial Conference on Perspectives of System Informatics. pp. 95–104. Springer (2017)
14. The Evolution of Microservices. https://www.slideshare.net/adriancockcroft/evolution-of-microservices-craft-conference, [Online; accessed 19-July-2022]
15. Ferdman, M., Adileh, A., Kocberber, O., Volos, S., Alisafaee, M., Jevdjic, D., Kaynak, C., Popescu, A.D., Ailamaki, A., Falsafi, B.: Clearing the clouds: a study of emerging scale-out workloads on modern hardware. Acm sigplan notices **47**(4), 37–48 (2012)
16. Gan, Y., Zhang, Y., Cheng, D., Shetty, A., Rathi, P., Katarki, N., Bruno, A., Hu, J., Ritchken, B., Jackson, B., et al.: An open-source benchmark suite for microservices and their hardware-software implications for cloud & edge systems. In: International Conference on Architectural Support for Programming Languages and Operating Systems (2019)
17. Grambow, M., Meusel, L., Wittern, E., Bermbach, D.: Benchmarking microservice performance: a pattern-based approach. In: Proceedings of the 35th Annual ACM Symposium on Applied Computing. pp. 232–241 (2020)
18. Grambow, M., Wittern, E., Bermbach, D.: Benchmarking the performance of microservice applications. ACM SIGAPP Applied Computing Review **20**(3), 20–34 (2020)
19. Gupta, P., Carey, M.J., Mehrotra, S., Yus, o.: Smartbench: a benchmark for data management in smart spaces. Proceedings of the VLDB Endowment **13**(12), 1807–1820 (2020)
20. Hauswald, J., Laurenzano, M.A., Zhang, Y., Li, C., Rovinski, A., Khurana, A., Dreslinski, R.G., Mudge, T., Petrucci, V., Tang, L., Mars, J.: Sirius: An open end-to-end voice and vision personal assistant and its implications for future warehouse scale computers. In: International Conference on Architectural Support for Programming Languages and Operating Systems. p. 223–238 (2015)
21. Henning, S., Hasselbring, W.: Theodolite: Scalability benchmarking of distributed stream processing engines in microservice architectures. Big Data Research **25**, 100209 (2021)

22. Jia, Z., Witchel, E.: Nightcore: efficient and scalable serverless computing for latency-sensitive, interactive microservices. In: International Conference on Architectural Support for Programming Languages and Operating Systems. pp. 152–166 (2021)
23. Jindal, A., Podolskiy, V., Gerndt, M.: Performance modeling for cloud microservice applications. In: Proceedings of the 2019 ACM/SPEC International Conference on Performance Engineering. pp. 25–32 (2019)
24. K3S. `https://k3s.io`, [Online; accessed 19-July-2022]
25. Kasture, H., Sanchez, D.: Tailbench: A benchmark suite and evaluation methodology for latency-critical applications. In: International Symposium on Workload Characterization (2016)
26. Kratzke, N., Quint, P.C.: Investigation of impacts on network performance in the advance of a microservice design. In: International Conference on Cloud Computing and Services Science - Volume 1 and 2. p. 223–231 (2016)
27. Kruger, C.P., Hancke, G.P.: Benchmarking internet of things devices. In: 2014 12th IEEE International Conference on Industrial Informatics (INDIN). pp. 611–616. IEEE (2014)
28. KubeEdge. `https://kubeedge.io`, [Online; accessed 19-July-2022]
29. Kubernetes. `https://kubernetes.io`, [Online; accessed 19-July-2022]
30. Kumar, H.A., Rakshith, J., Shetty, R., Roy, S., Sitaram, D.: Comparison of iot architectures using a smart city benchmark. Procedia Computer Science **171**, 1507–1516 (2020)
31. Microservices, "https://martinfowler.com/articles/microservices.html"
32. Microservices Workshop: Why, what, and how to get there. `http://www.slideshare.net/adriancockcroft/microservices-workshop-craft-conference`, [Online; accessed 19-July-2022]
33. Newman, S.: Building microservices. " O'Reilly Media, Inc." (2021)
34. Papapanagiotou, I., Chella, V.: Ndbench: Benchmarking microservices at scale. arXiv preprint arXiv:1807.10792 (2018)
35. Paul, S.K., Jana, S., Bhaumik, P.: On solving heterogeneous tasks with microservices. Journal of The Institution of Engineers (India): Series B **103**(2), 557–565 (2022)
36. Poess, M., Nambiar, R., Kulkarni, K., Narasimhadevara, C., Rabl, T., Jacobsen, H.A.: Analysis of tpcx-iot: The first industry standard benchmark for iot gateway systems. In: 2018 IEEE 34th International Conference on Data Engineering (ICDE). pp. 1519–1530. IEEE (2018)
37. Puppet Configuration Management. `https://puppet.com`, [Online; accessed 20-July-2022]
38. Shukla, A., Chaturvedi, S., Simmhan, Y.: Riotbench: A real-time iot benchmark for distributed stream processing platforms. arXiv preprint arXiv:1701.08530 (2017)
39. Slee, M., Agarwal, A., Kwiatkowski, M.: Thrift: Scalable Cross-Language Services Implementation (Apr 2007), facebook White Paper
40. Soldani, J., Tamburri, D.A., Van Den Heuvel, W.J.: The pains and gains of microservices: A systematic grey literature review. Journal of Systems and Software **146**, 215–232 (2018)
41. Sriraman, A., Wenisch, T.F.: usuite: A benchmark suite for microservices. In: International Symposium on Workload Characterization. p. 1–12 (2018)
42. Thrift Software Framework, "http://wiki.apache.org/thrift/"
43. Ueda, T., Nakaike, T., Ohara, M.: Workload characterization for microservices. In: International Symposium on Workload Characterization (2016)

44. Villamizar, M., Garcés, O., Castro, H., Verano, M., Salamanca, L., Casallas, R., Gil, S.: Evaluating the monolithic and the microservice architecture pattern to deploy web applications in the cloud. In: 2015 10th Computing Colombian Conference (10CCC). pp. 583–590. IEEE (2015)
45. Wang, L., Zhan, J., Luo, C., Zhu, Y., Yang, Q., He, Y., Gao, W., Jia, Z., Shi, Y., Zhang, S., Zheng, C., Lu, G., Zhan, K., Li, X., Qiu, B.: Bigdatabench: A big data benchmark suite from internet services. In: Proceedings of the First International Symposium on High-Performance Computer Architecture. p. 488–499 (2014)
46. Yeung, A.: The Six Most Common Microservice Architecture Design Pattern (Mar 2020), https://medium.com/analytics-vidhya/the-six-most-common-microservice-architecture-design-pattern-1038299dc396 [Online; accessed 20-July-2022]
47. Zhou, X., Peng, X., Xie, T., Sun, J., Ji, C., Li, W., Ding, D.: Fault analysis and debugging of microservice systems: Industrial survey, benchmark system, and empirical study. IEEE Transactions on Software Engineering **47**(2), 243–260 (2018)
48. Zhou, X., Peng, X., Xie, T., Sun, J., Xu, C., Ji, C., Zhao, W.: Benchmarking microservice systems for software engineering research. In: International Conference on Software Engineering. p. 323–324 (2018)