Operating Systems for Resource-adaptive Intelligent Software: Challenges and Opportunities

XUANZHE LIU, Peking University
SHANGGUANG WANG, Beijing University of Posts and Telecommunications
YUN MA and YING ZHANG, Peking University
QIAOZHU MEI, University of Michigan
YUNXIN LIU, Microsoft Research
GANG HUANG, Peking University

The past decades witnessed the fast and wide deployment of Internet. The Internet has bred the ubiquitous computing environment that is spanning the cloud, edge, mobile devices, and IoT. Software running over such a ubiquitous computing environment environment is eating the world. A recently emerging trend of Internet-based software systems is "*resource adaptive*," i.e., software systems should be robust and intelligent enough to the changes of heterogeneous resources, both physical and logical, provided by their running environment. To keep pace of such a trend, we argue that some considerations should be taken into account for the future operating system design and implementation. From the structural perspective, rather than the "monolithic OS" that manages the aggregated resources on the single machine, the OS should be dynamically composed over the distributed resources and flexibly adapt to the resource and environment changes. Meanwhile, the OS should leverage advanced machine/deep learning techniques to derive configurations and policies and automatically learn to tune itself and schedule resources. This article envisions our recent thinking of the new OS abstraction, namely, *ServiceOS*, for future resource-adaptive intelligent software systems. The idea of *ServiceOS* is inspired by the delivery model of "*Software-as-a-Service*" that is supported by the Service-Oriented Architecture (SOA). The key principle of *ServiceOS* is based on resource disaggregation, resource provisioning as a service, and learning-based resource scheduling and allocation. The major goal of this article is not providing an immediately deployable OS. Instead, we aim to summarize the challenges and potentially promising opportunities and try to provide some practical implications for researchers and practitioners.

This work was partially supported by the Key-Area Research and Development Program of Guangdong Province under the grant number 2020B010164002, the National Natural Science Foundation of China under the grant number 61921003 and 61725201, the Beijing Outstanding Young Scientist Program under the grant number BJJWZYJH01201910001004, the Alibaba Group's University Joint Research Program, and the MSRA Collaborative Research Project. Qiaozhu Mei's work was in part supported by the National Science Foundation under grant number 1633370.
Authors' addresses: X. Liu, Y. Ma, G. Huang, and Y. Zhang, Key Lab of High-Confidence Software Technology, Peking University, No. 5 Yiheyuan Road, Haidian District, Beijing, P. R. China, 100871; emails: {xzl, mayun, hg, zhang.ying}@pku.edu.cn; S. Wang, Beijing University of Posts and Telecommunications, Haidian District, P. R., China; email: sgwang@bupt.edu.cn; Q. Mei, University of Michigan, 3348 North Quad, 105 S. State St., Ann Arbor, MI 48109; email: qmei@umich.edu; Y. Liu, Microsoft Research, 5 Danling Street, Haidian District, Beijing, P. R. China, 100080; emails: yunxin.liu@microsoft.com, via xzl@pku.edu.cn, hg@pku.edu.cn.

1533-5399/2021/03-ART27 $15.00
https://doi.org/10.1145/3425866

CCS Concepts: • **Software and its engineering → Operating systems**; **Distributed systems organizing principles**;

Additional Key Words and Phrases: Operating systems, resource disaggregation, service-oriented, machine learning

ACM Reference format:
Xuanzhe Liu, Shangguang Wang, Yun Ma, Ying Zhang, Qiaozhu Mei, Yunxin Liu, and Gang Huang. 2021. Operating Systems for Resource-adaptive Intelligent Software: Challenges and Opportunities. *ACM Trans. Internet Technol.* 21, 2, Article 27 (March 2021), 19 pages.
https://doi.org/10.1145/3425866

1 INTRODUCTION

In the past decades, we have witnessed the tremendous rapid development and wide deployment of the Internet. Beyond the ubiquitous connectivity, the Internet also brews a variety of new computing paradigms, including the Web computing, cloud computing, mobile computing, Internet-of-Things, edge computing, and so on, and leads to the burst of applications. Today, we are also surrounded by the substantial software systems, which are "eating" the world and acting as the infrastructure of our human society and civilization.

Compared to the software running on the single machine, software running on the Internet computing can contain abundant heterogeneous resources that reside on zillions of distributed devices or platforms, including cloud data center, edge server, PCs, smartphones, IoT devices, and so on. The ever-continuous advances of information technology result in the resource updates or changes, e.g., new hardware, devices, architectures, and evolution of libraries/APIs. When these updates or changes happen, software developers must decide whether to re-engineer their systems to take advantage of new or improved resources, incurring the expenses that such migration imposes, or remain wedded to a less up-to-date system that may operate sub-optimally. Hence, software systems should be robust and intelligent enough to adapt to the changes of heterogeneous resources, both physical and logical, provided by their running environment. One exceptional example is the recently announced DARPA's push for software systems that "remain robust and functional in excess of 100 years" [34], far beyond the lifespan of their original system designers and hardware/software resources.

To meet the preceding challenges, the operating systems (OSes), which act as the "control plane" [35] to take responsibility of resource management, plays the key role. However, traditional OSes fall in limitations in both *structural* and *behavioral* perspectives.

• **Monolithic architecture limits resource adaptation.** The structures of most current OSes are substantially large and vastly complex, making them difficult to maintain, evolve, update safely, and run reliably. One key factor is that most popular OSes usually follow the "*monolithic*" architectural style where all resource components, such as CPU and memory, are packaged together to run the application [37]. Such a monolithic style is widely adopted in modern OS kernel design, monolithic kernel, micro-kernel, and exokernel, and establishes the foundation for a variety of commodity OSes such as data center OS and Android. However, there have been a lot of debates on the limitations and ineffectiveness of monolithic architecture. First, the monolithic architecture usually runs an OS on a single machine, assuming local resource components into shared memory, storage, network, sensors, and so on, hence the resource management for distributed resources is inherently absent. Second, various resource components are tightly coupled and tangled with one another, and the resource management is too fixed and inflexible to add, remove, and change resource components. Last but not the least, emerging heterogeneous new resources, such as ASIC

or new specific sensors, should be taken into account in the OSes. Unfortunately, adopting and deploying new resources into existing monolithic architecture is quite painful and costly.

• **Adequate OS tuning for specific purposes and scenarios is quite hard.** OSes are traditionally built by the professional software developers and domain experts with long and recurring engineering efforts. When installing most popular OSes such as Linux and Windows, we usually have to adopt general-purpose designs and leave various tuning options at or after the OSes are installed [51]. The common practice is to install OSes with their default configurations, policies, or mechanisms. Due to the various and changing user requirements, the "*One-Configuration-Fits-All*" policy is no longer adequate, and the timely and proper tuning for specific applications or scenarios is required. However, in practice, these configurations or policies cannot be easily and properly tuned to adapt to the application's changing needs and behaviors, as they usually require the domain knowledge and expertise from OS experts. For example, there are over 17K kernel configurations in Linux v 5.1, the manual tuning task is extremely ad hoc and time-consuming, and could lead to some errors or fatal failures.

Hence, we believe that the need for building an OS for resource-adaptive and intelligent software systems is urgent. The recent work emerging ubiquitous operating system (UOS) [33], i.e., the future OSes, should be designed by following the capability of "software-defined." To echo such a trend, this article envisions the design principles for the OS and shares our preliminary idea of a high-level OS abstraction, namely, *ServiceOS*. From the structural perspective, inspired by the Service-Oriented Architecture, *ServiceOS*proposes to break the monolithic architecture into a set of distributed resource components in form of self-described "resource component services" that can be accessed from external APIs and via high-speed network. These services can be dynamically composed to build "customized" application-oriented OSes for specific application needs or scenarios. From the behavioral perspective, rather than relying on the manual tuning, *ServiceOS* aims to employ the advanced machine learning techniques to predict and generate the adequate configurations or policies that are used to tune OSes dynamically with few human efforts.

Indeed, building OSes like *ServiceOS* is not easy in practice, requiring careful considerations or even radical designs in various system-wide issues, including performance, security, reliability, and so on. We realize that the principles of *ServiceOS* are at dawn but some relevant techniques have been demonstrated by various existing efforts. It should be mentioned that the goal of this article is not to provide an immediately actionable OS along with its design and implementations. We focus on analyzing the benefit of adopting *ServiceOS*, discussing challenges and opportunities, and sharing some practice and experiences.

2 DESIGN PRINCIPLES AND ABSTRACTION OF SERVICEOS

In this section, we briefly describe the design principles of *ServiceOS* and present its abstraction.

2.1 Design Principles

The goal of *ServiceOS* aims to make OSes adaptive to the underlying resource and environment changes, both robustly and intelligently. However, compared to traditional systems software, such as middleware or DBMS that are built as OS extensions, *ServiceOS* tries to rethink the OS design principles in terms of both the structure and behavior in the proposed OS abstraction.

In our opinion, although the managed resources and supported workloads of an OS become rather complex, the design guideline of OS should be as simple as possible. The basic guideline is that the applications running on the OS do not have to understand the resource and environment changes, while the OSes play as a control plane to promise that all changes do not interfere with the normal operations of already running applications. Rather than the current OSes that usually hold a monolithic structure, the OS should act as a control plane that can manage a set of loosely coupled

components, each of which can be developed and evolved independently. These components can be dynamically composed to form an OS according to the changes from either environment or application requirements. In other words, beyond the general-purpose "core" OS functionalities that can serve fixed and predefined applications requirements, the OS should be flexibly "*defined*" by the specific application domains and contexts.

Following the basic idea, we argue that the OS meeting resource-adaptive requirements should take into account the following issues:

- **Loosely coupled flexible architecture over resource disaggregation.** Recently, there have been some debates on the "disaggregation" of datacenter servers [14, 37]. It is believed that the OSes, including those running on the server, smartphones, and IoT devices, should consider breaking the monolithic structure and organize all hardware resources, including CPU, memory, storage, and other I/O devices, as independent, fine-grained, distributed, and network-attached components. In particular, rather than the hierarchical architecture in traditional OS, we enforce that resource components should run on a loosely coupled and flexible architecture, making it much easier for every single resource component to evolve independently. Instead of a physical motherboard, the connections between resource components can be made over the high-bandwidth network. In this way, the overhead of integrating all hardware physically can be reduced. Meanwhile, the disaggregation can also help the flexible scheduling of every single resource component across various parallel applications.
- **Autonomous resource provisioning as services.** Rather than a pre-installed general-purpose OS for all applications, the disaggregated and distributed resource components should be composed dynamically into an OS for a specific purpose or application scenarios. To make the whole OS robust, every single resource component should be responsible for its own trustworthiness, including security, reliability, fault-isolation, and so on. Inspired by the principle of Service-Oriented Architecture (SOA) and software-defined systems [33], we consider that all the resource components are provided as self-described and autonomous services, or, namely, Resource Component Service (RCS), which are accessed by externalizable APIs. Every single RCS is required to encapsulate all the software stacks and management capabilities with runtime facilities, e.g., in form of container, which is a popular mechanism in current SOA solutions such as microservice or serverless computing [19].
- **Learning-to-adapt automation for resource allocation and tuning.** Traditionally, when installing and initializing the OSes, we can apply some default configurations and policies, which can be manually and statically changed to achieve the best performance, typically from domain experts' knowledge in a "top-down" way. In contrast, we rethink that OSes should be dynamically tuned, in a data-driven and learning-based fashion, which should act as "*bottom-up*" way, or at least, the "*meet-in-the-middle*" style by synthesizing the learning results and domain knowledge. In a sense, the OSes should provide the controller that automates the "*Monitoring-Learning-Executing*" control loop: The *Monitoring* activity collects the runtime information, including system logs, resource state changes, environment, and so on; the *Learning* activity leverages the advanced machine learning techniques to build models that can accurately generate the adequate configurations for achieving the best application-level performance and predict the resource and environment changes; the *Executing* activity automatically applies the derived configurations and policies to tune the OSes, without having to involve too many human engineering efforts. In practice, the executable configurations can be updating the values of parameters specified in existing RCS APIs, adding new RCS, or removing existing RCS.

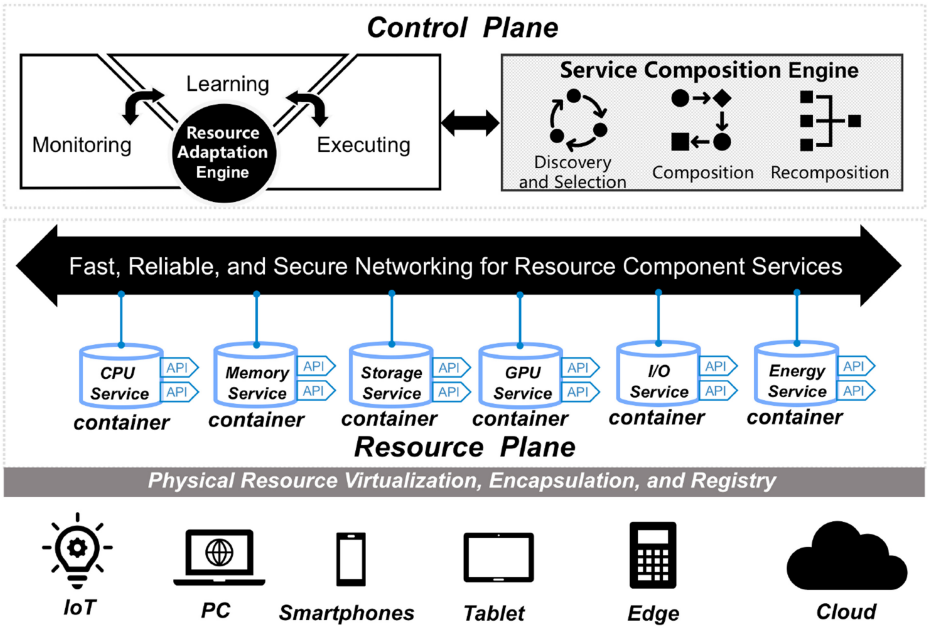

Fig. 1. Abstraction of *ServiceOS*.

2.2 *ServiceOS* Abstraction and Application Model

Based on the preceding design principles, we draw the conceptual abstraction that is visible to *ServiceOS*'s principals, i.e., users and applications, as illustrated in Figure 1.

The *resource plane* consists of distributed resource component services that are provided by the node, e.g., a server, a smartphone, or an IoT device. As the basic unit, a hardware resource is decoupled out of the monolithic OS kernel installed on the node where the resource resides and encapsulated as a resource service component. Every single RCS is exposed via a set of narrow but well-defined APIs. Every published RCS is registered on a global resource directory and naming service. As a result, in *ServiceOS*, there are a large amount of distributed RCSes that can be searched and discovered, e.g., from a global repository. From the user's or developer's perspective, *ServiceOS* exposes an RCS along with its own container that is deployed on the physical compute node. An RCS container has a unique ID, a unique accessible address (typically as a virtual IP), a meta-data description (e.g., the resource type and its owner), and a built-in monitor that introspects the resource state (e.g., idle, failure, overloaded, etc.). An RCS container can run various RCSes, each of which actually refers to an actual hardware resource type. Also, an RCS can be included in various containers. At the system level, *ServiceOS* takes the responsibility of isolating and protecting the RCS assigned to one container from others.

Over these RCSes and their containers, the *control plane* consists of two major components, the *Composition Engine* for orchestrating some RCS containers and the *Learning-based Adaptation Engine* that automates resource scheduling and OS tuning.

The composition engine takes charge of building an application-oriented OS according to user and application requirements. We assume that all RCS containers communicate with one another via a fast and reliable network connection. In practice, such a network can be RDMA in datacenters, or can be high-bandwidth and low-latency 5G/Wi-Fi among distributed devices. To flatten the RCS containers, *ServiceOS* supports the event-driven style, where the communication is done via

an event bus at the OS level. Note that the RCS should be autonomous; hereby, we make the container responsible for its hosted RCS's security and failure. Hence, the changes of a resource is completely transparent to user or application. Indeed, the change of one RCS can affect others, and the composition engine should process the problem globally. We will discuss some design policies in Section 3.

The learning-based adaptation engine interacts with the monitor that collects the status of RCSes hosted by the container, trains the model from the collected data, performs the classification or prediction task for resource adaptation (e.g., deriving configurations or policies), makes the adaptation decision, and finally synchronizes the decision with the composition engine to re-compose the OS to fulfill the new requirements caused by resource and environment changes. Note that the adaptation engine is considered as a system-level component rather than an application-level one and can run in the kernel space.

3 CHALLENGES AND OPPORTUNITIES

ServiceOS draws only an ideal blueprint for the OS abstraction that fits the requirements of resource-adaptive software. Indeed, in practice, building an OS is quite hard. This article does not aim to provide an immediate executable and deployable solution for *ServiceOS*. Instead, we try to summarize the challenges from recent related efforts and discuss opportunities for building an OS like *ServiceOS*.

3.1 Resource Disaggregation and OS Decoupling

The need for fine-grained resource elasticity and high resource utilization has led to the birth of resource disaggregation [18]. The key motivation of resources disaggregation is that different resources such as CPU, memory, storage, and network, exhibit significant trends in terms of cost, performance, and power scaling [14]. Resource disaggregation greatly improves resource utilization, elasticity, heterogeneity, and failure isolation, since each resource component can operate or fail on its own, and its resource allocation is independent from other components. Resource disaggregation has been demonstrated to be an efficient solution in terms of provisioning and scheduling of individual resources across multiple workloads [11, 14]. Resource disaggregation is regarded as a forthcoming shift of resource management paradigm, especially in the cloud computing and datacenters. To enable disaggregation in *ServiceOS*, there are some key techniques to be addressed.

Coherent co-design of memory utilization. Resource disaggregation decouples the computation and storage. Ideally, all hardware memory functionalities (e.g., page tables, TLBs) are encapsulated in a memory component, while only caches are kept in the computation component (e.g., a process). Given a clean separation of computation and memory hardware units, the allocation and management of memory can be completely transparent to the computation. In a sense, the memory component can autonomously choose its own memory allocation technique and virtual to physical memory address mappings [37]. In addition, memory can be further decoupled according to their types, i.e., local or remote, and physical or virtual. Memory disaggregation decouples physical memory allocated to virtual servers (e.g., VMs/containers/executors). The decoupling aims at allowing the server under high memory pressure to use the idle memory either from other servers hosted on the same physical node (node-level memory disaggregation) or from remote nodes in the same cluster (cluster-level memory disaggregation) [25]. In a sense, memory disaggregation can leverage the latency gap between network I/O and storage I/O to enable DRAM memory to expand to the faster tier(s) in the memory hierarchy before resorting to the slower external storage tier. In practice, we can decouple the physical memory allocated to virtual servers (e.g., VMs and containers) at their initialization time from the runtime management of the memory. A

considerable solution is to explore the coherent co-design of virtual server memory and node-level memory, and the co-design of local memory and remote memory.

High-bandwidth and low-latency networking. The high bandwidth and reliable network is undoubtedly the most significant technical foundation for adopting the resource disaggregation, since the resource components are attached via network and may transfer a large volume of data from one another, especially for big data analytics workloads. It is reported that, to preserve the application-level performance for Spark, Hadoop, and GraphLab workloads deployed in datacenters, the required network bandwidth should be 40 Gbps–100 Gbps, and the required latency should be around 3 μs–5 μs [14]. In practice, *ServiceOS* can take into account the state-of-the-art programmable switch, NIC, and RDMA, to promise the quality of bandwidth and latency in datacenters. Take the RDMA as examples. To enable the connections of disaggregated memory components, the RDMA write/read operations can help implement the connection for data transfer, and the RDMA send/receive operations can help implement the connection for control tasks. For each connection, two types of channels are established, i.e., the RDMA channel for maintaining the network connection and data transfer, and the disaggregated component channel for interacting with the remote node agent, maintaining the system status, and performing placement and eviction algorithms [25].

Disaggregation beyond datacenters. In datacenters, resource disaggregation tries to decouple different types of resource components and allows every single type of resource component to evolve independently. Each type of resource component is built as a standalone resource "blade" and interconnected with one another via a high-bandwidth network fabric. Compared to existing work, in the context of *ServiceOS*, the scope of resource disaggregation should not be limited to the rack scale, but should be much wider, e.g., even on other devices such as edge. Certainly, when performing disaggregation for resources, the network bandwidth and latency significantly matters. When orchestrating distributed resource components beyond datacenters, we cannot simply rely on the 5G or Wi-Fi connections. With respect to the principle of *ServiceOS*, unlike workloads deployed in datacenters, applications built over such a wider range of disaggregation should not exchange very large volume of data.

- **Decoupling Operating System**

Resource disaggregation inherently requires to break the monolithic OS into a flexible structure. Hence, it is straightforward to decouple the existing OS kernels into a set of loosely coupled components (which are delivered as a service, as illustrated later, e.g., following the RESTful architectural style). Then, an OS is composed by selecting a set of components. Following the idea from service composition, new components can be added, and a component can be removed, updated, or replaced by others, so the OS can then be dynamically re-composed according to the environment changes.

The Separation-of-Concern (SoC) is a widely adopted software engineering principle by decomposing a monolithic software system into a set of smaller components. Each component groups code and data for relatively independent functionalities, and the connections between components should be as loose as possible. However, decoupling an OS kernel is quite challenging. Applying the SoC principle in OS needs a lot of efforts to support fault isolation, runtime flexibility, live update, process migration, privilege separation, and so on. We briefly summarize three possible solutions to efficiently decouple the OS.

Lightweight and flexible OS kernels. To support the resource disaggregation, future OS kernels should be more lightweight and flexible. Unikernels, known as Library OS, becomes a promising solution to this end. The key insight of Library OS is to eliminate the privilege barrier between kernel and user spaces and build a standalone image that contains both the kernel and the

application. A major limitation of Library OS is the non-trivial developer efforts to deal with the applications that rely on Linux system call APIs [22]. In addition, current LibraryOS mainly runs in a single server and the management of distributed resources is a bit weak. In contrast to making existing kernels lightweight, we think that *ServiceOS* should leverage new kernels that are also proposed to meet the resource disaggregation. One promising solution is that the *splitkernel* disseminates an OS into pieces of different functionalities, each running at and managing a resource component [37]. Over the splitkernel, resource components can be heterogeneous and can be added, removed, and restarted dynamically without affecting the rest of the system. Each splitkernel can be equipped with a monitor that operates locally for its own functionality and only communicates with other monitors when there is a need to access resources there. At the system level, all resource components communicate by message passing over a common network such as RDMA, and the splitkernel globally manages resources and component failures.

Application-specific OS debloating. Another application-oriented OSes, rather than the general-purpose OSes, are a bit more adequate and desired in the resource-adaptive software systems. Application-oriented OS kernel debloating aims at reducing the kernel code that is not needed by the target applications and is effective to reduce the complexity of OSes. A debloated OS kernel includes only features for supporting the target application workloads. To this end, *ServiceOS* can consider leveraging the *configuration-based* approach, which can reduce the kernel size, attack surface, and security vulnerabilities. However, to better meet the resource-adaptive requirements, existing debloating solutions still suffer from the lack of support for fast application deployment, coarse-grained tracing, and incomplete coverage for the kernel footprint [22]. Another problem is that most of existing debloating approaches usually need manual efforts to configure the kernel options and are not very practically effective. Some efforts such as *KConfig* [20] provide user interfaces for selecting kernel configuration options, while the usability is yet quite poor. To address the challenges, one possible solution is to leverage the lower-level tracing to track the kernel code and automatically map the kernel code to the kernel configurations. For example, `COZART` applies the instruction-level tracing and generates a set of configurations that are specific to an application and its running environment offline and builds the kernel by composing a set of configurations [22]. In addition, the configurations can be reused to incrementally build new kernels. Indeed, such efforts can be quite helpful to customize an OS kernel. To better accommodate the resource-adaptive requirements, the configurations should be derived on-the-fly and updated dynamically to support the *online* and *continuous* re-building of a debloated OS kernel.

Reducing state spill to achieve flexible composability. To enable the better and efficient decoupling of an OS, a recent promising trend is to reduce the *state spill* [9], which is considered to be a root cause of entanglement within the OS. State spill indicates that the software component's state undergoes a lasting change as a result of handling an interaction with another component. For example, in the typical client/server system, when the changes to a server-side component's internal state persist beyond the end of its interaction with the client-side component, the state spill occurs. State spill can limit OS decoupling, as it can lead to the entanglement between components in the OS. More specifically, to promise the correctness and consistency, the OS cannot release the states until interactions between two components terminate. Eliminating the state spill requires a significant change for the OS design, because the current OS is very complex: The interactions between components are severely entangled and thus hard to distill. In practice, one possible solution is to decouple OS with the principles of no encapsulation, stateless communication, universal and connectionless interface, and pattern reuse, as proposed in the `Theseus` system [10]. Modules in `Theseus` eschew traditional encapsulation in favor of decoupling a module's state, and thus its notion of progress with other modules, from its entity bounds. In this way, we can force the caller component to assume the responsibility of maintaining the state of its progress with callee

component, while eliminating the state spill. The idea of Theseus is probably adequate in building *ServiceOS*, as it enables the flexible and dynamic composition of an OS. However, applying such a "*clean slate*" design for existing OSes requires substantial engineering efforts.

3.2 Resource Virtualizaiton and Service-oriented Provisioning

The "granularity" of resource provisioning can affect the capability and efficiency of resource adaptation. Inspired by the SOA, we regard that all resources should be encapsulated and provisioned as services. SOA was proposed in the early 2000s and has been widely adopted in web computing and cloud computing, e.g., the RESTful architectural style. The core principle of SOA is to make the software component as an autonomous entity that can be accessed by externalizable application programming interfaces (API). From the software engineering perspective, SOA provides a perfect programming paradigm that can achieve various principles such as autonomous, isolation, loose coupling, dynamical composability, and so on. Essentially, the principle of SOA establishes the technical foundation of software-defined platforms such as SDN and SDS [33], and thus can support the development of ideal capabilities for resource-adaptive software systems.

• **Secure and lightweight container model**

Most SOA systems support the multitenancy. Multitenancy allows resources to be shared across various applications. To protect the security, an application must be isolated (so one workload cannot access, or infer, data belonging to another application) and for operational concerns (so the noisy neighbor effect of one workload cannot cause other workloads to run more slowly) [3]. In the early age of cloud computing, the virtual machine is the basic service provisioning unit that encapsulates the resources and their states and deploys them in the form of services. However, traditional VM like Xen imposes a rather monolithic, coarse-grained, and heavy-weight model, which is demonstrated to be inefficient for resource adaptation [19].

The microservices architecture and serverless such as Function-as-a-Service (FaaS) become quite popular recently. The microservices architecture aims to decouple a monolithic application into a set of components, each of which runs in an isolated runtime context, typically a container. Containers have quickly become the de facto solution to manage and deploy large-scale distributed applications, such as web servers, data processing and analytic frameworks, in-memory key-value stores. Containers, in contrast to traditional VM, provide a way to virtualize and isolate the operating system, allowing multiple applications to run in a single operating system. Essentially, containers make the software rather than the hardware that is virtualized. With container management systems like Docker and orchestration systems like Kubernetes to control applications and dynamically provision their resources, current cloud services can be extremely scalable, reliable, and reactive. FaaS, considered as the next-generation cloud computing, further promotes the SOA principle to more extreme end [19]. FaaS is proposed as the interface to usage-driven, stateless, or even serverless backend services and offers an intuitive, event-based interface for developing cloud-based applications.

To make the software systems adaptive to environment, resources should be provisioned with new functions along with the supporting runtime environment (e.g., containers) very fast, e.g., in milliseconds. In this way, resources can be switched between workloads quickly when environment changes. Intuitively, both microservices and FaaS are built upon containers such as those who use Docker and LXC. These containers rely on the isolation mechanisms built into the Linux kernel. To support multitenancy, container implementors can choose to improve security by limiting syscalls, at the cost of breaking code that requires the restricted calls [3]. In practice, we realize that the current cloud service providers need to deal with tradeoffs between the hypervisor-based virtualization and Linux containers. Some recent efforts provide specialized container, rather than

general-purpose solutions. For example, to support the serverless computing, Firecracker [3] accommodates KVM with a new VM monitor, device model, and API for managing and configuring MicroVMs, and can achieve better isolation along with high speed (about 100 ms for booting a container) and low system overhead (5 MB per container). These efforts have some limitations, e.g., without the support of a BIOS and the capability of booting arbitrary kernels. In addition to security and containers should be able to emulate legacy devices or PCI, and support VM migration from legacy software systems.

• **Ubiquitous resource virtualization for service provisioning**

Resource virtualization is the system foundation for service deployment and delivery. Beyond CPU virtualization in the cloud, we regard that the virtualization should be more ubiquitous, making more resource types on heterogeneous devices delivered as services.

GPU virtualization. Given the increasing demand of machine learning and deep learning, virtualization techniques for other hardware components such as GPU are also required. Compared to CPU virtualization, GPU virtualization has some unique challenges. To achieve high throughput and low latency, GPU virtualization needs to process the data flow, including I/O operations, task batching, and data transferring in the CPU-GPU pipeline, and promises the data isolation for security concerns. It also requires the careful parameter tuning, such as the GPU batch size and the number of CPU and GPU threads. These parameters are highly correlated and need a lot of manual efforts and expertise domain knowledge. In practice, the GPU virtualization should be designed for both temporal sharing [40] and spatial sharing [50]. However, to the best of our knowledge, the GPU virtualization is still under low utilization.

Network virtualization. Indeed, there have been numerous efforts for Network Function Virtualization. As *ServiceOS* proposes the service-oriented resource provisioning in form of containers, there are still some challenges and improvement spaces for communications among containers. The container overlay networks provide the portability to allow a set of containers to communicate with one another based on their own independent IP addresses and port numbers [13]. As VM/container has its own network stack, the hypervisor has to send/receive raw overlay packets without the context of network connections. As a result, each packet has to traverse the network stack twice in both the sender and the receiver's host OS kernel, which introduces non-trivial overhead. Some recent efforts have made efforts to alleviate the problem. For example, `Slim` designs a low-overhead container overlay network that provides network virtualization by manipulating connection-level metadata [52]. By removing packet transformation from the overlay network's data-plane, `Slim` makes the packet go through the OS kernels only once.

Peripheral I/O device virtualization. In addition to the compute, storage, and network, virtualizing the peripheral I/O devices and publishing them as services is also required. Nowadays, I/O devices become more diverse and heterogeneous, especially for the mobile and IoT devices, which are equipped with a large number of cameras, audio, accelerometer, and compass, several network devices such as Wi-Fi, Bluetooth, and NFC. Virtualizing and publishing these I/O devices in form of services can not only support flexible composition, i.e., I/O devices can be shared among various applications, but also allow the new security mechanisms. Traditionally, the OS takes the responsibility to promise security of all the I/O devices that are loaded. The OS also assumes that the I/O devices along with their drivers are bug-free and trustworthy. As a result, the trust model in current OS and the I/O devices are tangled. Given that each device can evolve and update independently, the security threats introduced by I/O devices become more challenging. In 2016, it was reported that 85% kernel bugs in Android were originated from device drivers. When I/O devices along with their software stack are virtualized and encapsulated as services, they should be responsible for their own security, not relying on the OS [36]. In this way, the mutual trust

between the OS and these devices is eliminated. Meanwhile, the devices along their drivers are isolated from the OS kernel, their vulnerabilities (and even malice) do not lead to kernel exploits. Indeed, such a design is quite radical, but it has to hold a strong assumption that employs a secure enclave, e.g., the ARM TrustZone or SGX, to construct a secure channel.

Mobile device virtualization. The scope of virtualization should be also expanded for other devices beyond the cloud-side resources. Virtualization support for the types of processors used in client-side devices (such as smartphones and wearables), edge, and IoT systems is much less common than for server/cloud side CPUs [44]. Improved remote management of virtual systems will most likely be necessary, given the impracticality of physical access and potentially limited bandwidth. Virtualization also needs to take into account power and bandwidth limitations. Virtualization for client-side devices such as smartphones, IoT devices can bring benefits, allowing multiple virtual devices to run simultaneously on the same physical device in an isolated, secure manner and to serve various requirements. Indeed, supporting virtualization on these devices has unique challenges. These devices are relatively resource-constrained, and running an entire additional OS and user space environment in a VM imposes high overhead and limits the number of instances that can run [4]. Slow responsiveness is less acceptable, as it can significantly compromise user experiences. Another problem is that these devices incorporate a plethora of resources that applications can use, such as GPS, cameras, and GPUs. Hence, compared to the cloud-side container model that can run isolated operating system instances for various applications, it requires the lightweight OS virtualization to provide virtual namespaces that can run multiple virtualized devices on a single OS instance. In addition, the virtualization should support the fine-grained resource access and utilization, e.g., the data inside an app.

Web browser virtualization. Due to the advancement of JavaScript and HTML5, Web applications can provide the comparable user experiences against the native apps. Client-side virtualzation for Web applications is also inspiring. Based on the idea of browser OS such as iBOS [41] and MashupOS [42], the Embassies system moves the cloud-side vendor code down to the client [16]. On the client, apps can have the fast, reliable access to the resource, but the semantics of isolation remain identical to the cloud model. Each vendor has autonomous control over its software stack, and each vendor interacts with other vendors (remote and local) only through opt-in network protocols. Essentially, such a virtualization manner consistently copies the isolation and security mechanisms from the cloud to the client-side device while making the client act as an *pico-datacenter*. In practice, however, this browser-level virtualization still has some performance overhead when directly accessing the remote resources and requires a lot of efforts to refactor the browser kernel to support communications between pico-datacenters, i.e., preserving the same origin policy and sandboxing.

- **Service discovery, selection, and composition**

When applying the resource disaggregation and SOA principle at an extreme level, all resources are completely independent from one another. Hence, resource allocation and scheduling can quickly adapt to workload requirements and environment changes, packing services (e.g., at function level) into containers, and composing these services (e.g., as a function chain) to build an application-oriented OS.

The service composition problem has been studied for a long time in the Web service community, and a huge body of valuable preliminary efforts are inspiring and useful [24]. In practice, as mentioned previously, every RCS container has a unique ID and a unique virtual IP address, and the RCS can be developed by following the RESTful architectural style that has been widely adopted for Web applications, mobile apps, and cloud applications. In this way, an RCS can be exposed to *ServiceOS* principals, both users and applications, with universal interface semantics.

Given that there can be a huge number of RCS as candidates, *ServiceOS* should be capable of quickly selecting the most adequate service for composing the OS for application and user requirements. In particular, when the resource or environment changes, the composition should be *dynamic* rather than *static* and *fixed*. With the adaptation policies generated from the adaptation engine described later, the composition requires to replace an old RCS with a substitute, add new RCSes, or remove the RCS. To this end, the composition logic can follow the *Event-Condition-Action* rules that encode the logic of composition in function of runtime events, as rules can easily be added, modified, or removed to adapt to resource or environment changes. It is worth mentioning that the composition itself is also delivered as a service, so all the adaptations can be transparent to the OS principals. In other words, with the support of *ServiceOS*, applications and users are not aware of the underlying changes.

• **Stateless and stateful policies in service composition**

In practice, service composition should take into account the states. To make the composition flexible enough for the resource adaptation goal, we have proposed two considerations for service composition, i.e., the stateless communication, and the consistency of shared state.

First, to comply with the "reducing state spill" policy, the communications among composed services should be as *stateless* as possible to make the interactions between services clean and simple. Formally, given the message passing m for two services S_1 requesting S_2, m is expected to contain all data required for S_2 to process the request and make response. In other words, S_2 can access only the data passed in from S_1, without involving any other state of S_1 after the m occurs. Supporting stateful communication does not mean that no state is included in m, but promises that the composition has no assumption of prior state. In this way, the communication does not have to maintain intermediary states that can lead to additional tangling between S_1 and S_2.

Second, the composition should carefully take into account the consistency of shared state among services while preserving the performance, cost-efficiency, and scalability. For example, a machine learning task that is composed by some low-latency services needs to scale automatically for the possibly emergent burst of model inference requests and dynamically manipulate the data based on request parameters. To this end, the service composition engine needs to maintain the necessary states, such as user sessions, intermediate data in model serving, and so on. Efficiently supporting the shared state service composition is urgently required, especially in current FaaS when several containers are orchestrated [15]. It was reported that latencies and costs of shared auto-scaling storage for serverless applications are orders of magnitude worse than underlying infrastructure such as shared memory, networking, or server-based shared storage [43]. Indeed, it requires the tradeoff between the disaggregation and state dependency. Some recent efforts, such as Cloudburst [38], present the idea, namely, logical disaggregation with physical colocation (LDPC). It deploys resources to different services in close physical proximity, where a running function's "hot" data should be kept physically nearby for low-latency access.

3.3 Machine Learning–empowered Resource Scheduling and Tuning

Besides the structural level changes supported by resource disaggregation and service-oriented provisioning, we believe that the behavioral level changes should be made in OS tuning. To make the resource adaptation timely and efficient, OSes should embrace the advanced machine learning or deep learning techniques to automate the generation of configurations, policies, and scheduling mechanisms and reduce the human efforts.

Although ML has been used in many domains and recently more in system software like DB, OSes have rarely adopted ML techniques, and most research proposals that date decades back are really quite debatable. For example, there are several proposals of using ML techniques (e.g.,

C4.5 decision tree, linear regression) to improve application job average turn-around time. In this section, we discuss some possible OS management that can be facilitated by the machine learning.

• **Learning to schedule and allocate resource**

The first and straightforward benefit from machine learning in OS can help improve the resource allocation and scheduling policies with respect to the user request and system overhead. Traditionally, the policies are made based on some heuristics or simple algorithms. For example, in the Internet-based services such as online video serving systems, the OS needs to allocate more storage space and bandwidth for those popular and frequently accessed files. File systems usually use the policy that statically allocates close-by spaces for files under the same category. However, given the dynamic and diverse user requests, such a policy is not always adequate. Intuitively, OSes can leverage learning techniques to predict the forthcoming application/user requests and more efficiently plan the resource allocation in advance [28]. Another possible scenario is to leverage the ML to perform the mapping for memory/file management, e.g., the mapping from virtual memory addresses to physical memory addresses (which is currently performed by page table), and the mapping from a file name and offset to disk logical block address (which is done by file system multi-level index structure) [51]. In traditional OS, these two types of mapping tasks are crucial to the performance of all memory and storage systems. ML models can help reduce the space costs. In addition, ML models can help derive customized management policies for different workloads. Unlike fix-sized memory pages in current OS, an ML-based mapping is expected to be capable of inferencing any size and offset of memory space, and can potentially be smaller and run faster than a multi-level page table that is predefined in OS. In Lynx system, ML is used to better perform prefetching from SSDs [23]. It leverages Markov Chains to detect I/O workload patterns and compute the transition probabilities between file pages.

• **Learning to derive configurations**

OS configuration is known to be very time-consuming and error-prone, and requires non-trivial manual engineering efforts. Configurations usually contain a lot of parameters, e.g., cost, performance, security, and so on. These parameters are usually highly correlated. It requires the domain knowledge and expertise to make the tradeoffs among these parameters. Usually, OS configuration tasks are done with some heuristics, by trial and error, or with offline experiments, and the configuration options usually keep fixed and unchanged after the OS is installed. Machine learning can facilitate the OS configuration by training the model based on the data such as code evolution, historical traces, and user behaviors of existing applications. In addition, the model for generating configurations can be updated and tuned dynamically to automatically adapt to the resource and environment changes.

It is reported that a lot of configurations, both time-related ones (e.g., interruption frequency, buffer flush frequency) and space-related ones (cache size) can be made by machine learning techniques [51]. More specifically, the configurations can be derived by the *reinforcement learning* and *transfer learning*, where the model made from one application can be transferred and applied onto a new application. It can be also expected that the model can be continuously tuned and improved when more application-specific configurations are accumulated. In practice, we can assume that the models for application-specific configurations can be stored in a repository, and some existing visualization tools can help make the configurations in an iterative and interactive fashion [27].

In practice, deriving precise configurations is challenging. The machine learning models need to explore two search spaces. One is the space covering all possible policies, and other refers to the space of all possible resource states, i.e., idle, busy, or expired. The challenge stems from the fact that individually exploring each of the search spaces can be prohibitive [7]: A search for the true

policies is hard, since they are a small fraction of the policy space, while a search for the violated policies is hard, since they are often sparse.

• **Learning to system diagnosis**

Given that the resource-adaptive systems should support continuous and reliable serving, e.g., running for 100 years [34], the system diagnosis should be fast enough to locate the bugs, errors, and failures. In this way, OSes can immediately take actions to fix them. Due to the high complexity of OSes and applications, the diagnosis should be performed by comprehensively synthesizing a large-scale set of information, including runtime logs, source/binary code, configuration settings, and so on, which can hardly be done by manual efforts. As a result, machine learning–based system diagnosis is a promising approach to improve the efficiency, such as bug location and prediction [6], misconfiguration detection [49], anomaly detection [28], and so on. The diagnosis results can provide immediate insights to make decisions. In practice, the major concern is the accuracy, as the inaccurate diagnosis can contain false positives that may lead to unexpected system failures.

• **Learning as system-level facility**

As discussed above, machine learning can potentially facilitate the OS management in various aspects. However, as we make the learning capability as a core system-level component of the *ServiceOS* abstraction, rather than an application-level optimization, there are some challenges that should be carefully addressed in practice.

Overhead of deploying machine learning models in OS. To build machine learning models for OSes, tuning requires collecting sufficient data from applications, which can inevitably introduce some runtime overhead. In practice, it is more suitable to train the model offline rather than online [45, 51]. The reasons are two-fold. On the one hand, to make the trained model robust and accurate, we require an amount of fine-grained data that can be processed only offline. On the other hand, the offline training can have less impact on the user experiences on the application. Similar to the training process, at the OS level, we also need to preserve some compute, storage, and energy resources that are used to run the learning task. Along with the performance overhead, there is also the runtime overhead to store machine learning models and some intermediate data for inference [39]. In practice, to avoid the consumption of core OS resources, we can consider employing easy-to-deploy resources serving for inference. For example, some recent work like Pocket [21] employs the elastic ephemeral storage for inference tasks at very low cost, which can provide a preliminary effort in this direction. In addition to the training and inference tasks, how to quickly and accurately deploy the machine learning–generated configurations and policies in the OS requires additional system-level resources.

Privacy concerns of building machine learning models. The privacy concerns of data that are collected for building the model should not be ignored. In practice, to produce an accurate machine learning model that can tune OSes for specific purposes or scenarios, it needs to rely on not only the application state and system runtime logs, but also the user behaviors and user-generated data. It requires the OS to transparently monitor how the users interact with their apps, extract meaningful information about their interests, habits, and behaviors, and expose the information to various apps in a secure, private, and uniform way [12]. Due to the recently increasing concerns of data ownership, e.g., the recent release of GDPR [1], some sensitive user data cannot be arbitrarily collected by the app developers. However, whether such data can be collected by OS and shared among apps installed on the OS still remains a grey area. To alleviate such an issue, recent efforts such as federated learning (FL) [8] and differential privacy (DP) [2] can be helpful. For example, we can make the FL and DP as OS services, where user data generated in different apps can be securely shared and trained to build a desirable machine learning model. Another possible way is the on-device training [45], which performs the training tasks over the private data purely on the device,

without having to upload data to the cloud or share the data with externals. It is certain that the on-device training can promise the privacy, but the insufficient data cannot promise the accuracy. In practice, we can pre-train an accurate model on the cloud with public data or non-private data and apply the reinforcement learning or transfer learning on the device.

Personalization of machine learning–generated adaptations in OS. As mentioned previously, the resource-adaptive software systems can concurrently serve various users and applications as principals, which may require different variants of the machine learning model. We need to carefully take into account the personalization of a derived model that can achieve the best user experience. However, in practice, producing and maintaining the model per user or per application can impose a lot of cost and overhead and may be even impractical due to the insufficient data fed for model training. One promising solution is to make tradeoffs between the "global" model and "personalization" model, i.e., training the global model (typically with some public data), deploying the model for the user/application, and tuning the model (typically with user/application-specific data) for personalization purpose [45]. To improve the efficiency, emerging *model-less* fashion can be beneficial, which maintains some model variants (similar network structures) and personalize the model with hyperparameters tuning [48].

Reliability and performance of applying machine learning–generated adaptations in OS. The machine learning models are expected to produce the adaption configurations and policies, which are considered to be automatically and fast applied to the OSes. In practice, such adaptations mainly target at the performance optimization. However, using machine learning to determine the system-level functionalities adaption, such as the CPU interrupts, virtual memory mappings, and file systems, needs to be extremely accurate and reliable [51], as the inaccurate tuning can lead to fatal errors and failures.

4 PRELIMINARY PRACTICE AND EXPERIENCES

The preceding section discussed the challenges and opportunities. Obviously, building an executable system covers a wide spectrum of system research and requires a lot of research and development efforts. In the past a few years, we have made some efforts towards this direction. We briefly enumerate some preliminary work as case studies that can be potentially useful in some aspects to help build *ServiceOS*.

Machine learning–empowered resource management and scheduling. In terms of machine learning–empowered resource adaptation and tuning, we demonstrated how ML can help predict the CDN workloads and allocate the cache size. Compared to the fixed cache policy like LRU, our approach can use ML to decide candidates of cache assignment or eviction. We designed the machine learning model based on the power-law-based app download/update distributions, co-installation patterns, diurnal patterns, and community structure detection to dynamically place the cache for candidates that are more likely to be accessed. We conducted extensive experiments over a large-scale Appstore service provider that serves over 30M daily active users [26, 28]. The results evidenced that the improvement is significant by reducing the cache storage volume while increasing the cache hit ratio of frequently accessed files. For example, with the help of ML models, the cache ratio increases from 82% to 95%. In addition, the ML models reduce the storage spaces and improve the storage utilization by about 40×. Furthermore, we demonstrated that more dimensions such as implementation-level features (the code complexity of a class, the coupling degree among classes, etc.), the description-level features (the textual descriptions, the app category, the quality of illustrating pictures), and the user-behavioral features (download, update, and uninstall) can be synthesized by machine learning models to predict the popularity of a mobile app [30]. The

results can efficiently help the release planning and resource allocation to improve the application experience quality and have been deployed on a leading appstore service provider in China.

Similarly, for the Web cache, we synthesized the user-request patterns and the update logs of resources on Web servers and presented a learning technique that can accurately identify which resources can be loaded from the local storage for a considerably long period [29, 32]. We implemented an in-browser adaptation engine to perform efficient resource packaging where stable resources are encapsulated and maintained into a package, and such a package shall be loaded always from the local storage and updated by explicitly refreshing. *ReWAP* maintains resource packages that can accurately identify which resources can be loaded from the local storage for a considerably long period. Compared to the original Web apps with cache enabled, *ReWAP* can significantly reduce the data traffic with the median saving up to 51%.

Resource provisioning for mobile/edge devices. In contrast to the server/cloud side resources provisioning, we realize that the client-side resource provisioning is still at dawn, and the resources of mobile and edge cannot be simply delivered as services and used by other applications. Previous efforts like Cells [4] proposed a VM-level solution to virtualize the smartphone for enabling app isolation, but cannot well support the fine-grained specific resources such as data, sensor, or energy. We developed the *Aladdin* tool to help automate the release of deep-link APIs to access the data inside mobile apps [31]. *Aladdin* includes a novel cooperative framework by synthesizing the static analysis and the dynamic analysis while minimally engaging developers' inputs and configurations without requiring any coding efforts or additional deployment efforts. *Aladdin* provides a lightweight virtualized layer that can be loaded as a library and translates all Android storage API calls to the calls to our derived deep-link APIs. Compared to similar efforts like *uLink* [5], the virtulization *Aladdin*'s CPU usage is smaller while the memory usage is almost the same. The reason is that *Aladdin* passes a smaller number of activities before reaching the target activity than *uLink* does, requiring more CPU resources to execute the transition activity's logic.

Reducing machine learning overhead. As mentioned previously, deploying machine learning for system management is very challenging, and the additional overhead cannot be simply ignored, especially for the resource-constrained devices. With respect to the possible overhead for foreground applications introduced by learning tasks, our previous work [17] made preliminary efforts, namely, *ShuffleDog* to make the OS adaptive to the learning tasks without compromising the foreground applications. The key idea is to identify all delay-critical threads that contribute to the slow responses and build a resource manager that can efficiently schedule various system resources including CPU, I/O, and GPU, for optimizing the performance of these threads. Based on *ShuffleDog*, our developed *DeepCache* [47] optimized the inference performance by 2× by synthesizing reusable cache among layers in a learning model. To demonstrate how the *DeepWear* [46] proposed various novel techniques such as context-aware offloading, strategic model partition, and pipelining support to efficiently utilize the processing capacity from nearby paired handhelds or edge devices, *DeepWear* brings up to 5.08×−23.0× execution speedup for running machine learning tasks, as well as 53.5%−85.5% energy saving. These techniques can help mitigate the system overhead. However, they are not specific to the OS management and may require additional concerns such as security when being integrated in the OS kernel space.

5 CONCLUDING REMARKS

The ubiquitous connection leads to the era of Internet-based computing environment, where there are abundant resource types spanning over the cloud, edge, PCs, smartphones, and IoT devices. These resources along with their residing environment keep evolving and require the continuous adaption support from software systems. This article envisions the new operating

system abstraction *ServiceOS* for the resource-adaptive software systems. The OS abstraction proposes three major design principles, i.e., resource disaggregation, service-oriented resource provisioning, and machine-learning empowered resource allocation and scheduling. We discuss the rationales behind *ServiceOS*, along with some challenges and opportunities.

In practice, designing and implementing an OS complying with *ServiceOS* is challenging: Some techniques are under development, some still have a long way to go, while some may be unreasonable or unrealistic. We do expect that the idea of *ServiceOS* can provide some implications for the research community and practitioners to build the resource-adaptive, intelligent, and robust software systems.

REFERENCES

[1] European Union. 2018. GDPR: The European Union's General Data Protection Regulation. Retrieved from https://gdpr-info.eu/.

[2] Martín Abadi, Andy Chu, Ian J. Goodfellow, H. Brendan McMahan, Ilya Mironov, Kunal Talwar, and Li Zhang. 2016. Deep learning with differential privacy. In *Proceedings of the ACM SIGSAC Conference on Computer and Communications Security*. 308–318.

[3] Alexandru Agache, Marc Brooker, Alexandra Iordache, Anthony Liguori, Rolf Neugebauer, Phil Piwonka, and Diana-Maria Popa. 2020. Firecracker: Lightweight virtualization for serverless applications. In *Proceedings of the 17th USENIX Symposium on Networked Systems Design and Implementation (NSDI'20)*. USENIX Association, 419–434. Retrieved from https://www.usenix.org/conference/nsdi20/presentation/agache.

[4] Jeremy Andrus, Christoffer Dall, Alexander Van't Hof, Oren Laadan, and Jason Nieh. 2011. Cells: A virtual mobile smartphone architecture. In *Proceedings of the 23rd ACM Symposium on Operating Systems Principles*, Ted Wobber and Peter Druschel (Eds.). ACM, 173–187.

[5] Tanzirul Azim, Oriana Riva, and Suman Nath. 2016. uLink: Enabling user-defined deep linking to app content. In *Proceedings of the 14th International Conference on Mobile Systems, Applications, and Services*, Rajesh Krishna Balan, Archan Misra, Sharad Agarwal, and Cecilia Mascolo (Eds.). ACM, 305–318.

[6] Johannes Bader, Andrew Scott, Michael Pradel, and Satish Chandra. 2019. Getafix: Learning to fix bugs automatically. *Proc. ACM Program. Lang.* 3, OOPSLA (2019), 159:1–159:27.

[7] Rudiger Birkner, Dana Drachsler-Cohen, Laurent Vanbever, and Martin Vechev. 2020. Config2Spec: Mining network specifications from network configurations. In *Proceedings of the 17th USENIX Symposium on Networked Systems Design and Implementation (NSDI'20)*. USENIX Association, 969–984. Retrieved from https://www.usenix.org/conference/nsdi20/presentation/birkner.

[8] Keith Bonawitz, Hubert Eichner, Wolfgang Grieskamp, Dzmitry Huba, Alex Ingerman, Vladimir Ivanov, Chloé Kiddon, Jakub Konecný, Stefano Mazzocchi, H. Brendan McMahan, Timon Van Overveldt, David Petrou, Daniel Ramage, and Jason Roselander. 2019. Towards federated learning at scale: System design. *CoRR* abs/1902.01046 (2019).

[9] Kevin Boos, Emilio Del Vecchio, and Lin Zhong. 2017. A characterization of state spill in modern operating systems. In *Proceedings of the 12th European Conference on Computer Systems*. 389–404.

[10] Kevin Boos and Lin Zhong. 2017. Theseus: A state spill-free operating system. In *Proceedings of the 9th Workshop on Programming Languages and Operating Systems*. 29–35.

[11] Facebook. 2013. Facebook Disaggregated Rack. Retrieved from http://goo.gl/6h2Ut.

[12] Earlence Fernandes, Oriana Riva, and Suman Nath. 2015. My OS ought to know me better: In-app behavioural analytics as an OS service. In *Proceedings of the 15th Workshop on Hot Topics in Operating Systems*, George Candea (Ed.). USENIX Association.

[13] Daniel Firestone, Andrew Putnam, Sambrama Mundkur, Derek Chiou, Alireza Dabagh, Mike Andrewartha, Hari Angepat, Vivek Bhanu, Adrian M. Caulfield, Eric S. Chung, Harish Kumar Chandrappa, Somesh Chaturmohta, Matt Humphrey, Jack Lavier, Norman Lam, Fengfen Liu, Kalin Ovtcharov, Jitu Padhye, Gautham Popuri, Shachar Raindel, Tejas Sapre, Mark Shaw, Gabriel Silva, Madhan Sivakumar, Nisheeth Srivastava, Anshuman Verma, Qasim Zuhair, Deepak Bansal, Doug Burger, Kushagra Vaid, David A. Maltz, and Albert G. Greenberg. 2018. Azure accelerated networking: SmartNICs in the public cloud. In *Proceedings of the 15th USENIX Symposium on Networked Systems Design and Implementation*, Sujata Banerjee and Srinivasan Seshan (Eds.). USENIX Association, 51–66.

[14] Peter Xiang Gao, Akshay Narayan, Sagar Karandikar, Joao Carreira, Sangjin Han, Rachit Agarwal, Sylvia Ratnasamy, and Scott Shenker. 2016. Network requirements for resource disaggregation. In *Proceedings of the 12th USENIX Symposium on Operating Systems Design and Implementation*, Kimberly Keeton and Timothy Roscoe (Eds.). USENIX Association, 249–264.

[15] Joseph M. Hellerstein, Jose M. Faleiro, Joseph Gonzalez, Johann Schleier-Smith, Vikram Sreekanti, Alexey Tumanov, and Chenggang Wu. 2019. Serverless computing: One step forward, two steps back. In *Proceedings of the 9th Biennial Conference on Innovative Data Systems Research*.

[16] Jon Howell, Bryan Parno, and John R. Douceur. 2013. Embassies: Radically refactoring the web. In *Proceedings of the 10th USENIX Symposium on Networked Systems Design and Implementation*, Nick Feamster and Jeffrey C. Mogul (Eds.). USENIX Association, 529–545.

[17] Gang Huang, Mengwei Xu, Felix Xiaozhu Lin, Yunxin Liu, Yun Ma, Saumay Pushp, and Xuanzhe Liu. 2017. ShuffleDog: Characterizing and adapting user-perceived latency of Android apps. *IEEE Trans. Mob. Comput.* 16, 10 (2017), 2913–2926.

[18] Jaehyun Hwang, Qizhe Cai, Ao Tang, and Rachit Agarwal. 2020. TCP ≈ RDMA: CPU-efficient remote storage access with i10. In *Proceedings of the 17th USENIX Symposium on Networked Systems Design and Implementation (NSDI'20)*. USENIX Association, 127–140.

[19] Eric Jonas, Johann Schleier-Smith, Vikram Sreekanti, Chia-che Tsai, Anurag Khandelwal, Qifan Pu, Vaishaal Shankar, Joao Carreira, Karl Krauth, Neeraja Jayant Yadwadkar, Joseph E. Gonzalez, Raluca Ada Popa, Ion Stoica, and David A. Patterson. 2019. Cloud programming simplified: A Berkeley view on serverless computing. *CoRR* abs/1902.03383 (2019).

[20] kernel.org. 2018. KConfig. Retrieved from https://www.kernel.org/doc/Documentation/kbuild/kcong-language.txt.

[21] Ana Klimovic, Yawen Wang, Patrick Stuedi, Animesh Trivedi, Jonas Pfefferle, and Christos Kozyrakis. 2018. Pocket: Elastic ephemeral storage for serverless analytics. In *Proceedings of the 13th USENIX Symposium on Operating Systems Design and Implementation*, Andrea C. Arpaci-Dusseau and Geoff Voelker (Eds.). USENIX Association, 427–444.

[22] Hsuan-Chi Kuo, Jianyan Chen, Sibin Mohan, and Tianyin Xu. 2020. Set the configuration for the heart of the OS: On the practicality of operating system kernel debloating. In *Proceedings of the ACM on Measurement and Analysis of Computing Systems (SIGMETRICS'20)* 4, X (2020), 4:1–4:25.

[23] Arezki Laga, Jalil Boukhobza, Michel Koskas, and Frank Singhoff. 2016. Lynx: A learning Linux prefetching mechanism for SSD performance model. In *Proceedings of the 5th Non-Volatile Memory Systems and Applications Symposium*. IEEE, 1–6.

[24] Angel Lagares Lemos, Florian Daniel, and Boualem Benatallah. 2016. Web service composition: A survey of techniques and tools. *ACM Comput. Surv.* 48, 3 (2016), 33:1–33:41.

[25] Ling Liu, Wenqi Cao, Semih Sahin, Qi Zhang, Juhyun Bae, and Yanzhao Wu. 2019. Memory disaggregation: Research problems and opportunities. In *Proceedings of the 39th IEEE International Conference on Distributed Computing Systems*. IEEE, 1664–1673.

[26] Xuanzhe Liu, Wei Ai, Huoran Li, Jian Tang, Gang Huang, Feng Feng, and Qiaozhu Mei. 2017. Deriving user preferences of mobile apps from their management activities. *ACM Trans. Inf. Syst.* 35, 4 (2017), 39:1–39:32.

[27] Xuanzhe Liu, Gang Huang, Qi Zhao, Hong Mei, and M. Brian Blake. 2014. iMashup: A mashup-based framework for service composition. *Sci. China Inf. Sci.* 57, 1 (2014), 1–20.

[28] Xuanzhe Liu, Huoran Li, Xuan Lu, Tao Xie, Qiaozhu Mei, Feng Feng, and Hong Mei. 2018. Understanding diverse usage patterns from large-scale appstore-service profiles. *IEEE Trans. Softw. Eng.* 44, 4 (2018), 384–411.

[29] Xuanzhe Liu, Yun Ma, Shuailiang Dong, Yunxin Liu, Tao Xie, and Gang Huang. 2017. ReWAP: Reducing redundant transfers for mobile web browsing via app-specific resource packaging. *IEEE Trans. Mob. Comput.* 16, 9 (2017), 2625–2638.

[30] Xuan Lu, Zhenpeng Chen, Xuanzhe Liu, Huoran Li, Tao Xie, and Qiaozhu Mei. 2017. PRADO: Predicting app adoption by learning the correlation between developer-controllable properties and user behaviors. *Proc. ACM Interact., Mob., Wear. Ubiq. Technol.* 1, 3 (2017), 79:1–79:30.

[31] Yun Ma, Ziniu Hu, Yunxin Liu, Tao Xie, and Xuanzhe Liu. 2018. Aladdin: Automating release of deep-link APIs on Android. In *Proceedings of the World Wide Web Conference on World Wide Web*. 1469–1478.

[32] Yun Ma, Xuanzhe Liu, Shuhui Zhang, Ruirui Xiang, Yunxin Liu, and Tao Xie. 2015. Measurement and analysis of mobile web cache performance. In *Proceedings of the 24th International Conference on World Wide Web*, Aldo Gangemi, Stefano Leonardi, and Alessandro Panconesi (Eds.). ACM, 691–701.

[33] Hong Mei and Yao Guo. 2018. Toward ubiquitous operating systems: A software-defined perspective. *IEEE Comput.* 51, 1 (2018), 50–56.

[34] Sandeep Neema, Rinku Parikh, and Suresh Jagannathan. 2019. Building resource adaptive software systems. *IEEE Softw.* 36, 2 (2019), 103–109.

[35] Simon Peter, Jialin Li, Irene Zhang, Dan R. K. Ports, Doug Woos, Arvind Krishnamurthy, Thomas E. Anderson, and Timothy Roscoe. 2016. Arrakis: The operating system is the control plane. *ACM Trans. Comput. Syst.* 33, 4 (2016), 11:1–11:30.

[36] Ardalan Amiri Sani and Thomas Anderson. 2019. The case for I/O-device-as-a-service. In *Proceedings of the Workshop on Hot Topics in Operating Systems*. ACM, 66–72.

[37] Yizhou Shan, Yutong Huang, Yilun Chen, and Yiying Zhang. 2018. LegoOS: A disseminated, distributed OS for hardware resource disaggregation. In *Proceedings of the 13th USENIX Symposium on Operating Systems Design and Implementation*, Andrea C. Arpaci-Dusseau and Geoff Voelker (Eds.). USENIX Association, 69–87.

[38] Vikram Sreekanti, Chenggang Wu, Xiayue Charles Lin, Johann Schleier-Smith, Jose M. Faleiro, Joseph E. Gonzalez, Joseph M. Hellerstein, and Alexey Tumanov. 2020. Cloudburst: Stateful functions-as-a-service. *CoRR* abs/2001.04592 (2020).

[39] Ion Stoica, Dawn Song, Raluca Ada Popa, David A. Patterson, Michael W. Mahoney, Randy H. Katz, Anthony D. Joseph, Michael I. Jordan, Joseph M. Hellerstein, Joseph E. Gonzalez, Ken Goldberg, Ali Ghodsi, David Culler, and Pieter Abbeel. 2017. A Berkeley view of systems challenges for AI. *CoRR* abs/1712.05855 (2017).

[40] Yusuke Suzuki, Shinpei Kato, Hiroshi Yamada, and Kenji Kono. 2014. GPUvm: Why not virtualizing GPUs at the hypervisor? In *Proceedings of the USENIX Annual Technical Conference*, Garth Gibson and Nickolai Zeldovich (Eds.). USENIX Association, 109–120.

[41] Shuo Tang, Haohui Mai, and Samuel T. King. 2010. Trust and protection in the Illinois browser operating system. In *Proceedings of the 9th USENIX Symposium on Operating Systems Design and Implementation*. 17–32.

[42] Helen J. Wang, Xiaofeng Fan, Jon Howell, and Collin Jackson. 2007. Protection and communication abstractions for web browsers in MashupOS. In *Proceedings of the 21st ACM Symposium on Operating Systems Principles*. 1–16.

[43] Liang Wang, Mengyuan Li, Yinqian Zhang, Thomas Ristenpart, and Michael M. Swift. 2018. Peeking behind the curtains of serverless platforms. In *Proceedings of the USENIX Annual Technical Conference*, Haryadi S. Gunawi and Benjamin Reed (Eds.). USENIX Association, 133–146.

[44] Marilyn Wolf. 2019. Machine Learning + Distributed IoT = Edge Intelligence. In *Proceedings of the 39th IEEE International Conference on Distributed Computing Systems*. 1715–1719.

[45] Mengwei Xu, Feng Qian, Qiaozhu Mei, Kang Huang, and Xuanzhe Liu. 2018. DeepType: On-device deep learning for input personalization service with minimal privacy concern. *Proc. ACM Interact., Mob., Wear. Ubiq. Technol.* 2, 4 (2018), 197:1–197:26.

[46] Mengwei Xu, Feng Qian, Mengze Zhu, Feifan Huang, Saumay Pushp, and Xuanzhe Liu. 2020. DeepWear: Adaptive local offloading for on-wearable deep learning. *IEEE Trans. Mob. Comput.* 19, 2 (2020), 314–330.

[47] Mengwei Xu, Mengze Zhu, Yunxin Liu, Felix Xiaozhu Lin, and Xuanzhe Liu. 2018. DeepCache: Principled cache for mobile deep vision. In *Proceedings of the 24th International Conference on Mobile Computing and Networking*, Rajeev Shorey, Rohan Murty, Yingying Jennifer Chen, and Kyle Jamieson (Eds.). ACM, 129–144.

[48] Neeraja J. Yadwadkar, Francisco Romero, Qian Li, and Christos Kozyrakis. 2019. A case for managed and model-less inference serving. In *Proceedings of the Workshop on Hot Topics in Operating Systems*. ACM, 184–191.

[49] Jiaqi Zhang, Lakshminarayanan Renganarayana, Xiaolan Zhang, Niyu Ge, Vasanth Bala, Tianyin Xu, and Yuanyuan Zhou. 2014. EnCore: Exploiting system environment and correlation information for misconfiguration detection. In *Proceedings of the Conference on Architectural Support for Programming Languages and Operating Systems*. 687–700.

[50] Kai Zhang, Bingsheng He, Jiayu Hu, Ze-ke Wang, Bei Hua, Jiayi Meng, and Lishan Yang. 2018. G-NET: Effective GPU sharing in NFV systems. In *Proceedings of the 15th USENIX Symposium on Networked Systems Design and Implementation*, Sujata Banerjee and Srinivasan Seshan (Eds.). USENIX Association, 187–200.

[51] Yiying Zhang and Yutong Huang. 2019. "Learned": Operating systems. *Oper. Syst. Rev.* 53, 1 (2019), 40–45.

[52] Danyang Zhuo, Kaiyuan Zhang, Yibo Zhu, Hongqiang Harry Liu, Matthew Rockett, Arvind Krishnamurthy, and Thomas E. Anderson. 2019. Slim: OS kernel support for a low-overhead container overlay network. In *Proceedings of the 16th USENIX Symposium on Networked Systems Design and Implementation*, Jay R. Lorch and Minlan Yu (Eds.). USENIX Association, 331–344.

Received May 2020; revised August 2020; accepted September 2020