Unifying serverless and microservice tasks with SigmaOS

Ariel Szekely, Adam Belay, Robert Morris, and M. Frans Kaashoek

MIT CSAIL

Abstract

Many cloud applications use both serverless functions, for bursts of stateless parallel computation, and container orchestration, for long-running microservices and tasks that need to interact. Ideally a single platform would offer the union of these systems' capabilities, but neither is sufficient to act as that single platform: serverless functions are lightweight but cannot act as servers with long-term state, while container orchestration offers general-purpose computation but instance start-up takes too long to support burst parallelism.

σOS is a new multi-tenant cloud operating system that combines the best of container orchestration and serverless in one platform with one API. σOS computations, called `procs`, can be long-running, stateful, and interact with each other, making them a good match for both serverless and microservice tasks. A key aspect of the σOS design is its *cloud-centric* API, which provides flexible management of computation, a novel abstraction for communication endpoints, σEPs—which allow `procs` of a tenant to communicate efficiently but prohibits `procs` from sending packets to other tenants—and a flexible naming system to name, for example, σEPs.

Quick `proc` start-up is important for serverless uses. A key enabling observation is that both serverless and microservice applications rely on cloud services for much of the work traditionally done by the local OS (e.g., access to durable storage and additional compute resources). σOS exploits this observation by providing only a small and generic local operating system image to each `proc`, which can be created much more quickly than a container orchestration instance since σOS need not install application-specific filesystem content or (due to σOS's σEPs) configure an isolated overlay network.

Microbenchmarks show that σOS can cold start a `proc` in 7.7 msec and can create 36,650 `procs` per second, distributing them over a 24-machine cluster. An evaluation of σOS with two microservice applications from DeathStarBench, a MapReduce application, and an image processing benchmark, shows that the σOS API supports both microservices and lambda-style computations, and provides better performance than corresponding versions on AWS Lambda and Kubernetes.

1 Introduction

A typical cloud tenant executes a variety of tasks on many machines: long-running analytics, short background job queues, fleets of web and web API servers, sharded database servers, and more. An important axis on which these workloads vary is the rapidity with which tasks come and go. Some are long-running stateful services, most recently typified by microservices. A contrasting workload is serverless functions [4, 17, 22, 33, 52], which are often invoked in large parallel bursts and tend to be short-lived. For convenience, this paper will refer to these two workload classes as "microservices" and "serverless."

Cloud applications require infrastructure to deploy software, manage execution and communication, and enforce isolation. For microservice-style workloads, container orchestration [10, 21, 34] works well: each instance provides the full facilities of a traditional operating system, and is thus quite general-purpose. These instances, however, are ill suited to serverless workloads. The core problem is that container orchestration instances start slowly. This is reasonable in static situations but a problem if instances come and go rapidly. One source of slowness is that each instance involves a user-level Linux installation: an isolated read/write file system populated from an application-specific Linux container image. Another is that, in order for cooperating instances to communicate, the start-up process must configure each with an IP address and a connection to an isolated overlay network. Isolation is important when a provider has many independent tenants. Finally, the mechanism for finding a machine with enough resources to execute a new instance is typically not fast enough for rapidly-initiated serverless functions.

An ideal platform would be able to initiate new work rapidly enough for serverless uses, but also provide enough flexibility to satisfy microservice applications. This would simplify applications that need both kinds of support, for example burst parallel functions that need to communicate [27, 28] or need to keep state [45, 61], or micro-services whose fleet size should vary with load. Such a platform would

SOSP '24,, November 4–6, 2024, Austin, TX, USA
ACM ISBN 979-8-4007-1251-7/24/11.
https://doi.org/10.1145/3694715.3695947

need fast instance start, powerful facilities for instances to interact, and isolation for both execution and communication.

σOS is a provider-hosted multi-tenant distributed operating system that addresses the above challenges. A key enabling insight is that cloud applications depend chiefly on cloud infrastructure, such as access to networked services, which the trend towards "cloud-native" applications [30] illustrates. Such applications can be designed to require little from the local machine beyond CPU, memory, and network access to cloud services. This simplifying assumption allows σOS to provide both the communication services that microservice workloads require and the creation speeds that serverless workloads need.

A unit of work in σOS is called a `proc`. σOS assumes that `proc`s will use cloud storage, and so does not go to the expense of creating per-`proc` local read/write file systems, but instead shares generic read-only file systems among multiple `proc`s. σOS provides a network addressing scheme (σEPs) for `proc`s that is more efficient than per-instance IP addresses, but allows communication only among the `proc`s of the same tenant. σOS isolates each `proc` in a lightweight "σcontainer," preventing the `proc` from using system calls not needed by the σOS API or runtime using Seccomp and AppArmor.

To help a tenant's `proc`s cooperate, σOS provides a per-tenant naming service (`named`) inspired by `etcd` [25] and Plan 9 [60]. `proc`s use `named` to register σEPs as well as to store configuration information and small items of shared and/or fault-tolerant state. As an example, σOS has a fast and scalable placement service that chooses a machine to execute each newly spawned `proc`; this service organizes itself via `named`.

Following Borg [71], σOS asks developers to mark each `proc` as either latency-critical (LC), with reserved CPU time and RAM, or best effort (BE). σOS schedules BE `proc`s as CPU and memory become available, either on the completion of previous BE `proc`s, or because LC `proc`s are temporarily under-using their reserved resources. The two classes capture a common distinction in cloud tasks, between services that are some or all of long-running, stateful, and in need of performance guarantees (LC); and tasks that are some combination of short-running, non-latency-critical, and in need of burst parallelism (BE).

Because σOS can create `proc`s rapidly, it is suitable for serverless tasks. Because `proc`s can communicate with each other, coordinate through the σOS name system, and can be long running, σOS is also suitable for stateful microservices, as well as offering this set of facilities to serverless tasks.

We have implemented a prototype of σOS in Go [1] on Linux. A microbenchmark of start times, critical for serverless workloads, shows that σOS has warm- and cold-start times lower than those of AWS Lambda, Docker, and Kubernetes; for cold-start 7.7 milliseconds vs 1.3 seconds, 2.7 seconds, and 1.1 seconds respectively. σOS start times are slower than Mitosis' 3.1 milliseconds [75], though σOS requires neither Linux kernel changes nor RDMA and is suitable for stateful long-running microservices with strong network isolation. A communication microbenchmark shows that σOS's σEPs deliver 48% lower per-packet latency and 14% higher throughput than Docker and Kubernetes overlay networks at the cost of higher dial latency. To demonstrate generality and application-level performance, we have implemented `proc`-based versions of a MapReduce library, the Hotel and Social Network web sites from DeathStarBench [29], and an image-processing service. σOS provides better throughput and latency for these applications than those provided by Docker and Kubernetes, while also delivering fairness among tenants and guaranteed resources for LC `proc`s (§6).

The contributions of this paper are: (1) the design of σOS, a new multi-tenant distributed operating system that supports both long-running stateful executions and short-lived serverless tasks; (2) σEPs, a novel abstraction which provides low-overhead network communication between `proc`s with strong security isolation; (3) a σcontainer implementation of `proc`s that provides strong isolation with quick start; and (4) an evaluation showing that σOS provides high throughput for both serverless and microservice workloads. The σOS source code is open-source and at available at `https://github.com/mit-pdos/sigmaos`.

2 Related work

Compared to prior work, σOS's main contribution is unifying support for serverless and microservices tasks in a single, cloud-centric platform with unique support for network isolation through σEPs and for `proc` isolation through σcontainers. This section discusses related work that σOS builds upon along the dimensions below. §6 measures the start times of several prior systems and compares them with σcontainers and σEPs.

Faster containers. Many techniques have been explored for fast start, since it is critical for burst-parallelism and transparent scaling of tenants' workloads. SAND [3] shares a single container among the serverless functions of a given application to ease local communication, and relies on process boundaries to isolate functions within the application. For many applications SAND's isolation is too weak. Consider an application which processes end-users' images with a serverless function invocation per image. In SAND, each image would be handled by a different Linux process in a shared container. If a malicious user could craft an image to exploit a bug in the application, they could take control of the Linux process handling the application's function invocation and corrupt or steal images of other users being handled by other Linux processes in the application's container. σOS makes this attack more challenging, as each image processing `proc` would run in a separate σcontainer with strong isolation.

SOCK [55] introduces "lean" containers specialized for serverless functions; they cannot communicate directly, which allows SOCK to avoid the cost of network isolation. σOS's σEPs allow `procs` to communicate directly with low overhead and strong isolation.

Particle [69] amortizes costs of configuring network namespaces and overlay networks when launching batches of containers on a single node. σOS's σEPs don't use network namespaces or overlay networks, and allow quick start on cold and warm nodes with strong network isolation between tenants.

Sidecar-less service meshes [16, 38] provide network isolation without per-container user-space proxies. However, they rely on overlay networks and network namespaces. σOS applications use σEPs, which provide networking isolation without these costs.

Isolation. AWS Lambda uses microVMs [2] to provide stronger isolation than containers with less overhead than a full VM; functions of the same tenant may share a microVM [73]. AWS Lambda targets serverless workloads and isn't suitable for microservice-style workloads, because Lambdas cannot communicate directly and cannot maintain long-term state (they may be terminated after 15 minutes).

Gvisor [32] reimplements much of Linux in user space while restricting the set of system calls made available by the underlying host. σOS σcontainers also restrict the set of system calls, to those needed to implement the σOS API.

LightVM [49] can start a VM with a unikernel and devices in 4ms. However, LightVM's reported start time is not comparable to σOS's 1.8ms, because while σOS provides strong network isolation, LightVM's start time does not include the cost of establishing network isolation (i.e., creating and configuring veth devices, network namespaces, and overlays).

Faasm [65] runs functions from different tenants in a shared WASM runtime, which allows for quick start times, but may be unsuitable for providers who are uncomfortable with the isolation guarantees of a shared WASM runtime [31] or the performance cost of WASM [39].

Fast application start. Application initialization after creating an isolated execution context (e.g., container or VM) is often another major bottleneck in fast start. Catalyzer [24] introduce `sfork` to start an application from a previously-checkpointed application image. Faasm, and SEUSS [13] also speed up application-start by checkpointing, respectively a WASM runtime or a unikernel. REAP [70] records and replays the working sets of functions to make starting from a snapshot faster. AWS Lambda [9] and FaasNET [72] reduce image fetch time with caching and efficient distribution. Mitosis [75] improves on Catalyzer's `sfork` by introducing `rfork`, which provides fast application start using remote fork over RDMA, reducing the need for caching. The above systems target serverless applications and aren't suitable for microservice-style workloads because these systems don't provide direct communication or network isolation. Like `rfork`, σOS leverages demand paging (but without modifying Linux and RDMA) and fetches pages from other nodes that have run the `proc` recently to start both serverless and microservice-style `proc` quickly.

Single system image. σOS's naming system inherits the idea of transparent access across a cluster from single-system image distributed systems [15, 20, 54, 57, 60, 68] but σOS targets cloud computing and provides a single-system image per tenant. σOS extends Plan9's 9P protocol [36], which is widely supported[1], and which allows σOS to be administered from the Linux command-line. σOS extends 9P with σEPs, watches to wait for a file to be created or removed (inspired by Chubby [11], `etcd` [25], and ZooKeeper [37]), and RPCs for services that don't fit a file system interface. σOS uses `etcd` to implement the root of a tenant's name space.

Schedulers. σOS takes inspiration from prior work on schedulers to allow the σOS scheduler to quickly schedule best-effort `procs` and guarantee resources to latency-critical `procs`. Like the Kubernetes scheduler [10, 64, 71], σOS uses a centralized scheduler and resource requests to schedule long-running computations carefully. σOS takes ideas from the distributed schedulers of Ray [74] and Sparrow [58] to schedule BE `procs` quickly and scalably. σOS's two-scheduler design resembles Mercury's hybrid scheduler [44]. Like Apollo [8], σOS uses real-time resource utilization to inform scheduling decisions.

Improving serverless. AWS Step Functions [5] and Azure Durable functions [6] simplify coordination and sharing among serverless functions. Recent research has eased restrictions on serverless functions by proxying communication [27, 28], by shuffling data efficiently [3, 42, 59, 61, 65], by resuming terminated functions [78], by allowing functions to communicate and maintain transient state [63], by providing exactly-once semantics despite re-execution [40, 43, 62, 77], by making function invocation fast [12, 17, 41, 53, 65], and by running functions close to the data and providing transactions [14, 46, 66].

σOS is suitable for both serverless and microservice tasks, since it provides fast start, communication among `procs`, and support for stateful services. Like container orchestration systems, the σOS scheduler allows `procs` to reserve resources for latency-critical microservice tasks, which serverless systems don't support. For example, although Faasm can run serverless functions from different tenants, developers cannot reserve resources for a faaslet that is long-running and latency critical and the Faasm scheduler runs all faaslets in round-robin, making it inappropriate for long-running, latency-critical microservices.

---

[1] Linux, Windows Subsystem for Linux (WSL), QEMU, and Gvisor support or use 9P.

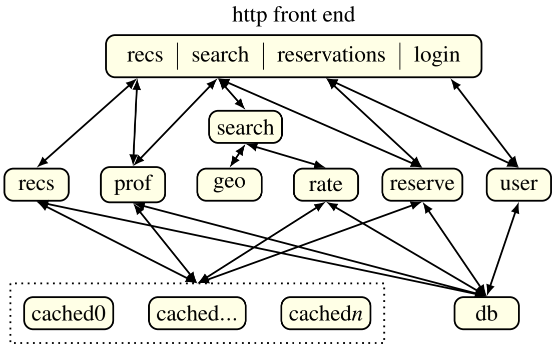

Figure 1: Each yellow box is a `proc` in the Hotel Web site. The cache service consists of one `proc` per shard.

3 σOS: a cloud-centric OS

σOS's goal is to provide tenants with a single platform that supports both serverless and microservice tasks and to multiplex the workloads from different tenants on the provider's servers. σOS's API caters to cloud applications' need to launch computations, to communicate among those computations, and to share storage; it aims to provide interfaces well-enough tailored to the cloud that traditional local OS interfaces are not needed. This section describes σOS and its interfaces from the perspective of the tenant, while the next section §4 describes how σOS is implemented on the provider's hardware.

3.1 `procs`: application execution units

Developers write their applications in terms of `procs`, which are executed by σOS. To illustrate how developers use `procs`, consider the following two running examples in this paper: `mr`, a MapReduce library, and `hotel`, a microservice application from DeathStarBench [29]. Both `mr` and `hotel` combine tasks that are best implemented as microservices with tasks that fit the serverless model.

The `mr` library creates a long-lived `proc` that coordinates the job. This coordinator spawns a serverless-style mapper `proc` for each shard of the input files and a reducer `proc` for each reducer bin. Current cloud offerings require developers to either use two platforms, a microservice platform to run the coordinator and a serverless platform to run the mappers and reducers, or run a coordinator together with a cluster of long-lived worker processes, which fails to take advantage of the elastic structure of MapReduce.

Methods	Description
`Spawn(descriptor)`	Queue `proc`, return `pid`
`Kill(pid)`	Kill `proc` `pid`
`WaitStart(pid)`	Wait until `pid` has started
`WaitExit(pid)`	Wait until `pid` has exited
`Started(pid)`	`pid` marks itself as started
`Exited(pid, status)`	`pid` marks itself as exited
`WaitKill(pid)`	`pid` waits for kill signal
`NewSigmaEP()` → `(Listener, SigmaEP)`	Create σEP
`Accept(Listener)` → `(Conn, SigmaEP)`	Accept connection
`Dial(SigmaEP)` → `Conn`	Connect σEP
`CloseEP(SigmaEP)` → `nil`	Close σEP
`Create(...)`, `Open(...)`, `Close(...)`, `Remove(...)`, `Rename(...)`, `Stat(...)`, `Read(...)`, `Write(...)`, `Lseek(...)`, `Watch(...)`,	Access files in realm
`AbsPath(path)`	Resolve `~local`
`OpenWatch(dir, func)`	Watch for changes in dir

Table 1: Summary of the σOS interface.

The `hotel` application creates a `proc` for the web front-end, a `proc` for each microservice, and a `proc` for each cache service shard. Figure 1 illustrates the `proc`-level organization of the `hotel` application. Some of the `hotel` microservices, like the sharded cache service and compute-intensive reservation service, benefit from the elasticity of serverless. Additionally, the `hotel` application may run periodic background jobs, such as image resizing or data analytics, which are a good fit for the serverless model. A developer who wishes to implement `hotel` on current cloud offerings would have to compose several cloud platforms to implement the different application tasks, and would be unable to make microservice tasks as elastic as serverless tasks.

In σOS, the tenant can use a single cloud platform and API to implement all tasks of `mr` and `hotel`. Long-lived microservices and short-running functions can both be implemented as `procs`. The tenant does not provision worker machines when implementing `mr` and `hotel`; σOS is in charge of choosing which of the provider's machines should run each `proc`.

Table 1 shows the interface with which σOS applications create and control `procs`. An application creates a new `proc` using `Spawn`, which returns a process identifier. The descriptor argument describes the desired attributes of the `proc`:

- the σOS name-system pathname of the binary to execute (§3.3).
- arguments to be passed to the `proc`.
- whether the `proc` is to be latency-critical (LC) or best-effort (BE), following Borg [71]. For example, a developer would likely specify BE for a map or reduce worker, and LC for a microservice `proc`.
- for LC `procs`, the amount of CPU power to reserve (typically chosen for peak expected load).
- for all `procs`, the amount of RAM to reserve.

```
func (c *Coord) runProc(p *Proc) {
  for {
    SigmaOS.Spawn(p)
    exitStatus := SigmaOS.WaitExit(p.GetPid())
    if exitStatus = SUCCESS {
      break
    } else if exitStatus = ERROR {
      // Need to retry, so continue in loop
    }
  }
  c.procDone(p)
}

func (c *Coord) runMR(procs []*Proc) {
  for _, p := range procs {
    go c.runProc(ch, p)
  }
}
```

Figure 2: Simplified version of the code a MapReduce coordinator might use to start map or reduce `procs`, wait for them, and re-start them on failure.

- optionally, a failure domain for situations where `procs` should execute on independently-failing machines.

`Spawn` adds the `proc` request to a queue managed by the σOS scheduler (§4). This queue gives the scheduler a view of demand, allows it to limit active load by deferring the start of some `procs`, and allows it to make informed decisions about how to distribute `procs` over machines. The caller of `Spawn` can wait until the scheduler starts its child by calling `WaitStart`. For example, `hotel` spawns microservices as LC `procs` and waits until they are started before accepting client requests. A `proc` signals to a parent that is running using `Started`.

A parent `proc` can wait until its child finishes using `WaitExit`, which returns an exit status provided by the child's call to `Exited`. σOS itself may also call `Exited` on behalf of a child `proc` if a machine crashes or a partition makes the `proc` unreachable; `WaitExit` returns an error in this case. For example, the `mr` coordinator waits for its mappers and reducers, as illustrated in Figure 2, and uses the returned exit status to decide whether any `procs` must be re-run due to failures.

3.2 Endpoints: `proc` communication

σOS must provide `procs` with high-performance network communication and must prohibit different tenants' `procs` from interfering with each other. σOS does so using a novel abstraction: σEPs. The σEP API, shown in Table 1, allows σOS to mediate connection setup, so that it can ensure that communication only occurs within each tenant.

When a server `proc` wishes to create an endpoint for incoming network traffic from other `procs` it calls `NewSigmaEP`, which requests σOS to create an σEP and a `Listener`. The σEP is an opaque shareable token identifying the new endpoint. The server `proc` calls `Accept` with the `Listener` to wait for incoming connections. Any `proc` of the same tenant can use the σEP to connect to the server by calling `Dial`. `Dial` returns an connection which can be used to exchange messages directly with the server `proc`.

```
lis, ep := SigmaOS.NewSigmaEP()
SigmaOS.Write("/s3/s3srv_3", ep.Marshal())
for true {
  conn := SigmaOS.Accept(lis)
  go HandleClientConn(conn)
}
```

Figure 3: Server code to create a σEP and register it under a name that clients can connect to.

```
b := SigmaOS.Read("/s3/s3srv_3")
ep := UnmarshalSigmaEP(b)
conn := SigmaOS.Dial(ep)
SendMsgToServer(conn)
```

Figure 4: Client code to retrieve a σEP given its name, and connect to the listening server.

An σEP is useable by any `proc` in the same realm. σEPs are assigned by σOS, however, so server `procs` must use the facilities in §3.3 to publish σEPs under well-known names.

3.3 Realms: per-tenant global name spaces

σEPs provide `procs` a low-level mechanism to create servers and establish connections. However, `procs` need a naming system in order to exchange σEPs, interact with each other, and share data. σOS supports this interaction with a per-tenant name space called a *realm*. A `proc` uses the σOS API to discover and access resources in a realm using pathnames (inspired by Plan 9 [60]). A `proc` in one realm cannot name or access σOS resources in other realms.

The root of a realm's file system is hosted by a name server called `named`, implemented using `etcd` [25], a Raft-replicated [56] persistent key/value store. σOS-provided services and `procs` extend the name space by hosting subtrees. As shown in Figure 3, a server which wishes to register a subtree creates a file in the name space containing its σEP object. When a `proc` traverses the name space and reaches the σEP link file, it will transparently connect to the hosting server using the σEP API and continue its traversal by communicating directly with the host of the subtree.

In this way, the logically global realm name space enables transparent access to a distributed set of resources, including filesystem-like services, named RPC services, proxies to storage systems, and storage for small configuration files. The namespace acts as a rendezvous so that `procs`, which are not directly aware of where they or other `procs` are physically executing, can find each other when needed. Furthermore, since a given pathname has the same meaning to all of a realm's `procs`, `procs` can directly exchange pathnames.

An example is σOS's `s3` proxy service, which exposes a tenant's Amazon's S3 buckets and keys in the tenant's name space. σOS stores binaries for procs in the σOS S3 bucket and reads/writes them using pathnames (e.g., `/s3/sigmaos/mr-mapper`).

To help applications spread load, a directory can list σEP link files for multiple proxy servers: `/s3/` can contain `s3srv_0`, `s3srv_1`, etc. Then a pathname with `~local` asks σOS to find a proxy on the same machine. Now, `mr` mappers can read input files (e.g., books from the S3 bucket `gutenberg`) using `/s3/~local/gutenberg/pg-being_ernest.txt` in parallel through local s3 proxies and avoid copying each input file twice over the network. The `~local` pathname component resolves a fundamental tension in the realm namespace between locality and location-obliviousness. `procs` which want to access local resources without having to know which machine they are physically running on can include `~local` in the pathnames they access.

As another example of the use of `~local`, σOS exposes per-realm scratch space on individual machines' local file systems under the pathname `/ux/<machine-name>/`; a `proc` can read and write scratch files on both its own machine and other machines with such pathnames. A `proc` that wishes to share a file in its local scratch space (e.g., `/ux/~local/a.txt`) with other `procs` first resolves the pathname to an absolute realm-global pathname with the `AbsPath(pathname)` API call, which translates `/ux/~local/...` to a global `/ux/<machine-name>` name. This ability to transparently access remote storage helps some data-intensive applications such as MapReduce [45, 48, 61, 67]. Uniform access via pathnames makes it straightforward for developers to choose between local and remote (e.g., S3) storage.

3.4 procs and failures

Many virtual machine and container systems re-start instances after a failure, which requires time-consuming persisting of configuration information at creation time. σOS does not restart `procs` in response to failure, and the σOS scheduling layer does not persist `proc` descriptors, which helps decrease creation cost. If a σOS scheduling component crashes while involved in spawning a `proc`, the spawn request may be lost. σOS guarantees that if such a failure prevented a `proc` from being spawned, `WaitStart`/`WaitExit` will return an error; the requesting `proc` can then re-issue the `Spawn` if appropriate.

σOS provides mechanisms to support fault-tolerant applications. A realm's `named` provides fault-tolerant storage, backed by `etcd`. `procs` can achieve fault-tolerance by storing critical state in `named`; if such a `proc` fails, another can be started (or already be waiting) to take over, and read the latest state from `named`. `named` uses `etcd`'s support for leader election to help fault-tolerant `procs` avoid split-brain (more than one `proc` thinking it is the active leader).

As an example, the `mr` library creates three coordinator `procs`. One is elected as the leader, and the other two are standbys. The leader stores its progress (e.g., which mappers have completed) in the directory `/mr/coord/`. If the leader crashes, one of the standbys takes over and picks up from where the crashed coordinator left off.

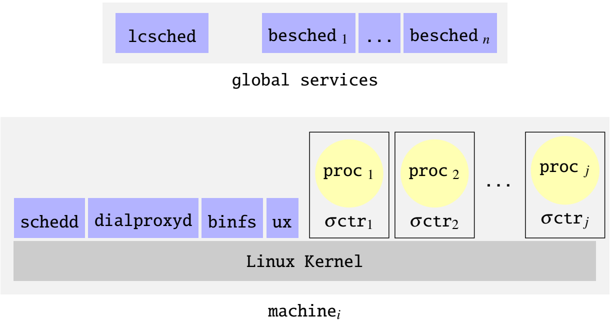

Figure 5: On each of a provider's machines, σOS runs a `schedd` (for creating σcontainers), `dialproxyd` (for mediating network connection setup), a `binfs` (for demand-paging `proc` binaries), and a `ux` (for exposing local storage). For distributed scheduling, each machine's `schedd` cooperates with one global `lcsched` for scheduling LC `procs` and several `bescheds` for scheduling BE `procs`. Not shown are provider `procs` that run in each realm (e.g., `named`, `s3`).

To survive a system-wide failure (e.g., all machines crash), σOS provides `initd`. An application can register a `proc` with `initd`; when σOS reboots, it restarts registered `procs`. For example, the `mr` library registers the coordinator `proc` with `initd`. `initd` stores its state persistently in `named`.

4 Implementing the σOS API

Implementing the σOS API poses three challenges: how to both isolate `procs` strongly and start them quickly; how to communicate efficiently with σEPs; and how to schedule BE and LC procs. σOS addresses these challenges using the components shown in Figure 5. `lcsched` and `besched` are global schedulers which place `procs` on machines. Each machine runs a local scheduler agent (`schedd`), a network isolation agent (`dialproxyd`), a `proc` binary server (`binfs`), and a server providing temporary local storage (`ux`). This section describes how σOS uses Linux primitives to implement these components.

4.1 Isolating procs with light-weight σcontainers

Each `proc` must be isolated to prevent it from disturbing provider infrastructure, other realms' `procs`, and (except as allowed by the σOS API) `procs` in the same realm. σOS isolates each `proc` with a σcontainer: a dedicated computing environment with low initialization overhead. What allows σcontainers to be light-weight is the fact that `procs` use only a small subset of the underlying operating system's facilities, relying instead on σOS' cloud-centric API. This allows σOS to avoid many expensive steps that traditional container systems must take to provide a full, private Linux environment. For example, σOS does not set up a network namespace, which would take around a hundred milliseconds, nor does it create an overlay file system, which would take around 5 milliseconds (plus the time to install files in the overlay file system). σcontainers provide `procs` with isolation as good as traditional containers but with faster start times.

Creating `procs`. An executing `proc` consists of a Linux process within an isolating σcontainer started by `schedd` with the following steps:

- **Namespaces:** `schedd` gives the `proc` private Linux namespaces for UTS, IPC, and PIDs. Because a `proc` doesn't need its own IP address (it uses σEPs and connections established via `dialproxyd`), the σcontainer doesn't include a network name space and overlay network. Further, because `procs` that need temporary local storage talk to the local `ux` server rather than making direct filesystem system calls, the σcontainer doesn't include an overlay file system.
- **Jail:** `schedd` jails the `proc`'s Linux process in a file system with just a few read-only configuration files and a few `/proc` pseudo-files. It mounts σOS's `binfs` read-only on `/mnt/binfs`; `binfs` is a FUSE [7] server through which the Linux kernel demand-pages the `proc`'s binary.
- **Seccomp:** `schedd` uses a seccomp filter to allow only system calls that allocate memory, create and manage threads, handle a few signals, access randomness, and manage timers. These calls are needed by the Go and Rust runtimes. The filter forbids networking system calls such as `socket`, `connect`, `bind`, `accept`, and `listen`, since connection setup occurs via `dialproxyd`.
- **AppArmor:** `schedd` drops all Linux capabilities and further restricts file system access using an AppArmor profile. This profile denies many uses of signals, and forbids access to directories in `/proc` other than the `proc`'s own `/proc` directory.
- **`cgroups`:** `schedd` provides performance isolation by assigning the σcontainer to a `cgroup`. `schedd` pre-creates and manages a pool of `cgroups` to move `cgroup` creation off the `proc` start path. `schedd` uses one `cgroup` to isolate each realm's BE `procs`, and another for each realm's LC `procs`. §4.3 describes how σOS configures `cgroups` to enforce resource reservations and utilize idle resources.

After this setup, the σcontainer calls `exec` with the path `/mnt/binfs/<binary-name>`. Linux demand-pages the binary via FUSE and `binfs`, allowing the `proc` to start quickly. The first time a `proc` binary runs in a σOS cluster, `binfs` reads its pages from S3. `binfs` caches these pages on the local disk, to speed future invocations of the same binary. The former situation is “cold start”; the latter “warm start”. σOS tracks where binaries have recently run so that `binfs` can take advantage of cached binaries on other machines in the σOS cluster.

σcontainer isolation. σOS restricts a `proc` to 67 system calls; the other 309 [47] are forbidden. For comparison, Docker's seccomp filter allows containers to use 352 system calls [23]; Kubernetes allows 340 [19]. Gvisor, which reimplements much of Linux in user space to reduce the number of host system calls a container requires, allows 55 system calls [76].

σcontainer startup time. An σcontainer is quicker to start than a traditional container because it doesn't unpack a container image, set up a network namespace (σOS uses σEPs), or create an overlay file system (`procs` cannot directly create files on the local host). Additionally, σOS performs some expensive operations in advance of running any `proc` (e.g, creating a pool of `cgroups`). Finally, σOS leverages Linux's demand paging via `binfs` to allow the `proc` to start without fetching the entire binary in advance.

4.2 σOS endpoints

σEPs allow σOS to control the setup of network connections among `procs` without the expense of setting up a virtual IP namespace for each `proc`. `dialproxyd` mediates all connection setup, both server-side and client-side, and verifies that each leads to a `proc` in the same realm. Once `dialproxyd` establishes and verifies a connection, it passes it to the `proc` with UNIX-domain message passing, and the `proc` directly reads and writes bytes on the socket. A σEP functions as a network address that `procs` can use without needing to know about host names, IP addresses, or port numbers.

Communication between `procs`. A typical scenario is that a `proc` offering a service creates a new σEP by calling `NewSigmaEP`, writes that σEP into a file in the `named` namespace, and waits for incoming connections with `Accept`. Clients find the relevant σEP from `named` and call `Dial`.

Both `Accept` and `Dial` are IPCs to the local `dialproxyd`, which makes the required kernel TCP socket calls. Initially the new connection connects the two local `dialproxyds`, which interact to verify that the two `procs`' realms are the same, and then passes the connection file descriptors to the `procs` via UNIX-domain message passing. After that, the two `procs` can communicate directly over the TCP connection without further involvement by `dialproxyd`.

Outgoing communication with external services. `dialproxyd` allows `procs` to connect to entities outside of σOS by creating and connecting to special *external* σEPs.

A malicious `proc` may try to create an external σEP that refers to a server `proc` in another realm. To prevent this, we expect the provider to deploy σOS in a private network with a known range of IP addresses. `dialproxyd` inspects external σEPs passed to it during connection setup to ensure that the IP addresses they contain lie outside its private range[2].

---

[2] An alternate design implemented in σOS, which does not rely on IP address verification, has the client's and server's `dialproxyd` exchange a

Incoming communication from external services. `procs` may need to accept connections from entities outside of σOS, such as HTTP clients. As is standard practice in Kubernetes and other container systems, we expect the client to configure a provider-managed IP endpoint and load-balancer to proxy connections to `procs`. The load-balancer would speak the `dialproxyd` identity verification protocol to ensure that external connections are delivered to the intended realm.

4.3 Scheduling `procs`

From the provider's perspective, σOS's job is to decide the placement of spawned `procs` onto the provider's machines, and, within each machine, to decide how to allocate CPU time and memory to running `procs`.

Placement. LC and BE `procs` have different placement requirements. LC `procs` must receive their guaranteed CPU and RAM, so σOS must take care that the sum of LC reservations on each machine is less than capacity. BE `procs` should use unreserved CPU and RAM, as well as reserved LC CPU currently left idle, though BE `procs` cannot be allowed to use RAM reserved by LC procs even if currently idle. Realms that have BE work should get roughly equal shares of total unreserved provider CPU, though a real deployment would likely use a different policy (e.g., based on resource pricing).

σOS places LC and BE `procs` with different mechanisms. When called for an LC `proc`, `Spawn` sends the descriptor to the provider's `lcsched`, a single provider-wide service. `lcsched` tracks, for each of the provider's machines, how much CPU power and RAM are currently reserved for existing LC `procs` on that machine. When a `Spawn` request arrives, `lcsched` chooses a machine with sufficient unreserved resources (if one exists) and forwards the request to the `schedd` on that machine; otherwise `lcsched` queues the `proc` request until a machine becomes available.

Because BE `procs` may be created at a high rate, σOS shards the work of BE placement over a set of `besched` servers running on a subset of the provider's machines. For a BE `proc`, `Spawn` sends the descriptor to a randomly selected `besched` server. That `besched` adds the descriptor to a private queue of `procs` waiting to run. Meanwhile, the `schedd` on each of the provider's machines monitors the machine's idle CPU time and idle RAM (less RAM reserved by LC procs), and if there are significant spare resources, sends a request to a randomly selected `besched`. That `besched` looks for a queued `proc` descriptor with a compatible memory request, giving each realm an equal chance. If the `besched` has nothing relevant queued, the machine's `schedd` asks a different `besched`.

Enforcement. `schedd` uses Linux `cgroups` to reserve CPU and memory for LC `procs`, and a combination of the `cgroups` network classifier and Linux's Traffic Control `tc` to prioritize LC `procs`' network traffic. `schedd` configures `cgroups` so that BE `procs` receive equal fractions of CPU reservations left idle by LC `procs`.

	Component	LOC
Core	proc/σcontainer	3,889
	`lcsched` `procq` `schedd`	2,340
	net/σEP	1,244
	file API	5,694
	realm: `named` `realmd`	2,419
	`s3` `ux` `db`	1,486
	boot	957
Libraries	`client`	4,059
	`server`	3,418
Applications	`hotel`	2,082
	`socialnet`	2,197
	`cache`	693
	`mr`	1,639
	`imgprocess`	447
	`kv`	1,931
Total		34,495

Table 2: Lines of Go code for σOS's components (excluding `etcd`, `etcd`'s Raft [26], and `protoc`-generated files for protocol buffers).

4.4 σOS prototype details

Most of σOS is written in Go; Table 2 shows each component's lines of code. To avoid the cost of exec-ing a Go binary, `schedd` uses a trampoline program when starting a new σcontainer, `exec-uproc`, written in Rust.

`proc` executables are statically-linked ELF files. Static linking results in larger binaries than dynamic linking, but allows the root file system to be generic, since it doesn't have to provide `proc`-specific files such as application shared libraries.

σOS supports interpreted languages like Python, which import modules at runtime dynamically, using the σOS API. The developer provides a statically-linked version of the interpreter of their choice (e.g., CPython) as the `proc` binary, and σOS provides a shim that intercepts the interpreter's accesses to module files and loads them via the σOS interface.

5 σOS applications

Table 2 lists the lines of code for each application. `hotel` (the running microservice example) and `socialnet` are ports of the corresponding DeathStarBench applications originally written for Kubernetes. `cache` is a sharded in-memory cache, much like `memcached`, used in `hotel` and `socialnet`. `mr` is the running example of the MapReduce library. `imgprocess` is an image processing service that spawns an LC coordinator `proc` to manage resizing images; for each image, it spawns a

---

cryptographically authenticated setup message to verify that connections internal to σOS are initiated via `dialproxyd`.

BE `proc` that stores its results in S3. The coordinator handles failures of `imgprocess` `procs`.

To demonstrate that σOS is complete enough to build fault-tolerant microservices, we built `kv`; it implements a linearizable, sharded, fault-tolerant in-memory key-value service. `kv` has groups of three `procs`; each group replicates its shards using `etcd`'s Raft library [26]. `kv`'s balancer can add a new group in response to changes in load; it moves shards between groups by spawning a "mover" `proc` for each shard with keys that must be moved. The balancer uses a realm's `named` to store the configuration for each epoch so that client `procs` can look up which group they should contact for a given shard. Clients set a watch on this configuration file to be alerted of a new configuration. The balancer itself has two standby `procs`, which will elect one as the new balancer if the current balancer fails.

Porting applications to σOS. σOS does not provide out-of-the-box backwards compatibility for existing serverless or microservice applications. However, in our experience, porting an application to σOS does not require major application rewriting; most σOS ports involve switching some function calls in the existing codebase with analogous functions from the σOS API. σOS supports cloud APIs and access to databases like MongoDB well via proxies (e.g., `s3`, `db`) and σEPs. Additionally, container- and serverless-structured applications are already broken up into execution units which map naturally onto `procs`.

σOS and σEP APIs (Table 1) provide replacements for most APIs that cloud applications use. σOS applications Read/Write files in the realm instead of using the local file system or the S3 Get/Put API, spawn `procs` instead of forking Linux processes or Lambdas, and access the network using σEP APIs like `NewSigmaEP` (analogous to `listen`) and `Dial`. The `NewSigmaEP` and `Dial` APIs are similar to those used to start servers and initiate network connections in Go and other high-level languages.

σOS developers can port web server front-ends like Nginx to σOS with σEPs (§4.2). σOS enables patterns similar to Kubernetes to in which web servers register public services with cloud provider load-balancers [35].

We ported `hotel` and `socialnet` from DeathStarBench, and most changes were "find-and-replace-all" operations. σOS versions use the realm for service discovery instead of a service registry and call `NewSigmaEP` instead of `listen` to start a server and accept connections. σOS's RPC library, implemented with σEPs, replicates the core of gRPC.

Applications that use a wide range of Linux system calls (e.g., shared memory) are harder to port to σOS. In our experience, most cloud applications don't do this. They rely on cloud APIs and are organized into containers and serverless functions.

Summary. There are two observations from the experience of implementing this set of applications. First, they can be implemented using just the σOS API. One reason is that the applications are cloud-centric and use few local OS services. Another is that σOS allows a provider to export services that applications need through proxies (e.g., `ux`, `s3`, `db`, etc.), which can be accessed using the σOS API. A final reason is that today's programming language like Go come with a large ecosystem of portable packages that allow application code to avoid much reliance on the local operating system.

The second observation is that σOS can support both microservice and serverless-style applications in a single framework. Furthermore, some applications combine both LC and BE `procs`. For example, in `kv`, the caching `procs` are LC while the mover `procs`, which copy shards between groups, are BE.

6 Evaluation

The primary goal of σOS is to provide a single API which seamlessly supports both serverless and microservice applications. To support short-running and burst-parallel serverless applications, σOS must start `procs` with low latency and high scheduling throughput. To support microservices, σOS must enable low-overhead communication between `procs` and allocate CPU to guarantee resource requests and achieve high utilization.

This section evaluates whether σOS achieves this goal by answering the following questions:

1. Do σcontainers allow fast `proc` start, and can σOS schedule `procs` at high throughput? (§6.1)
2. Do σEPs provide high-performance networking? (§6.2)
3. Do σOS applications perform as well as their serverless and container-based counterparts? (§6.3)
4. Can the σOS scheduler achieve good utilization with BE serverless tasks and LC microservice tasks, and provide fairness between tenants? (§6.4)

6.1 Fast `proc` start with σcontainers

We measure the start latency of σOS `procs` and compare it to several state-of-the-art platforms. For each platform, we time from the moment spawn is invoked until the first line of `main` executes. All but the AWS Lambda and Mitosis measurements use two AWS EC2 m6i.4xlarge VMs with 16 vCPUs backed by 2.9GHz Intel Ice Lake 8375C CPUs, 64 GiB of memory, up to 12.5Gbps network bandwidth, and up to 10 Gbps EBS burst bandwidth. The client invoking the function runs on one machine, and the platform runs on the other. We do not know what hardware AWS uses to run the Lambda functions. The Mitosis numbers are as reported in the Mitosis paper [75], which uses similar speed CPUs to the above configuration.

The `proc` being started is a simple BE Hello World program written in Rust to be able to measure σcontainer start times while excluding the 15 milliseconds that the Go runtime

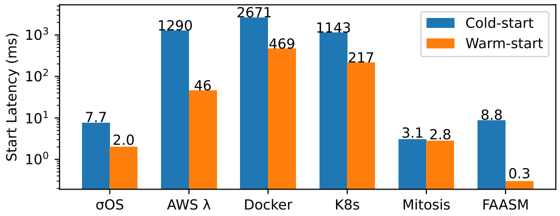

Figure 6: Start latency for a Hello World function on several platforms.

takes to start. Minimizing language runtime and application initialization time is an orthogonal problem, and existing techniques (like checkpoint-restore) are compatible with σOS.

Start latencies. Figure 6 shows the results. First we consider cold- and warm-start times for AWS Lambda, Docker containers, and Kubernetes (K8s) pods. All three systems isolate execution units using containers, and AWS Lambda additionally isolates functions with a MicroVM. Cold-starts in these systems require downloading and unpacking the container image (and, in AWS Lambda, additionally involve starting a MicroVM), whereas warm-starts exclude these costs.

σOS `procs` warm-start significantly faster than the other three systems' containers, because the other three systems start full containers with overlay file systems [73]. σOS' σcontainers (§4.1) avoid this cost because σOS offers neither custom file system contents nor direct file system write access; instead, `procs` access remote or (via the `ux` server) local storage using the σOS API. Additionally, σOS' `dialproxyd` (§4.2) design avoids expensive network isolation operations, and σOS moves expensive `cgroup` creation off the critical path. σOS' cold-start latency benefits from `binfs`, which demand-pages `proc` binaries over the network.

Mitosis [75] is a platform that provides fast start of containers for serverless functions—its containers and scheduler are not suited to stateful, long-running microservices—using a new remote-fork primitive, implemented using RDMA and a modified Linux kernel [75]. Mitosis thus sets a high bar for how fast serverless functions can be started. To compare with Mitosis, we benchmark σOS on an AWS EC2 instance with the same CPU clock rate as reported in the Mitosis paper[3]. Cold-starts in Mitosis involve remote-forking a running serverless function from a remote machine and creating a Mitosis "lean container" for the function. As expected, cold starts in σOS are slower than Mitosis (7.7 ms vs. 3.1ms), because σOS provides network isolation, σOS' `proc` binary demand-paging does not make use of specialized networking hardware (e.g., RDMA-capable NICs), and σOS runs atop an unmodified Linux kernel, which adds latency as the paged `proc` binary data moves back and forth across the user-kernel boundary.

		Time (ms)
Placement		0.42
σcontainer	Linux NS	0.28
	FS jail	0.42
	`seccomp`	0.46
	AppArmor	0.02
	`exec`	0.37
Total		1.97

Table 3: Breakdown of warm-start time for a Hello World BE `proc` written in Rust. Operations above the line are costs of `proc` placement, which includes time spent in the `besched` queue, whereas operations below the line are machine-local costs of σcontainer creation.

σOS component	Max Throughput (`procs`/sec)
`lcsched`	50,144
`besched` shard	53,306
`proc` start (1 machine)	1,590
`proc` start (24 machines)	36,650

Table 4: Maximum throughput of some components of `proc` creation on Cloudlab c220g5 machines.

Faasm [65], a serverless platform for WASM programs, achieves low start time by relying on the WASM runtime for isolation and for restricting system calls. Faasm cold-starts include the cost of downloading the WASM program from a local Redis instance, whereas warm-starts exclude this cost. σOS has slower warm-starts than Faaslets, because σOS pays the cost for separate address spaces and other OS isolation techniques such namespace isolation. σOS has faster cold-starts than Faasm because `binfs` demand-pages `proc` binaries, whereas Faasm downloads the entire WASM program before executing the function. We took the Faasm measurements without network isolation enabled; with network isolation, the cost of starting Faasm functions would likely be much higher.

Breakdown of `proc` start latency. Table 3 breaks down BE `procs`' warm-start latency, including the time required for `Spawn` to send the descriptor to `besched`, place the `proc` onto a `schedd` instance, create a σcontainer, and start the `proc`. The cost of a warm-start in σOS is dominated by σcontainer setup time, which consumes nearly 80% of the total.

Cold-starts, which occur when a realm runs a given `proc` binary on a machine for the first time, include the additional time for `binfs` to page the `proc` binary's first few pages over the network.

Spawn throughput. There are two main tasks that might limit the throughput at which σOS can spawn new BE `procs`:

---

[3] We were unable to access the RDMA hardware and software setup required to run Mitosis and so report the Hello World benchmark numbers from the Mitosis paper.

the sharded `besched` placement service (§4.3), and the time required to create the `proc` on the selected machine (§4.1). Both throughputs can be scaled up (by running more `besched` shard servers, and installing more machines, respectively). The following experiments examine quickly a single `besched` can process spawn requests, how quickly a single machine can create `procs`, and the end-to-end achievable `proc` start throughput on a cluster of 24 machines. Table 4 summarizes the results.

To find how many BE `Spawns` per second a single `besched` can handle, we run a σOS cluster consisting of 24 CloudLab c220g5 machines, scheduled by a single `besched`. An open-loop client on a remote machine `Spawns` "dummy" BE `procs` at a fixed rate for 10 seconds. The BE `procs` never actually run, but they traverse the full BE `proc` scheduling path; they are spawned onto the `besched`, which places them onto a machine's `schedd`, which ignores them. The maximum client `Spawn` rate at which the single `besched` can keep up is 53,306 per second.

Once `besched` has placed a `proc` on a machine, that machine's `schedd` needs to create a σcontainer and start the `proc`. To evaluate the throughput limits of this task on a single machine, an open-loop client `Spawns` BE Rust Hello World `procs` at a set rate, regardless of how quickly σOS responds. For this benchmark, a single `besched`, running on a dedicated machine, schedules the Hello World `procs` onto a single `schedd` running on a different machine.

`proc` start throughput saturates and queues begin to build up once the `Spawn` rate exceeds 1,590 `procs` per second. The bottleneck is Linux mount namespace creation which occurs every time a σcontainer is created and involves taking a global lock in Linux.

We evaluate the end-to-end sustainable `proc` start throughput on a cluster of 24 CloudLab c220g5 machines by running an open-loop client which `Spawns` BE Rust Hello World `procs` at a set rate, independently of how quickly σOS starts them. A single `besched` places the `procs` across the cluster. σOS is able to start up to 36,650 `procs` per second before the `Spawn` rate exceeds the start rate.

Even as the `Spawn` rate approaches the maximum start rate, σOS' median and 90% scheduling latencies remain low. σOS achieves 36,650 `proc` starts per second with p50 start latency of 5.8 ms and a 90% start latency of 11.6 ms. To put σOS' `proc` start throughput in perspective, we compare to Mitosis, which can fork 10,000 containers across multiple machines in one second. Mitosis' container fork rate translates to 97 container starts per core per second, and σOS' peak `proc` start rate translates to 76 `proc` starts per core second. σOS achieves this performance without kernel changes or RDMA.

Summary: σOS starts `procs` quickly and with high throughput. As a result σOS is a good fit for applications structured as many short-running `procs`. Finer-grained `procs` give providers more frequent opportunities to rebalance resource allocations, and to smooth out short drops in resource utilization as realms' application load varies.

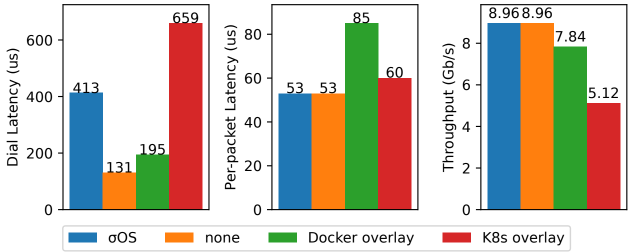

Figure 7: Different network isolation strategies' network latency and throughput between containers communicating across Cloudlab c220g5 machines. Measurements are taken from Go server and client programs which communicate via TCP. The implementations are identical across all isolation strategies, with the exception that the σOS client and server communicate via the σEP API.

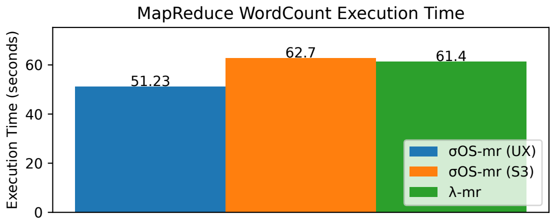

Figure 8: End-to-end execution time of MapReduce WordCount on σOS and on λ-`mr` from a warm-start, with 10GB of input. Both σOS-`mr` and λ-`mr` store input and output files in S3. σOS (`ux`) stores intermediate output files in `ux`, σOS's local file server.

6.2 σEP performance

To determine the performance of σEPs, we benchmark network throughput and latency between two Cloudlab c220g5 nodes using different network isolation techniques. The measurements use a simple Go client and server that communicate with TCP; the σOS version does this via the σEP API. Dial latency is the time to establish a new TCP connection. Per-packet latency is half a TCP round trip. Throughput is the bit rate at which the client can write 1MB buffers to the server. All measurements are the average of 1000 trials.

Figure 7 compares σOS' σEP-isolated network performance to no network isolation, isolation using Docker overlays, and isolation using Kubernetes (K8s) overlays. The σOS `dialproxyd`-mediated `Dial` protocol makes connecting more expensive than without network isolation. Once connected, σEPs provide the same latency and throughput as no network isolation, since `dialproxyd` sends the TCP socket to the `proc`, which reads and writes directly. Docker and K8s, in contrast, forward all connection data through a local proxy process.

6.3 σOS application performance

To evaluate application-level performance with σOS, we compare σOS-`mr` to Corral [18] (labeled λ-`mr`), a serverless

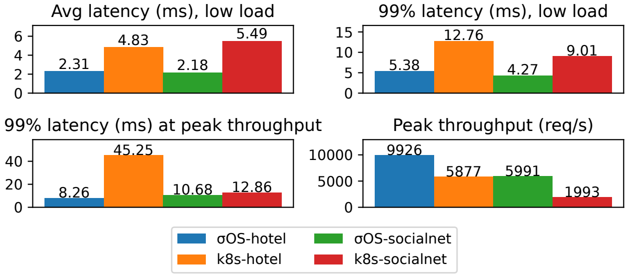

Figure 9: Latency and throughput of σOS-`hotel` and K8s-`hotel`, and σOS-`socialnet` and K8s-`socialnet` . The low-load 99% latency is measured at a request rate of 1000 Req/s. Peak throughput is the maximum request throughput with median request latency below 10ms.

MapReduce framework for AWS Lambda, σOS-`hotel` to K8s-`hotel`, and σOS-`socialnet` to K8s-`socialnet` .

σOS-`mr` vs. λ-`mr`. We run σOS-`mr` on an AWS Virtual Private Cloud (VPC) composed of 16 EC2 `t3.xlarge` VM instances. Each instance has 4 vCPUs, 16GiB of memory, and a 20GiB EBS volume. Each instance has up to 5Gbps network burst bandwidth, and 2,085Mbps EBS burst bandwidth. To make the comparison fair, we provision λ-`mr`'s lambdas with 1760MB of memory, which gives them the equivalent of 1 vCPU, and disable 2 vCPUs on each of σOS's 4-core VMs. Since λ-`mr` starts 32 lambdas in parallel for both the map and reduce phase, this gives λ-`mr` and σOS-`mr` the same resources when σOS-`mr` runs on 16 machines. Both σOS-`mr` and λ-`mr` are written in Golang, and pay for the cost of the Go runtime (e.g., garbage collection). The input is 10GB from a snapshot of English HTML Wikipedia pages.

The input and final output files for both versions of MapReduce are stored in S3. For λ-`mr`, the intermediate files are also stored in S3. To evaluate the benefit of σOS-`mr`'s transparent access to σOS-exposed local storage, we run σOS-`mr` in two configurations: σOS (`s3`) and σOS-`mr` (`ux`); the latter writes and then remotely reads intermediate files via `ux`, σOS' local file server.

Figure 8 shows the results. σOS-`mr` performs comparably to λ-`mr` with intermediate files stored in S3, and 1.2× faster with intermediate files stored on local disk (really EBS) via `ux`. `ux` helps because it has higher throughput than S3. σOS-`mr` can transparently take advantage of `ux`'s local scratch space exposed by just changing the application's intermediate file pathnames.

σOS-`hotel` vs. K8s-`hotel` and σOS-`socialnet` vs. K8s-`socialnet`. We compare σOS-`hotel` to K8s-`hotel` and σOS-`socialnet` to K8s-`socialnet` on 8 Cloudlab `c220g5` machines. Each machine has two Intel Xeon Silver 4114 10-core CPUs at 2.20 GHz, 192GB of memory, and a 10Gb Intel X520-DA2 NIC. To induce contention for resources and force σOS and Kubernetes to place microservices on multiple hosts, all but 4 CPUs are disabled on each machine.

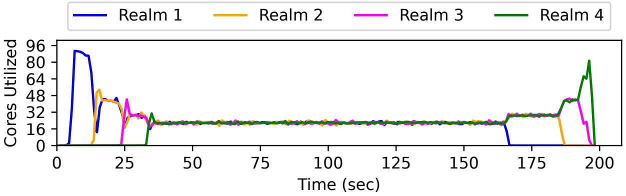

Figure 10: CPU utilization of 4 realms running BE `imgprocess` jobs starting at different times, on a 24 AWS `t3.xlarge` VM VPC scheduled by 2 `besched` shards. σOS dynamically resizes each realm's CPU allocation in response to load in order to give each realm an equal share of the cores. When one job's load decreases, σOS quickly reallocates its idle cores to the realms with remaining work.

Figure 9 shows the latency (average and 99%) and the peak throughput using the workload generator from DeathStarBench, running the search workload. Peak throughput is the maximum client request throughput for which median latency stays below 10ms. σOS yields 42% lower 99% latency under low load for σOS-`hotel` and 47% lower tail latency for σOS-`socialnet` under low load. σOS enables `hotel` and `socialnet` to achieve 1.68× and 3.01× higher peak throughput than Kubernetes.

The σOS implementations of `hotel` and `socialnet` outperform the Kubernetes implementations because σEP communication is more efficient than Kubernetes overlay networking, as shown in §6.2.

Summary: σOS's abstractions perform well and allow stateless- and microservices-style applications to achieve high performance.

6.4 Scheduling `procs`

σOS's scheduler goals include sharing resources fairly among realms with BE work and achieving high utilization by allowing BE `procs` to use resources reserved but left idle by LC `procs` (§4.3). This section evaluates whether σOS meets its scheduling objectives through case studies in which σOS multiplexes multiple realms' BE workloads and realms with BE and LC workloads.

Fair sharing between realms with BE `procs`. We evaluate σOS' ability to divide resources equally between realms with BE work using `imgprocess`, a representative extract, transform, and load (ETL) application [50, 51]. `imgprocess` involves a short `proc` to process each 50KB input image. Of the 665ms average time for each `imgprocess` `proc`, 503ms are CPU-intensive work, and the remainder is I/O. `imgprocess` input and output images are stored in `ux` since `ux` provides high-throughput access to storage.

Figure 10 shows four realms, each running a job consisting of many `imgprocess` `procs`, and illustrates how σOS shifts resources between them. The graph shows the CPU utilization of each realm as a function of time. There are 24 AWS `t3.xlarge` machines, each with four cores, and 2 `besched` shards which distribute BE `procs` among them. Realm 1's

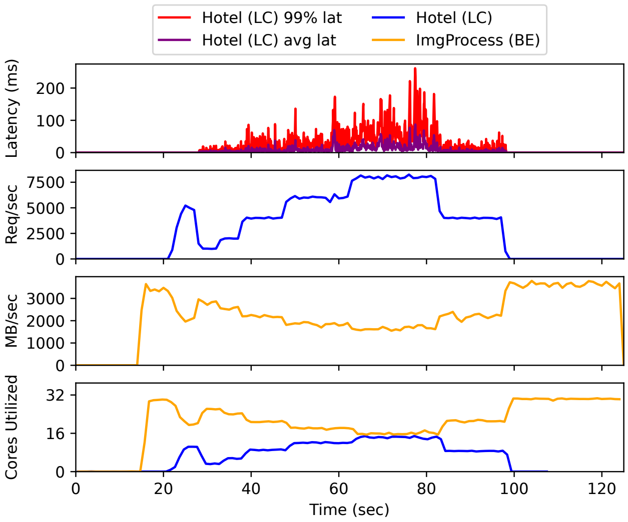

Figure 11: σOS multiplexing two realms across a cluster of 8 Cloudlab c220g5 machines each with 4 cores available, under varying load. One realm runs σOS-`hotel`, and the other runs `imgprocess`. As σOS-`hotel`'s load increases, it regains its reserved CPU time from the BE `imgprocess` realm. When LC load drops, the BE realm is allowed to use the idle CPU time.

job starts at time zero, Realm 2 starts 10 seconds later, Realm 3 starts at 20 seconds, and Realm 4 starts at 30 seconds.

Each realm `Spawns` `imgprocess` `procs` at a fixed rate for 30 seconds, and then stops and waits for all its `procs` to complete. `imgprocess` `procs` are spawned with a 1.5GB memory request, and `procs` which cannot fit on any machine once spawned are queued at `besched`.

σOS gives all 96 cores' worth of CPU time to Realm 1. The graph shows that Realm 1 uses almost all of the CPU time, with the shortfall lost to I/O. As Realm 2 starts to enqueue `procs`, σOS begins dequeueing `procs` from each realm in equal shares. The CPU utilizations of Realm 1 and Realm 2 quickly equalize because `imgprocess` `procs` are short-running, giving many resource rebalancing opportunities to the `besched` cluster-level scheduler. Together with the `cgroups` policies set up by `schedd`, this results in a rapid redistribution of cores between the realms as `procs` exit and release their memory requests.

Similarly, σOS redistributes cluster CPU time when Realms 3 and 4 start, giving each an equal share. As each Realm's work ends, σOS gives more CPU time to the remaining realms.

Summary: σOS automatically manages the allocation of compute resources to realms.

Guaranteeing LC reservations. Figure 11 shows a situation in which one realm runs a sequence of `imgprocess` `procs` while another runs the σOS-`hotel` site. The σOS-`hotel` `procs` are all LC, and the `imgprocess` procs are all BE. The x-axes show time; σOS-`hotel` receives requests from time 20 until time 100, but receives a burst from times 60 to 80. The upper graph shows the latency with which the σOS-`hotel` answers web requests; the second graph shows the σOS-`hotel` request rate; the third shows `imgprocess` throughput; and the bottom graph shows how σOS divides cores between the two realms.

When σOS-`hotel` is under low load, `imgprocess` receives the majority of the cores in the cluster and can achieve high processing throughput. In fact, those cores are reserved for the σOS-`hotel` `procs` (since they are LC), and merely borrowed for `imgprocess` when σOS-`hotel` `procs` underutilize their reservations. When σOS-`hotel` input load increases, σOS' `cgroups` configuration causes Linux to shift reserved CPU time back from `imgprocess` to σOS-`hotel`. Although `cgroups` are able to preferentially give σOS-`hotel` CPU time when it is under high load, the Linux `cgroup` API limits the degree to which one process can be prioritized over another. This causes σOS-`hotel` to experience occasional latency spikes under periods of particularly high load.

Summary: σOS's resource management helps utilization: the σOS-`hotel` application's reserved resources are available when needed, without preventing other tasks from using those resources when the σOS-`hotel` load is low.

7 Discussion and future work

The σOS prototype lacks features that industrial orchestration systems have; for example, σOS has limited support for authorization within a realm: AWS S3 tokens (e.g., to allow a `proc` to limit a child `proc` to specific S3 buckets) and the traditional ACLs that σP inherits from 9P. As another example, the prototype considers memory and CPU resources, but not other resources such as GPUs.

For some services, it will be too much work or infeasible to port them to the σOS API, because, for example, their code base is large or they rely on specific system calls. Such services can be incorporated in σOS through proxies such as `s3` and `db` or by running the service with its own container image and modifying the service to advertise itself using the σOS libraries in a realm's namespace.

8 Conclusion

This paper presented a multi-tenant cloud operating system, σOS, that supports both microservices and serverless functions by combining the best features of container orchestration systems and serverless frameworks. σOS provides developers with a cloud-centric API that allow developers to structure their applications using `procs` and that provides a shared namespace per realm, which hides machine boundaries and allows `procs` to easily communicate and coordinate. σOS can efficiently multiplex the `procs` of different tenants on the provider's hardware and responds automatically to shifts in load. Finally, by restricting `procs` to the σOS interface,

σOS can use σcontainers to cold start procs rapidly (in 7.7 ms) with strong isolation.

Acknowledgments

Thanks to our shepherd Ryan Huang, the anonymous reviewers of OSDI 2023, OSDI 2024, and SOSP 2024, Yizheng He, Hannah Gross, the students of 6.828 seminar fall 2023, and Gideon Witchel. Ariel Szekely was supported by an NSF Graduate Research Fellowship.

References

[1] The Go Programming Language. `https://golang.org/`.

[2] Alexandru Agache, Marc Brooker, Alexandra Iordache, Anthony Liguori, Rolf Neugebauer, Phil Piwonka, and Diana-Maria Popa. Firecracker: Lightweight virtualization for serverless applications. In *17th USENIX Symposium on Networked Systems Design and Implementation, NSDI 2020, Santa Clara, CA, USA, February 25-27, 2020*, pages 419–434, 2020.

[3] Istemi Ekin Akkus, Ruichuan Chen, Ivica Rimac, Manuel Stein, Klaus Satzke, Andre Beck, Paarijaat Aditya, and Volker Hilt. SAND: Towards High-Performance serverless computing. In *2018 USENIX Annual Technical Conference (USENIX ATC 18)*, pages 923–935, Boston, MA, July 2018. USENIX Association. ISBN 978-1-939133-01-4. URL `https://www.usenix.org/conference/atc18/presentation/akkus`.

[4] Amazon. AWS lambda. `https://aws.amazon.com/lambda/`.

[5] AWS. Aws step functions. `https://aws.amazon.com/tutorials/create-a-serverless-workflow-step-functions-lambda/`, 2024.

[6] Microsoft Azure. Azure durable functions. `https://learn.microsoft.com/en-us/azure/azure-functions/durable/durable-functions-overview?tabs=in-process%2Cnodejs-v3%2Cv1-model&pivots=csharp`, 2024.

[7] Ashish Bijlani and Umakishore Ramachandran. Extension framework for file systems in user space. In *Proceedings of the 2019 USENIX Annual Technical Conference*, pages 121–134, Renton, WA, July 2019.

[8] Eric Boutin, Jaliya Ekanayake, Wei Lin, Bing Shi, Jingren Zhou, Zhengping Qian, Ming Wu, and Lidong Zhou. Apollo: Scalable and coordinated scheduling for cloud-scale computing. In *Proceedings of the 11th USENIX Conference on Operating Systems Design and Implementation*, OSDI'14, page 285–300, USA, 2014. USENIX Association. ISBN 9781931971164.

[9] Marc Brooker, Mike Danilov, Chris Greenwood, and Phil Piwonka. On-demand container loading in AWS lambda. In *2023 USENIX Annual Technical Conference (USENIX ATC 23)*, pages 315–328, Boston, MA, July 2023. USENIX Association. ISBN 978-1-939133-35-9. URL `https://www.usenix.org/conference/atc23/presentation/brooker`.

[10] Brendan Burns, Brian Grant, David Oppenheimer, Eric Brewer, and John Wilkes. Borg, Omega, and Kubernetes. *ACM Queue*, 14(1), 2016.

[11] Mike Burrows. The Chubby lock service for loosely-coupled distributed systems. In *Proceedings of the 7th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, Seattle, WA, November 2006.

[12] Sergey Bykov, Alan Geller, Gabriel Kliot, James R. Larus, Ravi Pandya, and Jorgen Thelin. Orleans: Cloud computing for everyone. In *Proceedings of the 2nd ACM Symposium on Cloud Computing*, SOCC '11, New York, NY, USA, 2011. Association for Computing Machinery. ISBN 9781450309769. doi: 10.1145/2038916.2038932. URL `https://doi.org/10.1145/2038916.2038932`.

[13] James Cadden, Thomas Unger, Yara Awad, Han Dong, Orran Krieger, and Jonathan Appavoo. SEUSS: Skip redundant paths to make serverless fast. In *Proceedings of the Fifteenth European Conference on Computer Systems*, EuroSys '20, New York, NY, USA, 2020. Association for Computing Machinery. ISBN 9781450368827. doi: 10.1145/3342195.3392698. URL `https://doi.org/10.1145/3342195.3392698`.

[14] Michael J. Cafarella, David J. DeWitt, Vijay Gadepally, Jeremy Kepner, Christos Kozyrakis, Tim Kraska, Michael Stonebraker, and Matei Zaharia. A polystore based database operating system (DBOS). In *Heterogeneous Data Management, Polystores, and Analytics for Healthcare - VLDB Workshops, Poly 2020 and DMAH 2020, Virtual Event, August 31 and September 4, 2020, Revised Selected Papers*, volume 12633 of *Lecture Notes in Computer Science*, pages 3–24. Springer, 2020.

[15] David R. Cheriton. The V distributed system. *Communications of the ACM*, 31(3):314–333, March 1988.

[16] Cilium. Cilium service mesh. `https://cilium.io/use-cases/service-mesh/`.

[17] Cloudflare. Cloudflare workers. `https://workers.cloudflare.com/`.

[18] Ben Congdon. Corral. `https://github.com/bcongdon/corral`.

[19] Containerd. Kubernetes/containerd default seccomp filter. `https://github.com/containerd/containerd/blob/36cc874494a56a253cd181a1a685b44b58a2e34a/contrib/seccomp/seccomp_default.go`, 2024.

[20] Partha Dasgupta, Richard J. LeBlanc, Mustaque Ahamad, and Umakishore Ramachandran. The clouds distributed operating system. *Computer*, 24(11): 34–44, November 1991. ISSN 0018-9162. doi: 10.1109/2.116849. URL `https://doi.org/10.1109/2.116849`.

[21] Docker. Docker. `https://www.docker.com/`, 2023.

[22] Docker. Docker swarms. `https://docs.docker.com/engine/swarm/`, 2023.

[23] Docker. Docker default seccomp filter. `https://github.com/moby/moby/blob/master/profiles/seccomp/default.json`, 2024.

[24] Dong Du, Tianyi Yu, Yubin Xia, Binyu Zang, Guanglu Yan, Chenggang Qin, Qixuan Wu, and Haibo Chen. Catalyzer: Sub-millisecond startup for serverless computing with initialization-less booting. In *Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '20, page 467–481, New York, NY, USA, 2020. Association for Computing Machinery. ISBN 9781450371025. doi: 10.1145/3373376.3378512. URL `https://doi.org/10.1145/3373376.3378512`.

[25] etcd.io. Etcd. `https://github.com/etcd-io/`, 2023.

[26] etcd.io. Etcd Raft. `https://github.com/etcd-io/etcd/`, 2023.

[27] Sadjad Fouladi, Riad S. Wahby, Brennan Shacklett, Karthikeyan Vasuki Balasubramaniam, William Zeng, Rahul Bhalerao, Anirudh Sivaraman, George Porter, and Keith Winstein. Encoding, fast and slow: Low-latency video processing using thousands of tiny threads. In *14th USENIX Symposium on Networked Systems Design and Implementation (NSDI 17)*, pages 363–376, Boston, MA, March 2017. USENIX Association. ISBN 978-1-931971-37-9. URL `https://www.usenix.org/conference/nsdi17/technical-sessions/presentation/fouladi`.

[28] Sadjad Fouladi, Francisco Romero, Dan Iter, Qian Li, Shuvo Chatterjee, Christos Kozyrakis, Matei Zaharia, and Keith Winstein. From laptop to lambda: Outsourcing everyday jobs to thousands of transient functional containers. In *Proceedings of the 2019 USENIX Conference on Usenix Annual Technical Conference*, USENIX ATC '19, page 475–488, USA, 2019. USENIX Association. ISBN 9781939133038.

[29] Yu Gan, Yanqi Zhang, Dailun Cheng, Ankitha Shetty, Priyal Rathi, Nayan Katarki, Ariana Bruno, Justin Hu, Brian Ritchken, Brendon Jackson, Kelvin Hu, Meghna Pancholi, Yuan He, Brett Clancy, Chris Colen, Fukang Wen, Catherine Leung, Siyuan Wang, Leon Zaruvinsky, Mateo Espinosa, Rick Lin, Zhongling Liu, Jake Padilla, and Christina Delimitrou. An open-source benchmark suite for microservices and their hardware-software implications for cloud & edge systems. In *Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '19, page 3–18, New York, NY, USA, 2019. URL `https://github.com/delimitrou/DeathStarBench`.

[30] Dennis Gannon, Roger Barga, and Neel Sundaresan. Cloud-native applications. *IEEE Cloud Computing*, 4 (5):16–21, 2017. doi: 10.1109/MCC.2017.4250939.

[31] Daniel Genkin, Lev Pachmanov, Eran Tromer, and Yuval Yarom. Drive-by key-extraction cache attacks from portable code. In *Applied Cryptography and Network Security: 16th International Conference, ACNS 2018, Leuven, Belgium, July 2-4, 2018, Proceedings*, page 83–102, Berlin, Heidelberg, 2018. Springer-Verlag. ISBN 978-3-319-93386-3. doi: 10.1007/978-3-319-93387-0_5. URL `https://doi.org/10.1007/978-3-319-93387-0_5`.

[32] Google. Open-sourcing gvisor, a sandboxed container runtime. https://cloud.google.com/blog/products/identity-security/open-sourcing-gvisor-a-sandboxed-container-runtime, May 2018.

[33] Google. Cloud functions. `https://cloud.google.com/functions`, 2023.

[34] Google. Kubernetes. `http://kubernetes.io/`, 2023.

[35] Google. Kubernetes. `https://kubernetes.io/docs/tasks/access-application-cluster/create-external-load-balancer/`, 2023.

[36] Eric Van Hensbergen. Grave robbers from outer space: Using 9p2000 under Linux. In *USENIX 2005 Annual Technical Conference, FREENIX Track*, page 83–94, 2005.

[37] Patrick Hunt, Mahadev Konar, Flavio P. Junqueira, and Benjamin Reed. Zookeeper: Wait-free coordination for internet-scale systems. In *Proceedings of the*

2010 USENIX Conference on USENIX Annual Technical Conference, USENIXATC'10, page 11, USA, 2010. USENIX Association.

[38] Istio. Introducing ambient mesh. `https://istio.io/latest/blog/2022/introducing-ambient-mesh/`.

[39] Abhinav Jangda, Bobby Powers, Emery D. Berger, and Arjun Guha. Not so fast: Analyzing the performance of WebAssembly vs. native code. In *2019 USENIX Annual Technical Conference (USENIX ATC 19)*, pages 107–120, Renton, WA, July 2019. USENIX Association. ISBN 978-1-939133-03-8. URL `https://www.usenix.org/conference/atc19/presentation/jangda`.

[40] Zhipeng Jia and Emmett Witchel. Boki: Stateful serverless computing with shared logs. In *Proceedings of the ACM SIGOPS 28th Symposium on Operating Systems Principles*, SOSP '21, page 691–707, New York, NY, USA, 2021. Association for Computing Machinery. ISBN 9781450387095. doi: 10.1145/3477132.3483541. URL `https://doi.org/10.1145/3477132.3483541`.

[41] Zhipeng Jia and Emmett Witchel. Nightcore: Efficient and scalable serverless computing for latency-sensitive, interactive microservices. In *Proceedings of the 26th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS 2021, page 152–166, New York, NY, USA, 2021. Association for Computing Machinery. ISBN 9781450383172. doi: 10.1145/3445814.3446701. URL `https://doi.org/10.1145/3445814.3446701`.

[42] Eric Jonas, Qifan Pu, Shivaram Venkataraman, Ion Stoica, and Benjamin Recht. Occupy the cloud: Distributed computing for the 99 In *Proceedings of the 2017 Symposium on Cloud Computing*, SoCC '17, page 445–451, New York, NY, USA, 2017. Association for Computing Machinery. ISBN 9781450350280. doi: 10.1145/3127479.3128601. URL `https://doi.org/10.1145/3127479.3128601`.

[43] Konstantinos Kallas, Haoran Zhang, Rajeev Alur, Sebastian Angel, and Vincent Liu. Executing microservice applications on serverless, correctly. In *ACM SIGPLAN Symposium on Principles of Programming Languages (POPL)*, January 2023.

[44] Konstantinos Karanasos, Sriram Rao, Carlo Curino, Chris Douglas, Kishore Chaliparambil, Giovanni Matteo Fumarola, Solom Heddaya, Raghu Ramakrishnan, and Sarvesh Sakalanaga. Mercury: Hybrid centralized and distributed scheduling in large shared clusters. In *Proceedings of the 2015 USENIX Conference on Usenix Annual Technical Conference*, USENIX ATC '15, page 485–497, USA, 2015. USENIX Association. ISBN 9781931971225.

[45] Ana Klimovic, Yawen Wang, Patrick Stuedi, Animesh Trivedi, Jonas Pfefferle, and Christos Kozyrakis. Pocket: Elastic ephemeral storage for serverless analytics. In *13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 18)*, pages 427–444, Carlsbad, CA, October 2018. USENIX Association. ISBN 978-1-939133-08-3. URL `https://www.usenix.org/conference/osdi18/presentation/klimovic`.

[46] Peter Kraft, Qian Li, Kostis Kaffes, Athinagoras Skiadopoulos, Deeptaanshu Kumar, Danny Cho, Jason Li, Robert Redmond, Nathan Weckwerth, Brian Xia, Peter Bailis, Michael Cafarella, Goetz Graefe, Jeremy Kepner, Christos Kozyrakis, Michael Stonebraker, Lalith Suresh, Xiangyao Yu, and Matei Zaharia. Apiary: A dbms-integrated transactional function-as-a-service framework, 2023.

[47] Linux. Linux syscall table. `https://github.com/torvalds/linux/blob/master/arch/x86/entry/syscalls/syscall_64.tbl`, 2024.

[48] Frank Sifei Luan, Stephanie Wang, Samyukta Yagati, Sean Kim, Kenneth Lien, Isaac Ong, Tony Hong, Sangbin Cho, Eric Liang, and Ion Stoica. Exoshuffle: An extensible shuffle architecture. In *Proceedings of the ACM SIGCOMM 2023 Conference*, ACM SIGCOMM '23, page 564–577, New York, NY, USA, 2023. Association for Computing Machinery. ISBN 9798400702365. doi: 10.1145/3603269.3604848. URL `https://doi.org/10.1145/3603269.3604848`.

[49] Filipe Manco, Costin Lupu, Florian Schmidt, Jose Mendes, Simon Kuenzer, Sumit Sati, Kenichi Yasukata, Costin Raiciu, and Felipe Huici. My vm is lighter (and safer) than your container. In *Proceedings of the 26th Symposium on Operating Systems Principles*, SOSP '17, page 218–233, New York, NY, USA, 2017. Association for Computing Machinery. ISBN 9781450350853. doi: 10.1145/3132747.3132763. URL `https://doi.org/10.1145/3132747.3132763`.

[50] Microsoft. Extract, transform, and load (etl). `https://learn.microsoft.com/en-us/azure/architecture/data-guide/relational-data/etl`, .

[51] Microsoft. Extract, transform, and load (etl). `https://developers.cloudflare.com/reference-architecture/diagrams/serverless/serverless-etl/`, .

[52] Microsoft. Azure functions. `https://azure.microsoft.com/en-us/blog/introducing-azure-functions/`, .

[53] Philipp Moritz, Robert Nishihara, Stephanie Wang, Alexey Tumanov, Richard Liaw, Eric Liang, Melih Elibol, Zongheng Yang, William Paul, Michael I. Jordan, and Ion Stoica. Ray: A distributed framework for emerging AI applications. In *13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 18)*, pages 561–577, Carlsbad, CA, October 2018. USENIX Association. ISBN 978-1-939133-08-3. URL `https://www.usenix.org/conference/osdi18/presentation/moritz`.

[54] Roger Michael Needham and Andrew J. Herbert. *The Cambridge Distributed Computing System*. Addison Wesley, 1983.

[55] Edward Oakes, Leon Yang, Dennis Zhou, Kevin Houck, Tyler Harter, Andrea Arpaci-Dusseau, and Remzi Arpaci-Dusseau. SOCK: Rapid task provisioning with serverless-optimized containers. In *2018 USENIX Annual Technical Conference (USENIX ATC 18)*, pages 57–70, Boston, MA, July 2018. USENIX Association. ISBN 978-1-931971-44-7. URL `https://www.usenix.org/conference/atc18/presentation/oakes`.

[56] Diego Ongaro and John Ousterhout. In search of an understandable consensus algorithm. In *Proceedings of the 2014 USENIX Annual Technical Conference*, pages 305–319, Philadelphia, PA, June 2014.

[57] John K. Ousterhout, Andrew R. Cherenson, Frederick Douglis, Michael N. Nelson, and Brent B. Welch. The sprite network operating system. *Computer*, 21(2): 23–36, February 1988. ISSN 0018-9162. doi: 10.1109/2.16. URL `https://doi.org/10.1109/2.16`.

[58] Kay Ousterhout, Patrick Wendell, Matei Zaharia, and Ion Stoica. Sparrow: Distributed, low latency scheduling. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*, SOSP '13, page 69–84, New York, NY, USA, 2013. Association for Computing Machinery. ISBN 9781450323888. doi: 10.1145/2517349.2522716. URL `https://doi.org/10.1145/2517349.2522716`.

[59] Matthew Perron, Raul Castro Fernandez, David DeWitt, and Samuel Madden. Starling: A scalable query engine on cloud functions. In *Proceedings of the 2020 ACM SIGMOD International Conference on Management of Data*, SIGMOD '20, page 131–141, New York, NY, USA, 2020. Association for Computing Machinery. ISBN 9781450367356. doi: 10.1145/3318464.3380609. URL `https://doi.org/10.1145/3318464.3380609`.

[60] Rob Pike, Dave Presotto, Sean Dorward, Bob Flandrena, Ken Thompson, Howard Trickey, and Phil Winterbottom. Plan 9 from Bell Labs. *Computing Systems*, 8(3): 221–254, Summer 1995.

[61] Qifan Pu, Shivaram Venkataraman, and Ion Stoica. Shuffling, fast and slow: Scalable analytics on serverless infrastructure. In *16th USENIX Symposium on Networked Systems Design and Implementation (NSDI 19)*, pages 193–206, Boston, MA, February 2019. USENIX Association. ISBN 978-1-931971-49-2. URL `https://www.usenix.org/conference/nsdi19/presentation/pu`.

[62] Sheng Qi, Xuanzhe Liu, and Xin Jin. Halfmoon: Log-optimal fault-tolerant stateful serverless computing. In *Proceedings of the 29th Symposium on Operating Systems Principles*, SOSP '23, page 314–330, New York, NY, USA, 2023. Association for Computing Machinery. ISBN 9798400702297. doi: 10.1145/3600006.3613154. URL `https://doi.org/10.1145/3600006.3613154`.

[63] Gyorgy Rethy. Process-as-a-service computing on modern serverless platforms. Master thesis, ETH Zurich, Zurich, 2022-11-23.

[64] Malte Schwarzkopf, Andy Konwinski, Michael Abd-El-Malek, and John Wilkes. Omega: flexible, scalable schedulers for large compute clusters. In *SIGOPS European Conference on Computer Systems (EuroSys)*, pages 351–364, Prague, Czech Republic, 2013. URL `http://eurosys2013.tudos.org/wp-content/uploads/2013/paper/Schwarzkopf.pdf`.

[65] Simon Shillaker and Peter Pietzuch. Faasm: Lightweight isolation for efficient stateful serverless computing. In *2020 USENIX Annual Technical Conference (USENIX ATC 20)*, pages 419–433. USENIX Association, July 2020. ISBN 978-1-939133-14-4. URL `https://www.usenix.org/conference/atc20/presentation/shillaker`.

[66] Athinagoras Skiadopoulos, Qian Li, Peter Kraft, Kostis Kaffes, Daniel Hong, Shana Mathew, David Bestor, Michael Cafarella, Vijay Gadepally, Goetz Graefe, Jeremy Kepner, Christos Kozyrakis, Tim Kraska, Michael Stonebraker, Lalith Suresh, and Matei Zaharia. DBOS: A dbms-oriented operating system. *Proc. VLDB Endow.*, 15(1):21–30, jan 2022. ISSN 2150-8097. doi: 10.14778/3485450.3485454. URL `https://doi.org/10.14778/3485450.3485454`.

[67] Vikram Sreekanti, Chenggang Wu, Xiayue Charles Lin, Johann Schleier-Smith, Joseph E. Gonzalez, Joseph M. Hellerstein, and Alexey Tumanov. Cloudburst: Stateful functions-as-a-service. *Proc. VLDB Endow.*, 13 (12):2438–2452, jul 2020. ISSN 2150-8097. doi: 10.14778/3407790.3407836. URL `https://doi.org/10.14778/3407790.3407836`.

[68] Andrew S. Tanenbaum, Robbert van Renesse, Hans van Staveren, Gregory J. Sharp, and Sape J. Mullender. Experiences with the Amoeba distributed operating system. *Commun. ACM*, 33(12):46–63, December 1990. ISSN 0001-0782. doi: 10.1145/96267.96281. URL `https://doi.org/10.1145/96267.96281`.

[69] Shelby Thomas, Lixiang Ao, Geoffrey M. Voelker, and George Porter. Particle: ephemeral endpoints for serverless networking. In *Proceedings of the 11th ACM Symposium on Cloud Computing*, SoCC '20, page 16–29, New York, NY, USA, 2020. Association for Computing Machinery. ISBN 9781450381376. doi: 10.1145/3419111.3421275. URL `https://doi.org/10.1145/3419111.3421275`.

[70] Dmitrii Ustiugov, Plamen Petrov, Marios Kogias, Edouard Bugnion, and Boris Grot. Benchmarking, analysis, and optimization of serverless function snapshots. In *Proceedings of the 26th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '21, page 559–572, New York, NY, USA, 2021. Association for Computing Machinery. ISBN 9781450383172. doi: 10.1145/3445814.3446714. URL `https://doi.org/10.1145/3445814.3446714`.

[71] Abhishek Verma, Luis Pedrosa, Madhukar Korupolu, David Oppenheimer, Eric Tune, and John Wilkes. Large-scale cluster management at Google with Borg. In *Proceedings of the 10th ACM EuroSys Conference*, pages 18:1–18:17, Bordeaux, France, April 2015.

[72] Ao Wang, Shuai Chang, Huangshi Tian, Hongqi Wang, Haoran Yang, Huiba Li, Rui Du, and Yue Cheng. FaaSNet: Scalable and fast provisioning of custom serverless container runtimes at alibaba cloud function compute. In *2021 USENIX Annual Technical Conference (USENIX ATC 21)*, pages 443–457. USENIX Association, July 2021. ISBN 978-1-939133-23-6. URL `https://www.usenix.org/conference/atc21/presentation/wang-ao`.

[73] Liang Wang, Mengyuan Li, Yinqian Zhang, Thomas Ristenpart, and Michael Swift. Peeking behind the curtains of serverless platforms. In *2018 USENIX Annual Technical Conference (USENIX ATC 18)*, pages 133–146, Boston, MA, July 2018. USENIX Association. ISBN ISBN 978-1-939133-01-4. URL `https://www.usenix.org/conference/atc18/presentation/wang-liang`.

[74] Stephanie Wang, Eric Liang, Edward Oakes, Ben Hindman, Frank Sifei Luan, Audrey Cheng, and Ion Stoica. Ownership: A distributed futures system for Fine-Grained tasks. In *18th USENIX Symposium on Networked Systems Design and Implementation (NSDI 21)*, pages 671–686. USENIX Association, April 2021. ISBN 978-1-939133-21-2. URL `https://www.usenix.org/conference/nsdi21/presentation/cheng`.

[75] Xingda Wei, Fangming Lu, Tianxia Wang, Jinyu Gu, Yuhan Yang, Rong Chen, and Haibo Chen. No provisioned concurrency: Fast RDMA-codesigned remote fork for serverless computing. In *17th USENIX Symposium on Operating Systems Design and Implementation (OSDI 23)*, pages 497–517, Boston, MA, July 2023. USENIX Association. ISBN 978-1-939133-34-2. URL `https://www.usenix.org/conference/osdi23/presentation/wei-rdma`.

[76] Ethan G. Young, Pengfei Zhu, Tyler Caraza-Harter, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau. The true cost of containing: A gVisor case study. In *11th USENIX Workshop on Hot Topics in Cloud Computing (HotCloud 19)*, Renton, WA, July 2019. USENIX Association. URL `https://www.usenix.org/conference/hotcloud19/presentation/young`.

[77] Haoran Zhang, Adney Cardoza, Peter Baile Chen, Sebastian Angel, and Vincent Liu. Fault-tolerant and transactional stateful serverless workflows. In *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*, pages 1187–1204. USENIX Association, November 2020. ISBN 978-1-939133-19-9. URL `https://www.usenix.org/conference/osdi20/presentation/zhang-haoran`.

[78] Wen Zhang, Vivian Fang, Aurojit Panda, and Scott Shenker. Kappa: A programming framework for serverless computing. In *Proceedings of the 11th ACM Symposium on Cloud Computing*, SoCC '20, page 328–343, New York, NY, USA, 2020. Association for Computing Machinery. ISBN 9781450381376. doi: 10.1145/3419111.3421277. URL `https://doi.org/10.1145/3419111.3421277`.