σOS: Elastic Realms for Multi-Tenant Cloud Computing

by

Ariel Szekely

B.S., University of Texas at Austin (2020)

Submitted to the Department of Electrical Engineering and Computer Science
in partial fulfillment of the requirements for the degree of

Master of Science

at the

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

September 2022

Author .....
Department of Electrical Engineering and Computer Science
August 25, 2022

Certified by .....
M. Frans Kaashoek
Charles Piper Professor of Electrical Engineering and Computer Science
Thesis Supervisor

Accepted by .....
Leslie A. Kolodziejski
Professor of Electrical Engineering and Computer Science
Chair, Department Committee on Graduate Students

σOS: Elastic Realms for Multi-Tenant Cloud Computing

by

Ariel Szekely

Submitted to the Department of Electrical Engineering and Computer Science
on August 25, 2022, in partial fulfillment of the
requirements for the degree of
Master of Science

Abstract

Despite the enormous success of cloud computing, programming and deploying cloud applications remains challenging. Application developers are forced to either explicitly provision resources or limit the types of applications they write to fit a serverless framework such as AWS Lambda.

σOS is a new multi-tenant cloud operating system that allows providers to manage resources for tenants while simplifying application development. A key contribution of σOS is its novel abstraction: *realms*. Realms present tenants with the illusion of a single-system image and abstract boundaries between physical machines. Developers structure their applications as processes, called `proc`s in σOS. Much like a time-sharing OS multiplexes users' processes across a machine's cores, σOS multiplexes tenants' `proc`s across the cloud provider's physical machines. Since each tenant tends to plan for peak load, realms can improve data center utilization by enabling providers to transparently reallocate partial machines to another tenant's realm when load dips.

An evaluation of σOS demonstrates that a σOS-based MapReduce (σOS-MR) implementation grows quickly from 1 core to 32 and scales near-perfectly achieving 15.26× speedup over the same implementation running on 2 cores. Similarly, an elastic Key-Value service built on σOS (σOS-KV) cooperates with σOS to scale the number of `kvd` servers and balance shards across them, according to client load. σOS also achieves high resource utilization when multiple tenants' realms compete for a shared group of machines. For example, when σOS multiplexes a long-running σOS-MR job in one realm and a σOS-KV service with varying numbers of clients in another realm, σOS keeps utilization above 90% and transparently moves partial machines between the realms as the σOS-KV client load changes.

Thesis Supervisor: M. Frans Kaashoek
Title: Charles Piper Professor of Electrical Engineering and Computer Science

Acknowledgments

I would like to sincerely thank everyone who has supported me on this first step into my graduate school journey. This work would not have been possible without Frans' dedication, patience, and mentorship. Robert's and Adam's comments and ideas have also been invaluable in the development of this work.

I am also tremendously thankful for my friends, both inside of PDOS and out, who have helped make made the difficult moments bearable, and who have always been willing to lend a sympathetic ear or another set of eyes throughout all of the bugs squashed in the making of this thesis.

Of course, I wouldn't even be here if it weren't for the love and encouragement of my family and my parents, Sally and Francisco Szekely. Thank you for always being there for me, for inspiring me with your passion for learning, and for always believing in me even when I didn't believe in myself. I am more grateful than you can possibly imagine.

Chapter 1

Introduction

σOS is a new operating system for cloud computing. Unlike existing cloud computing platforms, σOS allows cloud providers to manage resources such as machines, CPUs, and memory for tenants, while tenants develop cloud applications using convenient abstractions such as processes, pipes, and so on, without having to worry about provisioning machines or keeping track where processes run. σOS reallocates resources from one tenant's application to another tenant's application to meet each tenant's load, analogues to how a time-sharing operating system transparently reallocates CPUs and memory from one application to another application.

σOS's approach is based on the assumption that a provider has many tenants and that the provider multiplexes the tenants on the provider's infrastructure. Tenant loads vary and are hard to predict, so to achieve high utilization, the provider must be able to reassign the resources devoted to each tenant as that tenant's load, and the loads of competing tenants, change. The goal of σOS is to make it easy for tenants to write elastic applications and to offload resource provisioning to the provider.

To allow σOS to move resources between tenants transparently, σOS introduces the notion of a *realm*: a per-tenant, elastic cluster of machines with a "single-system image" shared across the cluster. Developers structure their applications as `proc`s (which are inspired by Unix processes [42]) and σOS runs those `proc`s on the realm's machines. The `proc`s can be indifferent to the machine they run on, because a realm provides a single name space with a root name server. Using pathnames, a `proc` can name and access realm resources such as files, pipes, and other `proc`, which may be on a different machine. This single-system image allows σOS, for example, to add a machine to a realm and then schedule a `proc` on this new machine, and allows the new `proc` to interact with other `proc`s in the realm transparently. To provide a single system-image, the σOS interface resembles the core of the Unix API, but doesn't provide a complete POSIX API.

To allow σOS to shrink and grow realms effectively, σOS encourages developers to use many `proc`s for both batch-style (e.g., MapReduce jobs [14]) and long-running services (e.g., Web sites). For example, σOS's MapReduce library organizes a MapReduce job as a coordinator `proc` which spawns a `proc` for each mapper and reducer task, and waits for their completion, perhaps spawning a new `proc` if a tasks fails. σOS's Web server spawns a new `proc` for each HTTP request. Spawning queues these

procs to σOS for execution, and σOS can decide when and where to run them proc.

To decide when to shrink and grow a realm σOS peeks inside the realm. For example, if σOS observes that a realm has a backlog of spawned procs, it can allocate a partial machine (with some CPU and RAM) to that realm and move procs from the spawn queue to that machine. Alternatively, if σOS observes that a realm is underutilizing its resources it can temporarily steal resources for other realms to use, or ask the realm to scale down its applications. In response, the realm can evict some of its application's procs or rely on natural death of short-lived procs; the partial machine freed-up by those exited procs can then be re-allocated to another realm.

To asses the feasibility of σOS's ideas we implemented a prototype of σOS. A provider runs σOS on its infrastructure and then σOS schedules realms' procs and reallocates partial machines between realms. The tenant's experience is much like writing application on a time-sharing operating system: tenants spawn proc and under the hood σOS allocates partial machines, boots σOS on them, assigns the machines to a realm, and schedules the realm's procs on those machines.

σOS provides a single protocol, σP, to build applications and with which σOS itself is built too. σP is based on 9P [25, 40] and is a simple protocol: 10-20 remote procedure calls. It supports a single name space, access to storage, direct communication between procs, watches for coordination, and fences for fault tolerance. In other words, σP doubles as a resource-discovery protocol, a single-system-image naming protocol, a storage protocol, an alternative to RESTfull APIs [19, 47], and a ZooKeeper-like coordination service [26].

To demonstrate it is convenient to build elastic applications using σOS's procs, we implemented a MapReduce library, a fault-tolerant, sharded in-memory key/value service, and a dynamic Web server. Preliminary experiments with a small-scale deployment on Cloud Lab and AWS EC2 demonstrate that σOS can dynamically and quickly allocate resources across realms in response to utilization changes in a realm.

The contributions of this thesis are:

- A new cloud operating system, σOS, that supports realms, which allows cloud providers to provision and manage physical resources and enables tenants to develop applications without having to worry about provisioning machines;
- A new protocol, σP, that bundles service discovery, storage access, single-system naming, and Zookeeper-like coordination;
- A two-level distributed scheduler that schedules procs within a realm and reallocates resources between realms;
- A demonstration that σOS can achieve high utilization and divide resources up well between competing applications.

The σOS source code is fully open-source at https://github.com/mit-pdos/sigmaos.

Chapter 2

Motivation and goals

High utilization is critical for cloud providers: they want to keep their expensive hardware busy running useful work for tenants. Electricity alone is a significant cost: data centers in the US consumed approximately 3% of the US's electricity in 2017, and are expected to consume 8-10% by the end of the decade [12, 49, 51]. Achieving high utilization, however, is challenging, since tenant loads vary and are hard to predict. In practice, cloud providers report low utilization [7, 31, 53]. Increasing utilization would allow providers to do more with each running machine at no extra costs.

This thesis aims to achieve high utilization through elasticity: the provider varies the resources devoted to each tenant as that tenant's load, and the loads of competing tenants, vary. Elasticity requires cooperation between provider and tenant: the provider needs to know when a tenant could profitably use more resources or could get by with fewer, and the tenant must be able to quickly exploit added resources and give up revoked resources without disruption.

As an example of why elasticity is hard, consider AWS (with similar parallels for the other major Cloud providers). AWS provides two main compute platforms: Amazon's EC2 and AWS lambdas. Tenants can create and destroy EC2 instances on demand, but the relationship between load and the number of instances is entirely up to the tenant. Because this is hard, and AWS doesn't help, many tenants allocate a fixed number of EC2 instances determined by likely peak load, so that many are idle in non-peak periods; the result is low utilization.

AWS's lambda service, in contrast, knows how much work a tenant would like to perform (if any), and thus can devote resources to a tenant only when it has work to do. However, lambda is suitable only for queuable batch jobs of finite duration since AWS uses queue lengths to drive elasticity, and uses completion of jobs to free up resources. To enforce completion AWS will unconditionally terminate a lambda that runs too long (15min at the time of writing). Thus, neither EC2 nor lambda has much to offer to tenants who need to build long-running stateful elastic services of their own (Web sites, storage services, load balancers, and so on), which is perhaps why AWS itself provides elastic versions of those services to tenants.

Chapter 3

σOS design

σOS's abstractions and interfaces are designed so that tenant and provider cooperate to obtain elasticity and high utilization. For both batch-style and long-running services, developers organize applications into processes, called `proc`s, and queue these `proc`s to σOS for execution when and where it is convenient for σOS. σOS thus knows when a tenant has a backlog of `proc`s, and can allocate machines and move processes from the queue to those new machines. Similarly, because most processes run for short quanta, σOS can observe when a tenant's load decreases by observing that the queue of spawned processes is shrinking, and can deallocate machines as work processes complete. The result is that σOS tenant software is naturally elastic, reducing tenant resource consumption and decreasing over-provisioning and consequent low utilization.

This section presents the main σOS abstractions, and how they fit together, to achieve elasticity.

3.1 System overview

Figure 3-1 gives an overview of σOS's design. Each tenant has a *realm*, an elastic of cluster of machines with a single system image across the cluster. Developers structure applications in terms of `proc`s (the P_is in Figure 3-1), which are the unit of isolation and work in σOS.

A tenant that creates a realm doesn't provision any resources such as CPUs and memory, unlike in traditional cloud platforms like AWS. If a tenant wants to run an application, the tenant doesn't need to create a Virtual Private Cloud, configure it with a several instances, decide what to run on those instances, what to run on spot instances, what to run as lambda functions, and so on. Instead, with σOS the tenant creates an account, creates a realm, and spawns `proc`s within that realm.

The σMGR allocates resources (CPU and RAM) to realms. The resources may constitute a partial machine or a full machine. Each machine runs the σOS kernel, which provides, for example, primitives to create and destroy `proc`s. In the prototype, the σOS kernel runs atop a locked-down Linux kernel and σOS runs `proc`s as Linux processes, hardened using seccomp [17].

There are two types of `proc`s: kernel `proc`s, which can use Linux kernel system calls, and user `proc`s, which don't have limited access to system calls and interact only through the σOS API. Kernel `proc`s include `procd`, which uses the kernel primitives

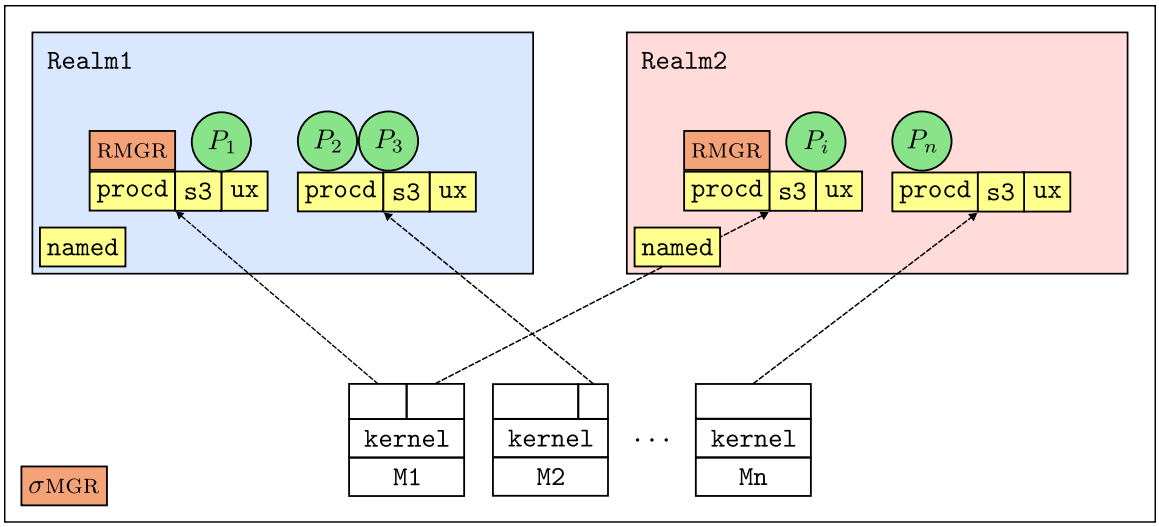

Figure 3-1: Overview of σOS design. User-space `procs` are green, kernel `procs` are yellow, realm management `procs` are orange, and machines (`Mi`) are white. The example σOS system has two realms, each with a few (partial) machines that are allocated to the realm by the σOS manager. Each machine runs the σOS kernel and kernel `procs` such `procd`, `ux`, and `s3`. The `procd`s of a realm collaboratively schedule the user `procs` (P_is) of a realm. RMGR monitors load in its realm and asks or returns machines to the σMGR. Each realm has a `named` that is the root of the realm's namespace.

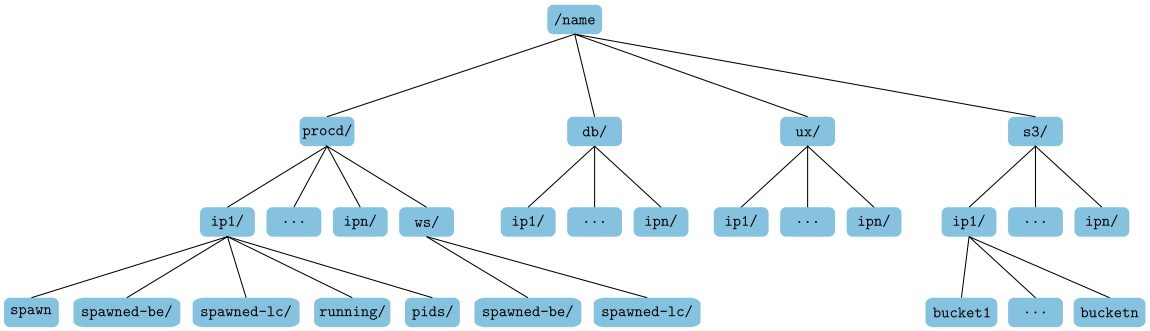

Figure 3-2: A realm's namespace.

for processes to run `procs`. The `procd`s of a realm collaboratively schedule the realm's user `procs`. Another kernel proc, RMGR, monitors a realm and collaborates with σMGR to request and return resources to the σMGR.

3.2 Single-system image realms

In a realm σOS `procs` often need to communicate and interact. To support this, σOS provides a single-system image through a distributed file-system-like naming and storage system that `procs` can use to cooperate with each other. Figure 3-2 shows parts of a realm's name space for a proc in the prototype.

Each realm has a `named` proc that implements the root of the realm's name space

name/. All procs talk to named to resolve pathnames. A given pathname has the same meaning to all of a realm's procs, so procs can meaningfully exchange pathnames.

The named service provides a file-system-like hierarchy of directories and files. procs can store modest amounts of data in these files. procs can coordinate with each other with these files, using atomic operations such as exclusive create. This aspect of the design is inspired by ZooKeeper [26].

A service proc makes itself available to client procs by mounting themselves in the named namespace. If a client looks up a pathname that refers to one of these mount points, the client will start speaking the file-system protocol directly to the service mounted there. This aspect of the design is inspired by Plan 9 [40] and SFS [32].

For example, in Figure 3-2, the procd directory lists the procd services in realm; there is one procd service for each partial machine allocated to a realm and each procd service advertises itself using its IP address as name. The procd services collaboratively offer services for procs to create and manage other procs. Each procd exports its internal state through the file system protocol; for example, each procd has a directory pids, which lists all the procs that this procd manages, much like /proc on Linux. It also exports its queue of running procs (running) and spawned procs (spawned-lc and spawned-be), which are waiting to be run.

named and procd—and any other service in a realm—speak the same file system protocol. For illustration, here are some ways that σOS procs use named and the σOS file system protocol (with details in subsequent sections):

- /proc-like status
- work stealing
- watches for events
- proxies that make other services available in a realm's namespace
- fault-tolerant state such as views for fault-tolerant services
- leader election, and leader failure detection

The named service is not suitable for storing large amounts of data, and does not support sophisticated queries, so it is expected that procs implementing more specialized storage and database systems will announce themselves via named to sOS client procs. For example, σOS has s3 proxies that expose S3 buckets to a realm's namespace, ux proxies that expose each machine's local storage to the realm, and a db proxy to export an SQL database.

3.3 σOS procs

σOS developers arrange their applications in terms of σOS procs, and σOS schedules these procs. Since procs in a realm share a single name space that contains σOS objects (e.g., files, other procs, etc.), σOS can schedule procs on any machine available to a realm. If σMGR gives a realm a new machine, the procs scheduled on this new machine can interact with the other procs in the realm transparently.

Method	Description
`Spawn(pid, descriptor)`	Create `proc` with `pid` as name
`WaitStart(pid)`	Wait until `pid` has started
`WaitExit(pid)`	Wait until `pid` has exited
`WaitEvict(pid)`	Wait until `pid` has been evicted
`Started(pid)`	`pid` marks itself as started
`Exited(pid, status)`	`pid` marks itself as exited
`Evict(pid)`	Ask `pid` to evict itself
`SpawnBurst(descriptors)`	Spray a list of `procs` across `procds`

Figure 3-3: Summary of the σOS `proc` API.

Figure 3-3 lists the `proc` API. A developer creates a new `proc` using `Spawn`, which takes as arguments the process identifier (PID) to assign to the `proc` and a descriptor, which holds a pathname for the binary, a list of arguments to be passed to the `proc`, environment variables, and so on. The developer specifies whether the `proc` performs latency-critical (LC) work or best-effort (BE) work. For LC `procs`, the developer also specifies how many CPUs the LC `proc` needs to achieve low latency at peak load. For especiallly memory-intensive `procs`, the developer can also specify the `proc`'s memory requirement. This BE/LC information effects how a `proc` is scheduled (§3.4).

The caller of `Spawn` can wait until its child starts running by calling `WaitStart` or wait until its child exits by calling `WaitExit`. `WaitExit` returns an exit status to the caller, which allows the child to pass a result to the parent. A `proc` signals to a parent that is running or has exited using `Started` and `Exited`, respectively.

The `Evict` call allows the RMGR to alert a `proc` that will be evicted soon to clear the machine it is running on, similar to how Borg evicts tasks [54]. On an eviction signal, a `proc` can checkpoint itself, save its state, or just exit and rely on another mechanism to redo its computation. If the `proc` doesn't exit within a given amount of time, the RMGR will instruct the local `procd` to terminate the `proc`. Once the `procs` on a machine have exited, RMGR returns the machine to the σMGR.

By default, `Spawn` writes the supplied descriptor to the local `procd`'s control file `spawn`. `SpawnBurst` sprays the descriptors for new `procs` across the `procds` of a realm. For each `proc` written to a `procd`, the `procd` exports state about that `proc` through a directory `pid`, where `pid` is the `proc`'s pid. This directory stores files and directories to implement a `proc`: semaphore files for `WaitStart`, `WaitExit`, and `Evict`, one directory for each child of the `proc` (if the `proc` spawned child `procs`), a symlink to the parent `pid` directory (which maybe stored at a `procd` on a different machine), the `proc`'s exit status file, and so on.

3.4 Scheduling `procs` in a realm

When `Spawn` or `SpawnBurst` writes a descriptor to a `procd`'s control file `spawn`, the `procd` also adds the pid to either the directory `spawned-be` or `spawned-lc`, depending whether the proc is BE or LC. The `spawned-be` and `spawned-lc` directories serve as

```
// Try to get a proc for this procd to run.
func (pd *Procd) getProc() (*Proc, bool) {
  localQs := "name/procd/" + pd.MyAddr() + "/"
  globalQs := "name/procd/ws/"
  // Claim order:
  // 1. local LC queue
  // 2. remote LC queue
  // 3. local BE queue
  // 4. remote BE queue
  queues := []string{
    localQs + SPAWNED_LC,
    globalQs + SPAWNED_LC,
    localQs + SPAWNED_BE,
    globalQs + SPAWNED_BE,
  }
  for _, q := range queues {
    procs := pd.GetQueuedProcs(q)
    for _, p := range procs {
      if pd.hasEnoughMem(p) && pd.hasEnoughCores(p) {
        // If this procd has sufficient resources to
        // run this proc, try to claim it.
        if ok := pd.tryClaimProc(p); ok {
          return p, true
        }
      }
    }
  return nil, false
}
```

Figure 3-4: `procd` selects a `proc` to run by scanning its realm's queues. First, the `procd` searches for any LC `proc` it can run, and if it can't find any, it searches for a BE `proc` to run.

queues, and the `procd`s of a realm monitor these queues and collectively schedule the `proc`s on their machines.

Figure 3-4 describes how `procd`s select `proc`s to run, and Figure 3-5 shows the criteria `procd`s use to decide whether or not they can run a `proc`. A `procd` claims an LC `proc` by first scanning its local queue. If there are any LC `proc`s at its local `spawned-lc` queue, and `procd` has the capacity to run one, it tries to claim the `proc`. A `procd` has capacity for an LC `proc` if it has sufficient cores and memory that it hasn't assigned to any other LC `proc`s.

If `procd` doesn't find any LC `proc`s it can run, it looks in the global work-stealing queue directory `/name/procd/ws`, to see if another `procd` advertised an LC `proc` that the other `procd` could not run, and which this `procd` may be able to.

If a `procd` has no capacity for LC `proc`s or there are no LC `proc`s to be run, but the utilization of one of its cores is below 90%, it looks for a BE `proc`. It tries to claim

```
func (pd *Procd) hasEnoughCores(p *Proc) bool {
  if p.IsLC() {
    // If this is a Latency Critical proc, and
    // this procd does not have enough cores for
    // it to run at peak load, don't run it.
    if pd.freeCores >= p.NumCores() {
      return false
    }
  } else {
    // If this is a Best Effort proc and the
    // CPU utilization is above the system's
    // utilization target, don't run it.
    if pd.cpuUtil() > CPU_UTIL_TARGET {
      return false
    }
    // If this procd has quickly claimed many
    // Best Effort procs, back-off for a while
    // to allow CPU utilization statistics to
    // stabilize.
    if pd.numBEClaimedRecently() > CLAIM_RATE_LIMIT {
      return false
    }
  }
  // All checks pass. This proc can be run.
  return true
}
```

Figure 3-5: `procd` determines whether it can run a `proc` by checking if it has enough unallocated cores, if the `proc` is LC, or if the `procd`'s CPU utilization is low, if the `proc` is BE.

a `proc` first from the local `spawned-be` queue. If the local queue has no BE `procs`, it looks in the gloabl work-stealing directory `/name/procd/ws` to see if another `procd` advertised a BE `proc`, and tries to claim it.

Figure 3-6 shows the process by which `procds` decide to advertise stealable `procs` to other `procds`. If a `procd` has insufficient capacity to run a `proc` for a sufficiently long period of time, it advertises that `proc` in `/name/procd/ws`, a directory which serves as the global work stealing queue. Other `procds` with unallocated resoruces scan this queue periodically, and if another `procds` can run the `proc`, the `procd` will steal it.

If a `procd` successfully claims a `proc`, it creates a Linux process for it. It assigns a high scheduling priority to LC `procs` so that if a LC `proc` is busy it gets most of its CPUs, which may otherwise be shared with BE `procs`.

Once the `proc` is running, the `proc` mounts its `pid` directory, which maybe remote, as `/procdir`, so that the `proc` can updates its state (e.g., up the semaphore for marking that it has started, which may unblock the parent, if it called `WaitStart`).

```
func (pd *Procd) offerStealableProcs() {
  localQs := "name/procd/" + pd.MyAddr() + "/"
  globalQs := "name/procd/ws/"
  queues := []string{
    SPAWNED_LC,
    SPAWNED_BE,
  }
  for !pd.Done() {
    time.Sleep(PROC_STEALABLE_TIMEOUT)
    for _, q := range queues {
      // Get procs from the local spawn queue.
      procs := pd.GetQueuedProcs(localQs + q)
      for _, p := range procs {
        if p.SpawnTime > PROC_STEALABLE_TIMEOUT {
          // If this proc has not been spawned for
          // a long time, offer it to other procds
          // by creating a symlink to its file in
          // the local spawn queue.
          pd.Symlink(globalQs + q, localQs + q)
        }
      }
    }
  }
}
```

Figure 3-6: `procd`s mark `proc`s as stealable and offer them to other `procd`s by adding them to the realm's global work-stealing queue, `/name/procd/ws`. `procd`s offer a proc as stealable after they have remained un-claimed in the `procd`'s local queue for a sufficiently long amount of time.

Similarly, it mounts its parent `proc` directory as `/parent` so that the child can interact with the parent. Note that a new `proc` mounts its directory and its parent directory with local pathnames (instead of global pathnames) so that a parent can run a child process without mounting `name`, thereby isolating the child.

This overall design for implementing `proc`s allows for good scalability: `proc`s can create children in parallel on different machines. To support work stealing σOS uses a hybrid push/pull model to avoid network communication: `procd`s push `proc`s to `/name/procd/ws`, from where other `procd`s pull `proc`s. An alternative design would be a purely pull-based one (i.e., `procd`s scan other `procd`s' run queues when they don't have spawned `proc`s locally) but that results in more communication. The hybrid design allows a `procd` to wait on a change in `/name/procd/ws` instead of continually scanning other `procd`s' directories. Finally, the design guarantees that LC `proc`s have dedicated CPUs, but achieves high utilization by using under-utilized CPUs to run BE `proc`s.

```
// Monitor the realm and request resources
// from smgr if needed.
func (rmgr *RealmMgr) monitor() {
  for !rmgr.Done() {
    rmgr.Sleep(REALM_RESIZE_FREQUENCY)
    // Get the aggregate length of all the
    // spawned queues in the realm.
    qlen := rmgr.realm.getQueueLen()
    avgCPUUtil := rmgr.realm.getAvgCPUUtil()
    // If the queue is long enough to fill up
    // another node, or the queue length
    // is non-zero and the average CPU
    // utilization in the realm is above the
    // utilization target.
    if qlen >= NODE_SZ ||
      (qlen > 0 && avgCPUUtil > REALM_AVG_CPU_UTIL_TARGET) {
      // Ask the smgr to grow the realm.
      rmgr.requestGrowRealm()
    }
  }
}
```

Figure 3-7: An RMGR monitors its realm's resource utilization and global `spawned` queue length, in order to decide whether or not the realm needs to grow to meet the tenant's applications' demands.

3.5 Growing and shrinking realms

A long queue of spawned `procs` in a realm is a signal to RMGR to ask the σMGR for more machines. The prototype σMGR assumes that the tenant specifies a minimum dollar amount run to run LC `procs` and a maximum rate the tenant is willing to pay to handle a burst of BE `procs`. σMGR will not scale the tenant's realm beyond the maximum rate. σMGR's goal is then to move unused resources to realms that can use more resources to run `procs` (e.g., BE `procs`), but rapidly re-allocate those resources if another realm suddenly develops a queue of spawned LC `procs`.

There are several major challenges in shrinking and growing realms: 1) how to measure load and decide if the realm requires more or fewer machines; and 2) the granularity at which physical resources are handed out; 3) how to move state when shrinking/expanding physical resources; and 4) how to handle rapid changes in load. In general, σMGR uses similar ideas to existing cluster managers such as Borg, Omega, and Kubernetes [4]. However, since applications in σOS are organized as collections of `procs`, the RMGR and can σMGR inspect the realm's state. This allows them to learn when more (or fewer) resources are needed, and enables σOS to achieve higher utilization.

Challenge 1. Figure 3-7 describes how RMGR monitors its realm's load, and decides to grow the realm. The RMGR monitors the utilization of all the realm's `procd`s (which `procd`s export as a file), and the realm's global work-stealing queue, which contain the `proc`s that exceeded `procd`s' capacity. If the work-stealing queue length crosses a threshold and the average CPU utilization in the realm is high, the RMGR asks σMGR for more machines.

Challenge 2. Figure 3-8 shows how σMGR assigns additional machines to realms which need them. σMGR hands out `node`s, a partial machine, consisting of some CPUs and a proportional amount of RAM. If σMGR grants a `node` to a realm with a `procd` that already manages another part of that machine, then that `procd` combines the two parts together. This allows a realm to grow vertically. If the `node` corresponds to a physical machine for which the realm has no `procd`, the `node` starts a new `procd` that manages that `node`. This allows a realm to scale horizontally.

Partial machines also help σMGR shrink realms gracefully. If σMGR asks an RMGR to free up resources and the RMGR's realm contains one or more merged `node`s, the RMGR can split one of the merged `node`s and return a partial machine to σMGR instead of a full machine. Any of the realm's `proc`s that were running on that physical machine are then constrained to a smaller set of resources (e.g., fewer CPUs), but are allowed to continue running. This allows RMGR to avoid evicting `proc`s unless the physical machine cannot be split into smaller `node` chunks. If the realm contains no merged `node`s RMGR can evict that `node` and all of its `proc`s as a last resort. σMGR avoids fully-evicting `node`s which run LC procs to avoid degrading interactive applications' performance. Instead, it relies on LC `proc`s to be elastic themselves (§5.2).

Even if a realm has only LC `proc`s, σOS can still achieve good utilization. If a realm requests more resources than its LC `proc`s are using, RMGR can steal underutilized resources and offer them to σMGR in the form of `node` partial machines. σMGR can temporarily allocate these `node`s to other realms to run BE `proc`s, and then quickly return them when the original realm's LC `proc` load increases. In this way σOS is able to avoid fully evicting LC `proc`s and still ensure high resource utilization even when some realms run no LC `proc`s.

Challenge 3. Figure 3-9 shows how σMGR decides whether to evict `node`s and trigger state migration. σMGR asks the RMGR of the lowest utilized realm with spare resources to free up a `node`. If RMGR doesn't free up a `node` within that time, then σMGR can forcefully evict some node in the realm and take it back. σOS uses the following strategies for the RMGR to free up a `node`:

- σOS encourages developers to structure their applications using many short-running `proc`s. For example, σOS's web server forks a `proc` to serve a connection; that `proc` may interact with `db`, retrieve a static file, or run some computation. With this design, the RMGR can ask the least loaded `procd` to stop running `proc`s and return that `procd`'s node to σMGR when the `proc`s on that node have exited.

- Some applications have idempotent `proc`s that can safely be re-executed. For such applications RMGR can just evict any `proc` without saving its state. For

```
// Grow a realm's resource allocation if possible.
func (smgr *SigmaMgr) growRealm(realm *Realm) {
  // If there are free nodes in the system,
  // allocate one to the realm.
  if smgr.hasFreeNodes() {
    smgr.allocNode(realm)
    return
  }
  // Try to find an overprovisioned node in
  // another realm.
  for _, realm2 := range smgr.realms {
    if realm == realm2 {
      continue
    }
    // Get nodes in sorted order by CPU
    // Utilization.
    nodes := realm2.nodes.sortedByCPUUtil()
    for _, node := range nodes {
      // If realm2 has an overprovisioned node,
      // free it and allocate the node to the
      // growing realm.
      if nodeIsOverprovisioned(node) {
        smgr.freeNode(realm2, node)
        smgr.allocNode(realm)
        return
      }
    }
  }
}
```

Figure 3-8: When growing a realm, σMGR first checks if there are unallocated resources available. If not, σMGR tries to find an overprovisioned realm and reclaim one of its `node`s. After reclaiming the `node`, it assigns it to the growing realm.

example, the coordinator in the σOS MapReduce library will create new `proc`s for failed mappers and reducers and another `procd` can run those new `proc`s, freeing up the node whose `proc`s were evicted.

- Some applications are written with elasticity in mind. For example, the σOS key/value store can move shards on demand. So, if the key/value store is lightly loaded, the RMGR can ask the key/value store to shrink the number of servers, which will cause shards to be moved to other key/value servers, and free up a node. If the load increases, the σOS key/value server will spawn new `proc`s to take over serving shards from other `proc`s. A queue of new `proc`s will cause RMGR to ask for `node`s to run these `proc`s.

- Some applications can migrate themselves: they can checkpoint their state on

```
// Returns true if node is overprovisioned.
func nodeIsOverprovisioned(node *Node) bool {
  stats := node.GetStats()
  // If removing some cores would cause there to be too
  // few cores to support the node's current LC proc CPU
  // utilization, the node is not overprovisioned.
  if node.NumCores - NODE_MIN_CORES < stats.CPU.Util.LC {
    return false
  }
  // If this node cannot be split further, shrinking it
  // will trigger a node eviction. We handle this
  // case specially.
  if node.NumCores == NODE_MIN_CORES {
    // If the total CPU utilization of this node is above
    // a minimum threshold, don't evict the node.
    if stats.CPU.Util.Total >= NODE_MIN_CPU_UTIL {
      return false
    }
    procd := node.GetProcd()
    // If there are LC procs queued at this node's procd,
    // evicting the node would force parent procs to
    // respawn them, which would harm latency. In this
    // case, we avoid evicting the node.
    if len(procd.GetSpawnedLCQueue()) > 0 {
      return false
    }
    // If there are LC procs running on this node, don't
    // evict the node.
    if len(procd.GetRunningLCProcs()) > 0 {
      return false
    }
  }
  // If all checks pass, the node is overprovisioned.
  return true
}
```

Figure 3-9: When shrinking a realm, σMGR needs to select the right `node` to reclaim from the victim realm. It makes this determination based on a variety of factors including the `node`'s CPU utilization, whether or not it is running any LC `procs`, and whether the `procd`'s `spawned` queue is long. If reclaiming cores from this `node` would cause it to be evicted, σMGR first checks that there are no LC `procs` running on the `node`.

eviction (e.g., in an S3 bucket exported through σOS) and spawn a new `proc` that starts from the saved checkpoint.

We considered migrating running `procs` to avoid having to evict them. Since

Method	Description
`AdvanceEpoch(pathname)`	Advance epoch number
`FenceAtEpoch(epoch, dir)`	Fence calls to objects in dir

Figure 3-10: The epoch and fence API.

transparent migration from one machine to another is challenging to implement correctly, however, we haven't explored this option. We expect that the above strategies cause unrecoverable evictions to be rare, which would be in line with reported experience with Borg [54] and Harvest VMs [2, 57].

Challenge 4. Since all tenants run `node`s with the σOS image, σOS can quickly allocate pre-initialized nodes from a hot-standby pool to a realm that experiences a rapid increase in load. A σOS pre-initialized node elides some of the initialization and environment setup costs that some cluster managers must endure to create a node with the right set of OS packages, because each node may use a different OS and different packages. The hot-standby pool gives σOS some head room to use one of the strategies above to free up nodes.

3.6 Coping with failures

A goal of σOS is to support stateful applications that can handle crashes (e.g., one of the machines in the realm loses power and takes out its `procd` and all the `proc`s it is running). As a first step to be able to recover from failures, developers can specify in the descriptor passed to `Spawn` the failure domain that they want the `proc` to run in.

When a `proc` creates a file it can mark the file as *ephemeral*, which is particularly useful for leader election. `proc`s that want to become a leader of a replicated service create an ephemeral symlink for the service that points to themselves. The first `proc` that succeeds in creating the symlink (i.e., the symlink doesn't exist) becomes the leader. The other candidate leaders will wait in the `Create` call until the first leader removes the symlink voluntarily or until the leader's session with the server storing the symlink is terminated. Sessions are terminated unilaterally when a `proc`'s connection to a server breaks and fails to be re-established for a sufficiently long period of time. When a session terminates, all ephemeral files associated with that session are removed.

To avoid a split-brain scenario in which both a partitioned old leader and a new leader are running at the same time, σOS supports epochs and fences (see Figure 3-10). Applications that want to avoid a split-brain create an epoch file, which contains an epoch number, on the same server where a leader creates the symlink (e.g., in `named`'s file system). The epoch file is an ordinary file.

The new leader uses `AdvanceEpoch` to atomically read the epoch file and increment the epoch number in the file; this call is implemented using σOS's core API (§4.1), which supports reads and writes conditional on a file's version number. The old leader can no longer interact with the server storing the epoch file because its session was

terminated.

To stop the old leader from writing to other servers after it has been partitioned, leaders can ask σOS to fence σOS calls for objects in a specified directory with an epoch number. Every σOS server will reject requests that carry an epoch number that is lower than what it already has seen. So, after the new leader sends a request to a server, the server will reject the old leader's requests because they carry an older epoch number. Note that servers don't have to be aware that they are part of fault-tolerant service: a fault-tolerant application that uses some server can count on that server's rejecting fenced requests, because the σOS protocol supports them.

Clients of a replicated service can use `FenceAtEpoch` similarly to ensure that their requests go to the right server for a given epoch by fencing their operations using the current epoch. If a client receives a stale epoch error in response to a request, the client knows that a configuration changes has happened and will lookup the new configuration and its epoch number.

Chapter 4

σOS implementation

This section overviews how σOS is implemented by describing: (1) the σOS API; (2) the σP protocol; (3) automounting of services; (4) the lines of code for the implementation; and (5) deploying σOS.

4.1 The σOS API

The σOS API is inspired by the core Unix API: it small—see Figure 4-1 for the main calls—but expressive: many of σOS abstractions are implemented using this API, including `procs` (§3.4). In principle, if a tenant likes to change `procd`'s policy, the tenant's developers can write their own using σOS API and run it instead.

The σOS API allows `procs` to name all resources in a realm using pathnames and access them through a file API using `Create`, `Open`, `Read`, `Write`, and `Close`. Procs can create file-like objects (e.g., files, directories, symlink, pipes etc.) and virtual file objects (e.g., `/proc`, semaphores, etc.). These objects are then accessible through pathnames to other `procs` in the realm, perhaps with restricted access due to limited permissions.

In σOS, some pathnames name files on a disk or S3 buckets, others name keys of an in-memory key-value stores, yet others name `procs` themselves, and so on. For example, to list all the running `procs` in a realm, the programmer lists the directory of running `procs`, analogues to `/proc` but for all machines in a realm.

Some `procs` offer services that don't directly fit in σOS's core API. Those `procs` typically export a virtual file to which client `procs` write commands, similar as in Plan9 [40]. For example, σOS has a `proc`, `db`, which exports an SQL database. Client write queries to `db`'s command file and read the query results from a response file.

Procs uses watches [26] for coordination: for example, `OpenWatch` will block until the specified pathname exists and then invoke `func`. σOS provides libraries that encapsulate watches in more convenient coordination primitives; for example, σOS implements `WaitExit` using semaphores, which in turn are implemented using watches.

4.2 σOS protocol

The σOS protocol, σP, has multiple functions: it is naming protocol that provides a single-system image through pathnames, it is a protocol to discover servers, it is a storage protocol for applications to read/write data; it is a communication protocol

Abstraction	Methods	Description
Files	`Create(path, perm, mode)`	Create (ephemeral) file/dir/link/pipe
	`Open(path, mode)`	Open file/dir/link
	`Close(fd)`	Close file descriptor returned by `Create`/`Open`
	`Remove(path)`	Remove object named by path
	`Rename(old, new)`	Rename old to new (within a file system)
	`Stat(path)`	Returns info about object named by path
	`Read(fd)`	Returns data from file descriptor
	`Write(fd, data)`	Write data to file descriptor
	`Lseek()`	Changes offset for file descriptor
	`Put(path, perm, mode, data)`	Create (ephemeral) file/dir/link with data
	`Get(path)`	Returns data
	`Set(path, data)`	Set file to data
	`Mount(path1,path2)`	Mount `path1` at `path2`
Watches	`OpenWatch(path, func)`	Open file or wait file is created
	`SetDirWatch(path, func)`	Call `func` when directory changes
	`SetRemoveWatch(path, func)`	Call `func` when path is removed

Figure 4-1: The core σOS API. Other APIs, including the `proc` API are implemented using the core API. `procs` can also create special files, such as named pipes and control files, and operate on them using the core API.

that allows applications to directly interact without dropping down to TCP; it is a RESTfull API alternative that defines a uniform interface for many servers, and it is a coordination protocol for `procs`.

Figure 4-2 shows the request messages in σP. The σOS protocol is derived from 9P [25, 40], which provides a small set of carefully thought-out remote procedure calls. σP extends 9P with support for self-certifying pathnames, sessions (to support transparent fail-over for clients and a reply table to filter duplicate requests), ephemeral files, watches, reads/writes at a file version, fences, and put/get/set of small files (which combine walking a pathname, opening a file, reading/writing the file, and closing it in single RPC). σOS supports these extension with a protocol that is only slightly more complicated than 9P: for example, supporting fences and reads/writes at a version require small changes to 9P, but many others extensions can be implemented in the client and server libraries that use σP.

Any `proc` (e.g., `procd`) that exports a file system must implement the σOS protocol; to simplify implementing servers, σOS provides a generic protocol library,

Operation	Description
VERSION	Return protocol version number
AUTH	Authenticate server
ATTACH	Attach server to client's name space
WALK	Walk a pathname, returns file identifier
OPEN	Open a pathname, returns file file identifier
CREATE	Create file/dir/etc
WRITE	Write data
READ	Read data
CLUNK	Release file identifier
REMOVE	Release file identifier
RENAMEAT	Rename a file at a server atomically
STAT	Return file info
WATCH	Watch a pathname
PUTFILE	Create pathname and set data
GETFILE	Get data for pathname
SETFILE	Set data for pathname

Figure 4-2: Request messages of the σOS protocol, which is derived from 9P [25, 40]. Each message has a corresponding response message (not shown). Each request may carry an epoch number to fence the request, and read/write can indicate that the request is conditional on the version number at which the file client opened the file.

`protsrv`, that takes care of all the generic protocol operation so that a `proc` just has to implement the file and directory objects. A `proc` can also import one of the existing file system implementations to export S3 buckets, Unix files, or an in-memory file system (which are all implemented using `protsrv`). An advantage of this design is that for servers that fit the file system API developers don't have to define RPCs or RESTfull APIs (which in practice involves quite bit of boilerplate). Furthermore, these servers can use features like pathname lookup, permissions, watches, and so on, out of the box.

Supporting fences requires little additional mechanism: each σOS request carries an epoch number, `protsrv` checks the epoch number, and the σOS library supports a call to specify which directories should be fenced at a particular epoch number.

4.3 Automounting

σOS allows a `proc` to export a namespace that clients will transparently mount when accessing a file in that name space. For example, a `proc` that implements a key-value service can announce its services by creating a symbolic link in the `name` name space with a special format: the symlink contains the network name (DNS name or IP address, and a port number) and the `proc`'s public key, inspired by SFS's self-certifying pathnames [32]. For example, a key/value service can create a symbolic link `name/kv`, which contains the DNS name of the key-value service and its public key.

	Component	LOC
Core	σOS protocol	4,825
	σOS API	5,734
	`realm`	1,722
	`procd`	1,780
	`named`	82
	`s3`	921
	`ux`	767
	`dbd`	210
	`proxy`	465
Libraries	`protsrv`	2,387
	`semclnt`	163
	`electclnt`	192
	`epochclnt`	219
	`groupmgr`	201
Applications	`mr`	1,678
	`kv`	1,579
	`wwwd`	382
Total		23,307

Figure 4-3: Lines of code for subsystems of σOS (excluding `etcd`'s Raft [18], which σOS uses to replicate `named` and `kv` shards)

When σOS resolves a pathname, say `name/kv/key-10`, it will automount the destination file system (i.e., `name/kv`), which involves starting an authenticated session with the `proc`, and continue resolving the remainder of the path (i.e., `key-10`) at the server. These symlinks allow `proc`s to hook their name space into the shared `name` name space, and allow other `proc`s to look up the `proc`'s file objects.

4.4 Lines of code

The prototype of σOS is implemented in the Go programming language [11] and runs on top of Linux. Although σOS uses Linux as its kernel, only kernel `proc`s have access to Linux system calls; all user `proc`s use only the σOS API.

Figure 4-3 lists σOS's parts, including test code but excluding `etcd`'s Raft [18]. σOS protocol includes the Go packages for the protocol definition, RPC stubs, sessions, which multiplex and demultiplex σOS messages on a TCP connection and provide fail-over and duplicate detection. σOS API are the Go packages that implement σOS API and the `proc` API.

4.5 Deployments

We deploy σOS in two ways: on CloudLab on dedicated machines and on AWS as a VPC. In both deployments, a user `sshs` into a realm and launches σOS programs that create `procs` in the realm. For convenience, users can explore the state of a realm using Unix utility programs such as `ls`, `cat`, etc., to see, for example, all the running `procs`. This is accomplished by mounting σOS's `proxy` program, which converts 9P into σP, as a 9P file system under Linux.

Chapter 5

σOS applications

To explore how to build elastic applications using σOS, we built a a MapReduce library, and a sharded, fault-tolerant key/value service, and dynamic web server.

5.1 MapReduce

σOS's MapReduce library, `mr`, illustrates how a developer can use σOS for implementing a data processing framework. `mr` uses a coordinator `proc` to manage the mappers and reducers. `mr` replicates the coordinator with the `groupmgr` library, which starts three coordinators. They elect one leader using the `electclnt` library and the others are hot standbys. The leader coordinator stores its progress (e.g., which mappers have completed) in `name/mr`. If the leader coordinator crashes, one of the standbys becomes leader and pick up from where the crashed coordinator left off. The `groupmgr` will also start another standby.

The coordinator spawns a `proc` for each mapper and reducer. The mapper `proc` stores the intermediate output file for each reducer persistently on the local machine using the pathname `name/ux/~local` and record a symbolic link in the directory `name/mr/r`i`/` (one per reducer). Reducer i watches this directory and if a new link output appears, the reducer reads the link, automounting the `ux` server that exports the file in the `proc`. This implementation allows overlap between the map and reduce phases; reducers can run as soon as a mapper's output is available.

If a mapper or reducer crash, the coordinator will restart it. If a reducer cannot read an intermediate file and fails, the coordinator will also restart the appropriate mapper.

Note that in this design there are no worker machines; it is the job of σOS to provision machines to run the coordinator, mapper, and reducer `proc`s. All of the MapReduce library's `proc`s are BE.

5.2 key/value service

σOS's key/value service, `kv`, illustrates how a developer can use σOS to build fault-tolerant applications. One possible approach to supporting a key/value service in σOS is side-stepping most of σOS: take an existing high-performance key/value store and modify it to export a control file in the realm's name space to which clients write requests (much like `db`). Alternatively, one could export the key space through a proxy

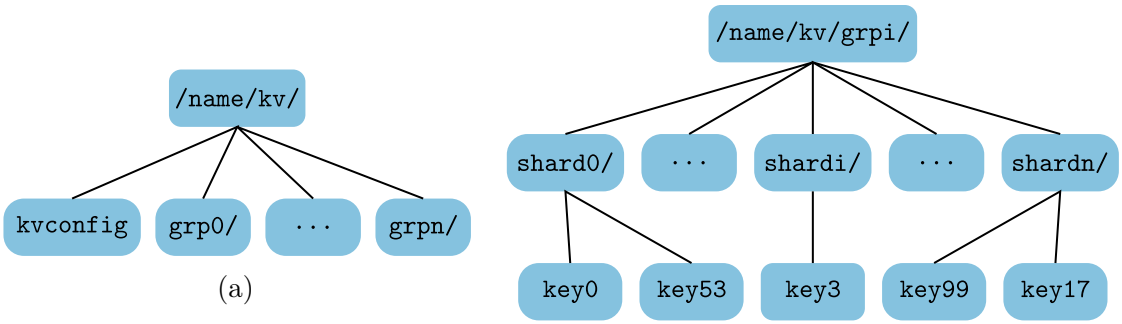

Figure 5-1: KV setup and namespace. Figure 5-1a shows the top-level directory structure of a `kv` service. The `kv` service exposes a config file which the clerks watch and the `balancer` modifies as it balances load across the `kvd` groups. Figure 5-1b shows a sample of the `kvd` directory strucutre. Shards are directories, and keys are files in the directories. the `balancer` moves shards across `kvd` groups by invalidating the `kvconfig` file, atomically moving the directories, and then publishing a new `kvconfig` file.

as `s3` does. This section doesn't take either approach but instead explores how to build a key/value service natively using σOS's API. Even though a realm provides a single system image that mostly hides machine boundaries, the key/value service must be linearizable in the presences of crashes and shard reassignment.

The kv store is organized as a group of balancer `procs` and one or more KV groups, which serve shards and which each consist of three `kv procs` for fault tolerance. One of the balancer `procs` elects itself as leader using the `electclnt` library, and it is responsible for adding KV groups, terminating KV groups, and balancing shards across KV groups in response to changes in load.

Figure 5-1 shows where state of the kv service is stored. The balancer stores all its state, which includes a table that maps shards to KV group i and an epoch number, in the file `kvconfig` in the `kv` directory at `named`. So, if a balancer crashes, a new elected balancer can find the last state in `name/kv/kvconfig`. Each KV group elects a leader using the `electclnt`, which uses `etcd`'s Raft library [18] to replicate `put` and `get` operations on each `kvd procd` of the group. Each `kvd` serves an in-memory file system with a directory for each shard that the group is responsible for, and with a file for each key in that shard (as shown at the right side of Figure 5-1). Each KV group posts a symbolic link `name/KV/`grp_i, to which each member of the group adds its DNS names (and public key).

Clients of the `kv` service, use the `kvclerk` library, which supports a `put` and `get` interface to interact with the service. The clerk reads the `name/KV/kvconfig` file, which it caches for subsequent access. The clerk computes the shard number for the key it wants to access, and looks up the group responsible in the shard table. Then, it performs an σOS `put` or `get` operation (Figure 4-1) with `name/KV/`$grp_i/shard/key$ as the pathname argument. When σOS resolves `name/KV/`grp_i, it will automount one of the `kvds` for `name/KV/`grp_i. The mounted `kvd` will receive a `Put` request (see

Figure 4-2) and insert it in `etcd`'s Raft log, so that each kvd will perform the `Put` on its in-memory file system. Note the latter `Put` operation is implemented by the `protsrv` library, which all σOS `proc`s use that serve the σOS protocol, whether they part of a replicated group or not.

If a `kv` server crashes, the `etcd` Raft library will remove it from the group and the `groupmgr` library will start a new one, which the `etcd` Raft library will add to the group. Any clerk that was using the failed `kvd` server will fail over to another one in the group transparently: σOS will fail to perform a `put` or `get` request because the connection to the `kvd` has been broken, which causes σOS to unmount the crashed `kvd`, automount one of the other `kvd`s in `name/KV/`grp_i, and send the request to that `kvd`.

The balancer serves request to grow and shrink the KV store. To grow, the balancer computes a new `kvconfig` file that minimizes shard movement, adds the list of shards to be moved to the file, increments the epoch number, and then posts the new file using `Rename` (which is atomic). Then, it creates a mover `proc` for each shard in the to-be-moved list. Each mover `proc` moves its shard by copying the shard directory and the key files from `name/KV/`$grp_i/shard/$ to `name/KV/`$grp_j/shard/$. It first copies them into a temporary directory `name/KV/`$grp_j/shard\#/$, and then atomically renames the directory to `name/KV/`$grp_j/shard/$. This extra step is necessary to handle the scenario in which the balancer starts a second mover `proc`, because it lost connection with the first one but, if there was only a network failure, the first one might still be running and also copying the shard. When the mover exits without a failure indicator, the balancer removes the shard from the to-be-moved list in `kvconfig`.

A mover `proc` uses `FenceAtEpoch` to fence its requests to move a shard with the epoch number that the balancer passed as an argument to the mover through `Spawn`. When a `kvd` server learns about the new epoch (because it is a source or destination of mover), it will reject requests from movers or clerks in older epochs. To ensure the latter, clerks also use `FenceAtEpoch` to tell σOS to fence their requests with the epoch number they read from `kvconfig`. If a clerk receives a stale epoch error, it rereads `kvconfig` to find out the new shard mapping and tries again. Fencing with epoch numbers ensuring linearizability of puts and gets.

If the leader balancer crashes, one of the standby leaders will elect itself as the new leader. The new leader reads `kvconfig` to to pick up where the old leader left off. The new leader increments the epoch number and spawns new movers, if to-be-moved list isn't empty.

A realm that use `kv` doesn't provision machines. If the balancer adds new KV groups to handle an increase in load and the realm has no capacity to run them, a queue of spawned `proc`s will develop, signaling the σMGR to give the realm more `node`s. If the kv realm's RMGR is asked by σMGR to return `node`s, it asks the `balancer` to shrink the number of KV groups so that it uses fewer resources.

The `kvd` and `clerk` `proc`s are LC. All other `kv` `proc`s are marked BE.

5.3 Web server

σOS has a simple Web server, which is structured in a similar way to OKWS [29]. The server consists of an `wwwd` `proc`, which accepts HTTP connections, and which

spawns `proc`s to serve content. The `wwwd` `proc` runs without `named` mounted so that if it is compromised, the attacker cannot explore the realm and mount other services. `wwwd` does mount the `pid` directory of each `proc` it spawns, so that it can wait for a child to start, to exit, etc.

For static content, `wwwd` spawns a `proc` that has access to the directory with static content. For dynamic pages, `wwwd` spawns a `proc` that can mount mount `db`. Before spawning a child, `wwwd` creates a pipe in its in-memory file system and whose names it passes along to the child so that it can return the response over the pipe, `wwwd` then returns to the client. The Web server runs as a LC `proc`, while the other `proc`s run as BE ones.

A realm that uses `wwwd` doesn't provision machines. As the load on `wwwd` increases, it will spawn more `proc`s. If the realm doesn't have enough resources to run those `proc`s, the realm will develop a long spawned queue and σMGR will allocate new resources to realm. If the load decreases, the realm's RMGR can return any `node` that doesn't run any `proc`s.

Chapter 6

Evaluation

In this section we seek to answer the following questions:

1. Can a single realm scale up quickly in response to increased load?
2. Is σOS able to effectively multiplex resources across multiple realms?
3. Do σOS applications perform well?
4. Are σOS abstractions efficient?

Experimental Setup. We run σOS's application-level benchmarks on an AWS Virtual Private Cloud (VPC) composed of 16 EC2 `t3.small` VM instances. Each instance has 2 vCPUs, 2GiB of memory, and a 200GiB EBS volume. For the comparison to Corral (§6.3), we run σOS on an AWS Virtual Private Cloud (VPC) composed of 8 EC2 `t3.medium` VM instances. Each instance has 4 vCPUs, 2GiB of memory, and a 20GiB EBS volume. Aditionally, each instance has up to 5Gbps network burst bandwidth, and 2,085Mbps EBS burst bandwidth. We run σOS's microbenchmarks on a cluster of 5 Cloudlab [16] r650 nodes. Each node has two 36-core Intel Xeon Platinum 8360Y CPUs at 2.4GHz, 256GB ECC DDR4-2666 Memory, a Dual-port Mellanox ConnectX-5 25Gb NIC, and a Dual-port Mellanox ConnectX-6 100 Gb NIC.

Datasets. We use a snapshot of all English html Wikipedia pages taken on May 4th, 2022 as the dataset for the MapReduce experiments. The full snapshot is 89GB (wiki-89G) of raw text which we trim down to 2GB (wiki-2G) for some experiments.

6.1 Elasticity within a single realm

σOS must quickly grow realms in response to spikes in application load. However, this poses a challenge for σOS. In order to achieve high resource utilization, the realm's `procd`s must balance `proc`s across across an elastic set of resources to ensure that few resources are idle over the duration of the job.

We evaluate σOS's ability to provide and make use of elasticity using two applications: MapReduce-grep (MR) and the σOS Key-Value Service (KV). MR is representative of Best Effort (BE), long-running batch Jobs. KV is representative of a Latency Critical (LC), stateful and interactive application, and thus introduces a different set of challenges in the face of dynamic scaling.

MapReduce-grep (MR). The MR workload relies purely on σMGR to manage elasticity. As the MR coordinator `proc` spawns mapper and reducer `proc`s, σMGR notices the realm's utilization and queue length increase. σMGR incrementally grows the realm, starting fresh nodes on free machines and gradually allocating more cores to existing heavily-utilized nodes. `procd`s started on fresh nodes begin to steal work from existing `procd`s, and mappers and reducers start to transparently access intermediate files on remote machines symlinked into the realm's namespace.

σMGR must respond to the realm's increase in load quickly to enable MR to derive full performance benefit from all of the provider's resources. Moreover, the realm's `procd`s should collaborate to balance `proc`s across the realm and keep node utilization high. Ideally, this would enable MR to speed up proportionally to the number of VMs that σMGR controls.

Figure 6-1 plots the MR job's aggregate throughput as it runs in a 16VM AWS VPC. Initially the tenant's realm has only half of a VM (1 core) assigned to it. σMGR detects the realm's growing queue size and the increase in the realm's CPU utilization. It gradually scales the realm's resource allocations up, eventually allocating all 16 VMs (32 cores) to the tenant's realm.

σOS is able to provide almost perfect linear speedup as the VPC it manages grows. Specifically, the MR job is 15.26× faster when running on 16 VMs than it is when running on 1 VM. The good speedup results from the `procd`s' collaboratively balancing load well across the realm's fluctuating set of resources. In the trace from Figure 6-1, all VMs are utilized until the benchmark is 79% complete, and 15 VMs are fully utilized until 85% of the job is complete. In short, σOS achieves both high performance for tenants as well as high utilization for cloud providers when dynamically scaling applications.

Note that roughly every 60-70 seconds, there is a sharp drop in the MapReduce job's aggregate throughput. We observed similar behavior with the same periodicity in all of the benchmarks. We believe this is due to variations in the VPC's network bandwidth. In order to confirm this, we ran `iperf` on two machines in the VPC, one acting as a client and the other as a server. We observed 20-40% drops in network throughput between the pair of machines roughly every 60-70 seconds as well.

Key-Value Service (KV) Unlike MR, the KV service must move state as it grows and shrink, and participates in scaling decisions. A balancer `proc` monitors the load of the `kvd` servers and then spawns additional `kvd` servers and redistributres shards in order to balance load evenly while minimizing data movement. As the balancer scales the application by spawning `proc`s, it indicates to σMGR that the realm needs to grow. This leads σMGR to assign more cores to the tenant's realm. Ideally, σMGR should quickly grow the KV service's realm in order to acommodate the balancer's scaling decisions while achieving high resource utilization.

Figure 6-2 shows the aggregate client-side throughput of 16 KV clerks as they execute operations against an elastic, dynamically-scaled KV service. The benchmark driver program begins by spawning the balancer, 16 KV clerks, and an initial `kvd` server populated with 1000 keys per clerk. For the first 45 seconds, the benchmark driver program waits for the `kvd`'s initial state to be set up and for σMGR to allocate

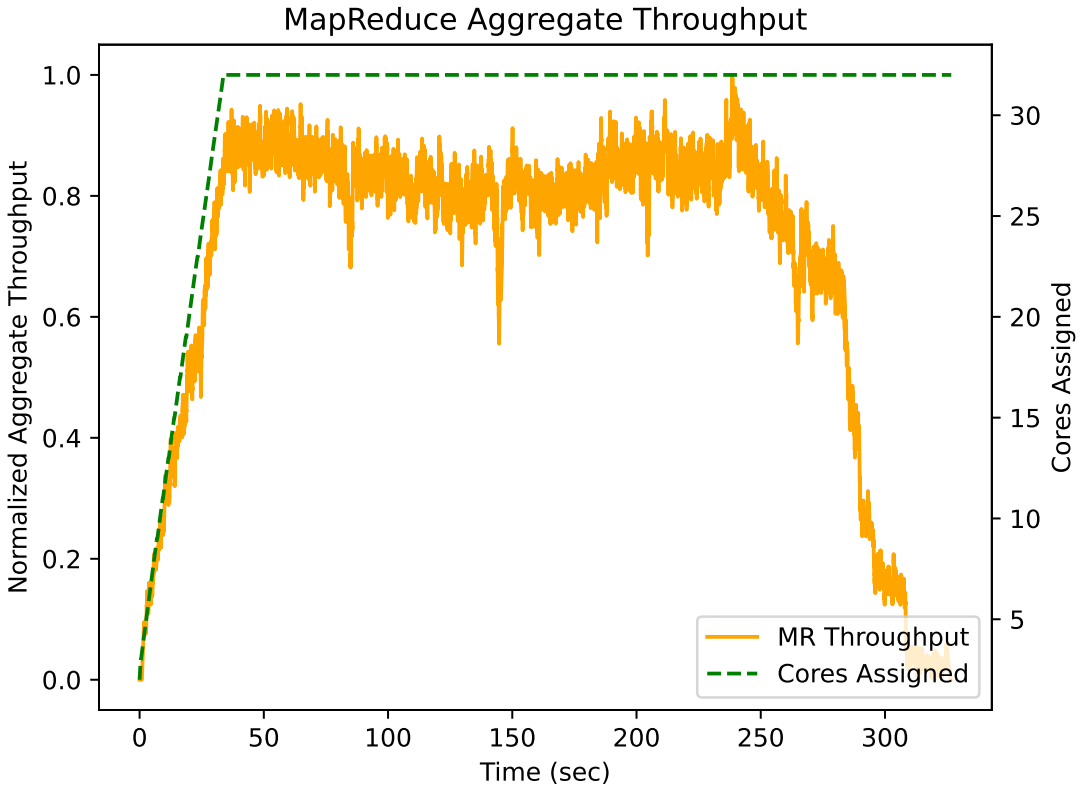

Figure 6-1: Aggregate throughput of MapReduce-grep running on the wiki-89G dataset in a 16 VM AWS VPC. Initially only half a VM (one core) is assigned to the realm. As the MR job's load increases, σMGR grows the realm to encompass all 32 cores in the VPC (the green line).

enough cores for all of the `procs` to start. Once the setup is complete the clerks simultaneously perform Set and Get operations against their set of keys in a tight loop.

At around 60 seconds the balancer detects high load at the `kvd` server and scales up the KV deployment to 2 `kvd` servers, each holding half of the shards. For a short period of time, the clerks' aggregate throughput drops to 0 as they wait for the shards to move and for the balancer to update the config file which indicates the new location of each shard. Then, the clerks continue executing Set and Get operations against the 2 `kvd` servers until the benchmark terminates after 135 seconds.

Figure 6-2 demonstrates that stateful, interactive applications can drive σOS's scaling and resource multiplexing decisions. That cooperation between a realm and σOS enables a stateful applcation to be elastic. Furthermore, σOS does not overprovision the KV server's realm, assigning only 20 cores to it: 1 for each of the 16 clerks, and 2 cores per `kvd` server. This ensures that CPU utilization is high for the duration of the benchmark.

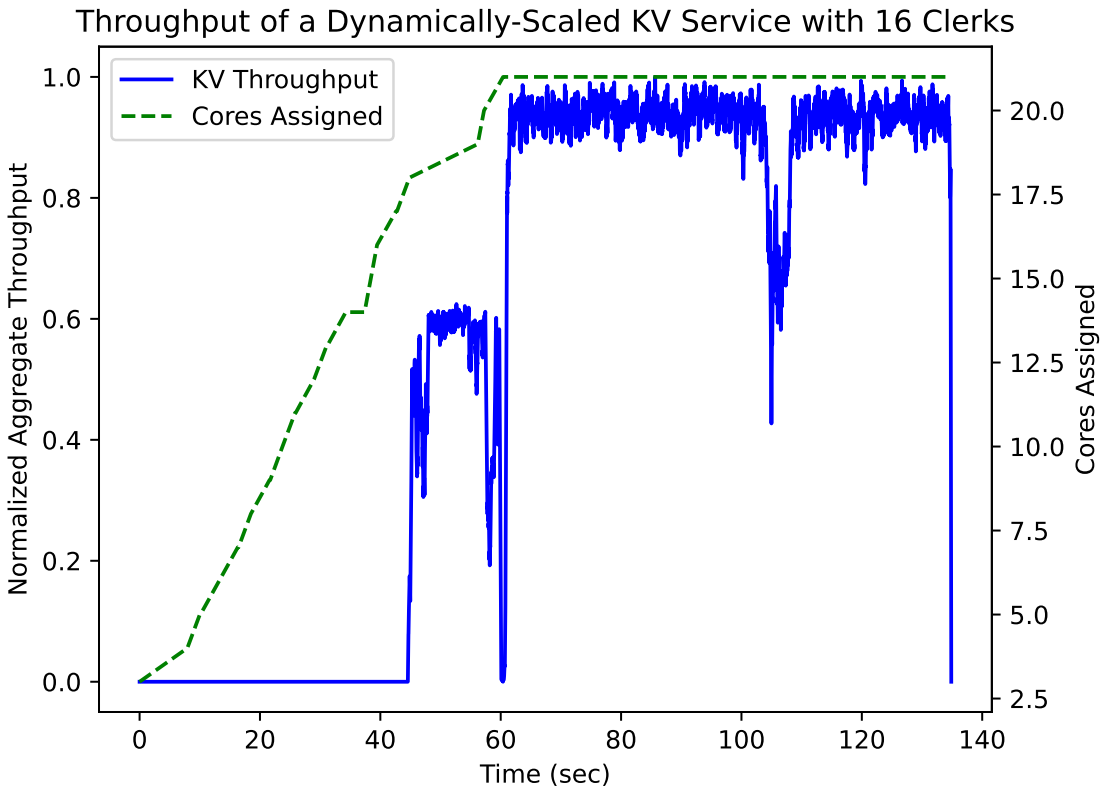

Figure 6-2: Client-side aggregate throughput of 16 clerks accessing a dynamically-scaled KV service. The first 45 seconds are spent initializing clerks' keys and setting up the KV service. At 45 seconds, 16 clerks start to perform Set and Get operations in a tight loop. At 60 seconds, the balancer detects high load at the `kvd` server and scales up to 2 `kvd` servers. The clerks' throughput briefly drops to 0 as they wait for the balancer to rebalance shards and publish a new config file.

6.2 Growing and shrinking multiple realms

A goal of σOS is to multiplex multiple realms across a fixed set of resources. In order to achieve high utilization, σOS steals resources from realms with Latency-Critical (LC) `procs` when the realm is idle. However, in order to meet LC `procs`' performance goals, σOS must quickly return stolen resources when they are needed.

We evaluate whether σOS is able to achieve high utilization and high application performance across multiple realms by setting up two competing realms on a 16 VM AWS VPC. Then, we start an LC application, KV with 16 clerks, in one realm and a BE application, MR-grep running on wiki-89G, in the other. We measure the performance degradation of each application relative to the unconctended setting, where each realm runs on dedicated hardware. Ideally, σOS should prioritize LC `procs` across both realms, and allow BE `procs` to make progress while the LC `procs` are idle.

As shown in Figure 6-3, σMGR quickly reallocates cores from the BE realm to the

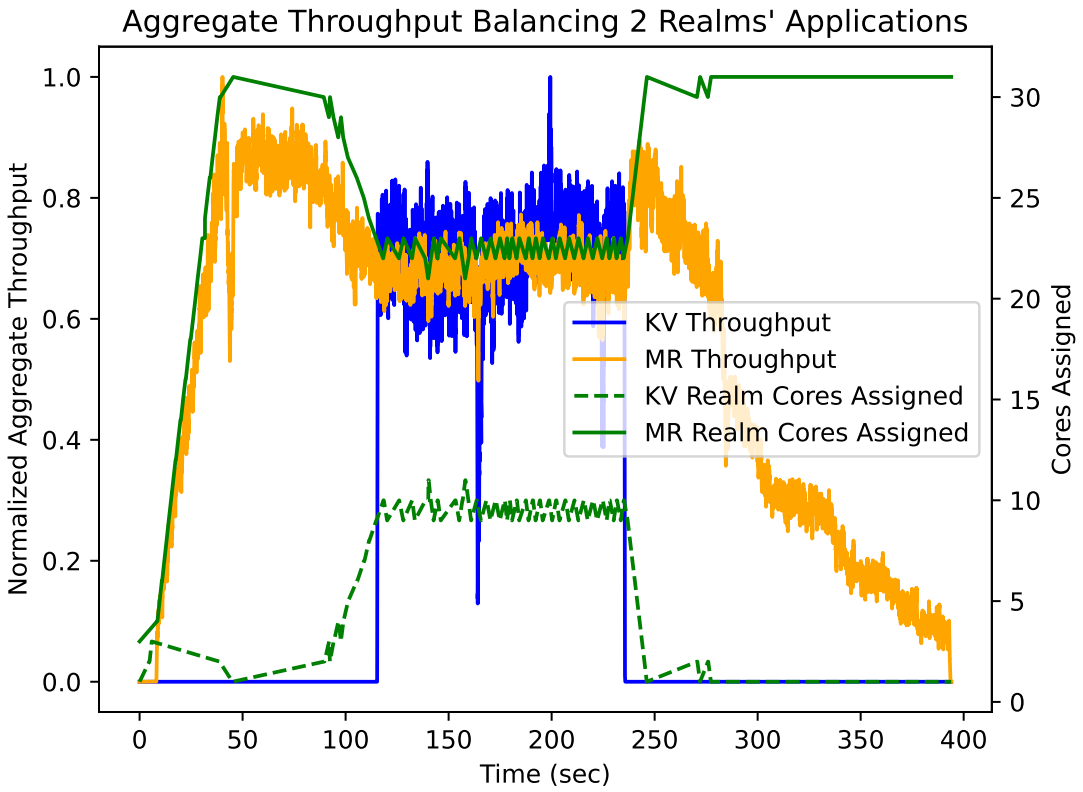

Figure 6-3: Aggregate throughput of two applications in different realms. The MR realm starts to execute a BE MR-grep job on the wiki-89G dataset. While the KV realm runs an LC KV service which is initially idle. 110 seconds later, 8 clerks begin performing Get and Set operations on the KV service, leading to an increase in load. This causes σMGR to reassign cores from the MR realm to the KV realm (as shown by the green lines), degrading MR's throughput but allowing KV to achieve 44% of the throughput it achieves running on dedicated hardware. At 240 seconds, the KV clerks stop issuing requests, and σMGR reassigns the KV realm's idle cores to the MR realm, allowing MR's throughput to increase again.

LC realm once the LC realm's load increases. The BE realm's application continues to make progress, albeit with lower throughput, while the LC realm is busy. When the clerks in the LC realm terminate and σMGR detects decreased resource utilization in the LC realm, σMGR reassigns its cores back to the BE realm. In response to the added resources, the BE realm's throughput improves until the BE job enters its final phase.

σOS manages to achieve high utilization, keeping all machines in both realms busy for the duration of the experiment. Both applications see some performance degradation from this resource sharing. The BE job (MapReduce-grep), which lost around 30% of its cores for roughly 30% of its execution time, took 23% longer to complete compared to the same MR job running on 16 dedicated VMs. The throughput

of the LC job (one `kvd` server with 8 clerks) degraded by 45% compared to the same `kvd` job running on 16 dedicated VMs.

The LC application's performance suffers for two reasons. First, the current σMGR implementation is conservative about evicting nodes from a realm. If the KV realm starts LC `proc`s which reserve all the cores on a machine and then go idle for some time, some of the cores may be stolen by a BE realm. When the LC `proc`s' load increases, the LC realm may try to recover all its stolen cores from the BE realm. However, since recovering all the stolen cores requires that σMGR evict the BE realm's node and some of its procs, σMGR's conservative eviction policy stops the LC realm from recovering all of its cores.

Second, the current implementation provides performance isolation for user-level `proc`s, but not kernel-level `proc`s. This means that the BE realm's kernel-`proc`s, such as its `s3` proxy and local storage server, may share cores with the LC realm's `proc`s, slowing them down.

In conclusion, σOS is able to mantain high resource utilization by multiplexing realms with LC and BE tasks across a the same set of hardware. σOS does not provide perfect performance isolation between `proc`s in competing realms, end-to-end application performance is within 23% of ideal for BE jobs, and within 55% of ideal for LC jobs.

6.3 Performance of σOS applications

σOS Applications must use the σOS API, which provides transparency across a realm, but incurs cost for implementing that transparency. Furthermore, σOS applications communicate using the API. To measure the benefit and cost of σOS, we compare the σOS-MapReduce (σOS-MR) implementation to Corral [10], a MapReduce framework which runs on AWS Lambda, and the Key-Value service (σOS-KV) implementation to Redis, a popular in-memory Key-Value service written in C.

σOS-MR vs. Corral. Much like σOS, serverless computing platforms such as AWS Lambda promise low-latency burst parallelism and automatic application scaling. However, Lambdas cannot reliably hold state across invocations, and must checkpoint intermediate state to durable storage services like Amazon S3. Moreover, Lambdas cannot reliably wait for other lambdas, as they have a strict timeout after which they are forcefully terminated. Through the realm abstraction, σOS allows `proc`s to access local storage, and the σP protocol allows a realm's long-lived `proc`s to communicate and wait for each other.

In order to make the comparison fair, we provision Corral's lambdas with 1760MB of memory, which gives them the equivalent of 1 vCPU. We use the wiki-2G dataset, as larger datasets have very large intermediate files which cause both σOS-MR and Corral's reducers to run out of memory.

Table 6.1 details σOS-MR and Corral's end-to-end runtime on these datasets. When pre-warmed, σOS-MR starts in a realm which already contains all the VMs in the VPC. When running from a cold start, σOS-MR starts in a realm which contains

Dataset	Dataset Size (GB)	σOS Execution Time (sec), Cold Start	σOS Execution Time (sec), Pre-warmed	Corral Execution Time (sec)
wiki-wc-2G	2	75.86	69.89	68.75

Table 6.1: End-to-end execution time of MR-wordcount built on σOS and on Corral. σOS's cold-start execution time includes time spent scaling up the realm from half a VM (one core) to 8 VMs (16 cores), whereas σOS's pre-warmed execution time does not include the time spent scaling up the realm.

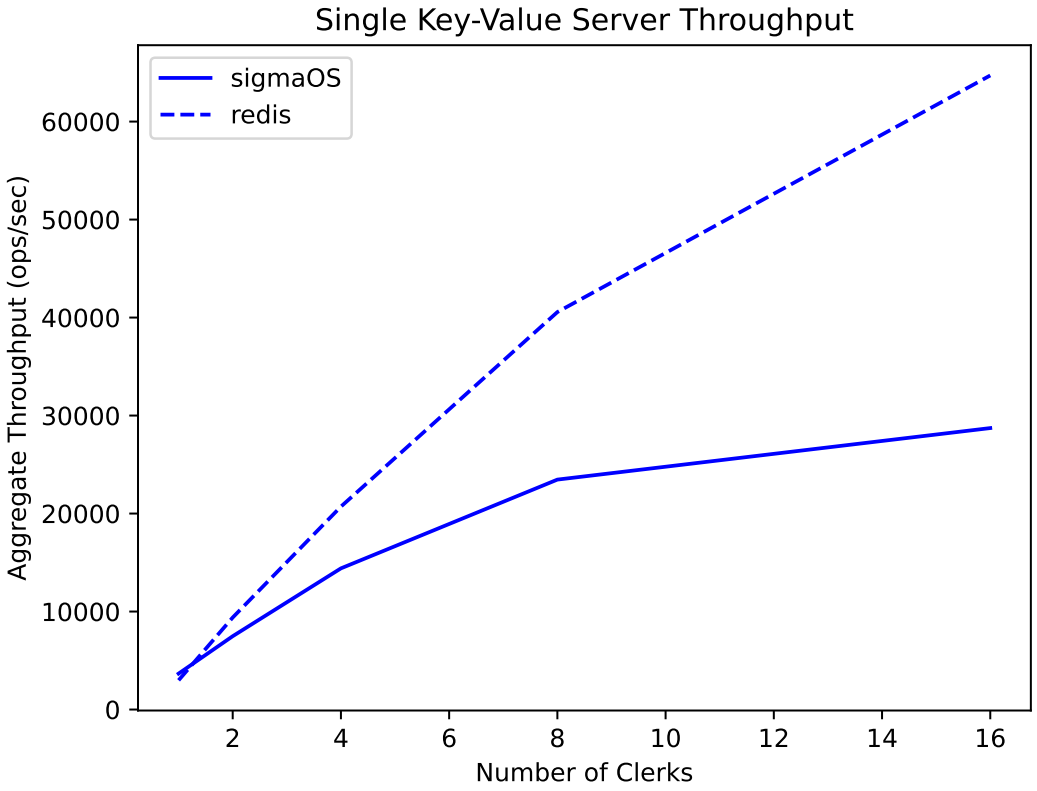

Figure 6-4: Aggregate throughput of KV and Redis servicing Get and Set requests for 90 seconds from varying numbers of clerks. Both KV and Redis are pinned to a single core.

only half of a VM (1 core), and the execution time includes the time spent waiting for σMGR to notice the growing `proc` queue and scale up the realm several times. σOS and Corral achieve similar performance when running on the wiki-wc-2G dataset. σOS's mapper lambdas achieve up to 28 MB/s streaming throughput, whereas Corral's mappers achieve up to 31.49 MB/s streaming throughput.

σOS-KV vs. Redis. Applications built direclty on Linux benefit from the ability to tune system performance via the Linux syscall interface, while σOS limits applications to the σOS API. This section seeks to determine the cost of using σOS abstractions to build a KV service, in which shards are implemented as directories, keys as files, and moving shards is implemented by copying directories.

To this end, we compare the performance of the σOS-KV service to Redis a highly-optimized, popular in-memory KV service written in C. One of Redis' features-of-merit is its ability to gracefully scale its aggregate throughput as its load increases. Since Redis applies operations on a single thread, whereas σOS-KV can apply multiple operations in parallel, we make the comparison fair by pinning Redis and σOS-KV to a single dedicated core for the purpose of this comparison.

Figure 6-4 shows how σOS-KV's and Redis' throughput scales under increasing number of clients. σOS-KV scales less gracefully than Redis, reaching 44% of the throughput achieved by Redis when servicing 16 clerks. This is partially due to the σP. While clerks accessing σOS-KV are able to perform a Set or Get in a single RPC, a single σOS-KV Get or Set RPC does more work than a single Redis RPC. A σOS-KV Get or Set involves a server-side directory `Walk`, `Open`, `Read` or `Write`, and a `Close`. Moreover, σOS-KV incurs additional overhead from the Go runtime and garbage collection which Redis elides by being implemented in C.

6.4 Microbenchmarks

In order to understand the performance of σOS further, we break down the performance of a few critical σOS components, namely the cost of spawning `proc`s and reading and writing data to and from σOS services, and compare them to corresponding operations on other platforms.

Spawning `proc`s. σOS supports burst-parallelism and low startup latency in order to rapidly scale tenants' applications. We evaluate how well σOS performs by comparing it to two other cloud computing platforms, Kubernetes and AWS Lambda.

The burst-parallelism experiment starts `proc`s that spin in a tight loop, and measure how long it takes for each platform to spawn, initialize, and run spinning `proc`s.

We run σOS and Kubernetes on a 10-machine Cloudlab cluster containing 720 cores total. We start one σOS realm with half of one machine assigned to the realm, and then burst-spawn 720 LC spinning `proc`s. For Kubernetes, we start a Kubernetes node on each machine, and deploy 720 pods consisting of a dockerized version of the spinning `proc`. For Lambda, we invoke 720 Lambda functions running the containerized spinning `proc`. We run the experiment on each platform with and without pre-warming. Pre-warming on σOS involves pre-downloading the `proc` binaries, whereas pre-warming for Kubernetes and Lambda involved pre-pulling docker images and pre-executing Lambdas respectively.

Table 6.2 shows how much time σOS, Kubernetes, and AWS Lambda take to burst-spawn 720 spinning `proc`s, with and without pre-warming. σOS's startup time

Platform	Latency (sec), No Pre-warming	Latency (sec), Pre-warming	Latency (sec), Realm Pre-growth
σOS	22.07	6.65	2.03
Lambda	2.26	1.79	-
Kubernetes	104	80	-

Table 6.2: Time required to burst-spawn 720 spinning `proc`s which each consume a single core, with and without prewarming (pre-downloading resources needed to run a `proc`, such as binaries and docker images), on σOS, Kubernetes, and Lambda. In the first two columns, σOS starts with half a machine (32 cores) assigned to the realm, and its burst-spawn latency includes the time required for σMGR to detect contention and grow the realm to 720 cores. In σOS Latency with Realm Pre-growth, the time taken to scale the realm to 32 cores is omitted.

in the first two columns includes the time spent waiting for σMGR to notice the increase in the realm's load and grow the realm several times until it encompasses all 720 cores in the cluster, as well as the time spent initializing a new node and σOS kernel on each machine. σOS's startup time in the Realm Pre-growth column does not include the time taken to scale up the realm. σOS burst-spawns the spinning `proc`s 7.26× faster than Kubernetes without pre-warming and 14.5× faster than Kubernetes with pre-warming.

The majority of the time is spent waiting for the spinning `proc`s' pods to be scheduled on a Kubernetes node. After a pod is scheduled, it takes 12.71 seconds on average for the pod to begin the container initialization process. Kubernetes starts a new container for each spinning `proc`, and traces show that initializing a new container and its namespaces takes less than one second on average.

When running with the realm pre-grown, σOS burst-spawns procs .24 seconds slower than AWS Lambda. On average, σOS takes 19 milliseconds to spawn and wait for a `proc` to start, with a standard deviation of 7 milliseconds, and Lambda takes 124 milliseconds on average, with a standard deviation of 163 milliseconds. We conclude that σOS provides burst parallelism comparable to existing cloud computing platforms. Although σOS takes longer to scale up realms, σOS offers much less variable spawn times and much lower average spawn times than Lambda.

Reading and writing data. Another critical σOS operation is reading and writing data to and from σOS kernel services. We measure client-side throughput while reading and writing files to and from a variety of σOS services. Table 6.3 presents the throughput of σOS services running on an AWS VPC.

σOS Service	Synchronous 2MB Read Throughput (MB/sec)	Asynchronous 2MB Read Throughput (MB/sec)	Synchronous 2MB Write Throughput (MB/sec)	Asynchronous 2MB Write Throughput (MB/sec)
S3	0.24	48.26	18.35	47.93
UX (local)	29.07	314.91	28.64	339.10
UX (remote)	8.86	226.17	8.30	228.05
MemFS (local)	31.00	453.72	26.48	189.97
MemFS (remote)	10.18	288.64	11.22	138.45

Table 6.3: Perfrormance of σOS kernel services when reading and writing files 2MB files synchronously and asynchronously on an AWS VPC. Local benchmarks colocate the client and service, whereas remote benchmarks place the client on one machine and the service on another.

Chapter 7

Related work

The shift from in-house computing to cloud computing has led to rapid innovation in academia and industry, and σOS builds on a large body of related work. σOS's distinguishing feature is its elastic realms, which shift the burden of resource provisioning from the tenant to the provider.

Lambdas and serverless applications Computing with lambdas has become popular, because users don't have to provision servers and can scale their applications easily with demand. Major cloud providers support it [1, 9, 23, 33] and there are also open-source platforms [3] available. Although lambdas are intended for event-triggered applications (e.g., file transcoding, WSGI web apps, etc.), developers have used lambdas in creative ways for other applications. Some examples include: ExCamera uses lambdas to process video using burst parallelism [21]; `gg` uses lambdas to parallelize desktop applications such as `make` [20]. Starling uses lambdas to speedup database query processing [39]; PyWren [27] and Locus [41], an extension to PyWren, use lambdas for data analytics.

Many of these systems have creative methods to work around lambdas' shortcomings. For example, ExCamera uses a proxy virtual machine to arrange for communication between lambdas. Several (e.g., `gg`, Starling, Locus) applications have creative ways of shuffling data between lambdas efficiently. Researchers also proposed new serverless frameworks that remove limitations. For example, Kappa helps split applications into lambdas and can checkpoint long-running lambdas [56]. Beldi [55] enables developers to write composable, fault-tolerant servereless workflows by ensuring exactly-once semantics with a transactional API. Faasm supports stateful lambdas by sharing memory and a distributed object store [50] between lambdas. Rather than try to circumvent lambdas' shortcomings, σOS takes a clean-slate approach. σOS makes `proc` coordination and state management a first-class concern, and addresses both by providing `proc`s with the realm namespace and the Unix-like `proc` API. This enables σOS to support applications with group and point-to-point communication patterns as well as stateful applications that must coordinate in the presence of failures, while preserving the elasticity resource provisioning transparency of lambdas.

Actor frameworks [5, 9, 34] are similar to lambdas in that they encourage developers to structure applications as a series of short functions which store long-lived state

in a persistent storage service. Unlike lambdas, actor frameworks make applications' objects first-class citizens, and expose RPCs as "methods" or function calls on an instance of a class in the application. σOS gives applications more flexibility in managing their state: `proc`s can be stateful and long-lived. σOS's Unix-like process API allows developers to express a wider set of applications natively and conveniently.

Elasticity σOS uses techniques from cluster management frameworks such as AWS Elastic Beanstalk [45], AWS Fargate [46] Kubernetes [4, 24], and Docker swarms [15] to determine when to scale a tenant's cluster and load-balance applications.

Moreover, these frameworks limit application elasticity. In Kubernetes, for example, developers deploy applications as a collection of containers called a pod, and pre-declare resource requests and limits for each pod at application launch time. Kubernetes cannot vertically scale application pods or change their resource limits while they are running. In order to resize pods, Kubernetes evicts and restarts them, which wastes any state which the application's pods may have built up. Kubernetes can scale applications horizontally by starting new instances of existing pods, but pods cannot share resources such as storage volumes, leading to stranded resources.

These frameworks force developers to choose between overprovisioning their applications' pods, and selecting tighter resource limits at the risk of overloading the application during spikes in load. σOS relieves developers of this burden; the realm abstraction makes it easy for σOS to resize applications vertically *and* horizontally. σOS asks tenants to declare the resources their `proc`s will require at peak load, and then transparently reallocates resources to other tenants when load drops. Realms allow σOS to monitor variety of indications of application load such as queue length, CPU utilization, and application-defined load metrics, and respond by transparently shifting the boundaries between physical machines.

Harvest VMs [2] are evictable and resizable VMs which Azure runs on physical machines with unallocated resources. Tenants lease Harvest VMs at discounted rates, as they are not guaranteed a fixed resource allocation; Harvest VMs grow and shrink as set of unallocated resources changes. Reserachers have proposed using Harvest VMs as a compute substrate for serverless computing platforms [57].

σOS draws inspiration from Harvest VMs and applies some of their ideas to the design of `procd`, which manages a resizable set of cores on each of a tenant's nodes. Unlike Harvest VMs, σOS is able to introspect into application load because all `proc`s in σOS communicate through the realm. σOS exploits this visibility to allocate free resources to tenants' nodes based on need, not just resource availability.

Single system image σOS adopts the single-system vision for each tenant's realm. Having one unified namespace for all of a tenant's resources is a key feature which enables σOS to easily grow and shrink realms while still allowing `proc`s to interact. Distributed systems such as Amoeba [52], Cambridge distributed computing system [35], Clouds [13], Plan9 [40], Sprite [36], and V [8], were pioneers of this vision. However, these systems targeted a different computing environment and applications than σOS. They were built to time-share a cluster of computers among a group of users, focusing on developing code and writing papers. In this setting fault tolerance was less important, whereas σOS targets datacenters and highly-available services

for which automatic recovery from failures is important. σOS follows the Plan9 approach of "everything is a file" but extends Plan9's 9P protocol [25] with support for ephemeral files, fences, and watches to handle failures and allow `procs` to coordinate in the presence of crashes.

New designs and APIs for cloud computing Schwarzkopf et al. argued for revisiting distributed operating systems for warehouse-scale data centers [44]. σOS falls in this line of research, with the goal of simplifying development of elastic applications, and managing multiple tenants' resource allocations in order to attain high resource utilization.

LegoOS [48] is a new OS design for datacenters built with disaggregated hardware resources so that, for example, one machine can take advantage of unused memory on another machine. AIFM [43] supports memory disaggregation at the application runtime level using remotable pointers. Both of these techniques simplify application resource provisioning by allowing applications to scale far beyond the boundaries of a single machine, and use remote resources transparently. σOS also enables transparent access to remote resources through the realm abstraction, which allows the provider to provision these resources transparently and on-demand.

A Berkeley view on serverless computing identifies several shortcoming of today's serverless computing platforms, including sharing, coordination, and communication between lambdas [28]. σOS is a partial answer to some of the identified shortcomings.

Lee and Ousterhout proposed granular computing in which applications are composed of many short-lived tasks that compute on the order of 10-100 μs [30]. This simplifies bin-packing for providers, as task boundaries serve as frequent opportunities to re-balance load across their infrastructure. σOS focuses on supporting and composing both long- and short-running millisecond-scale applications. Providers are able to shift tenants' resource allocations transparently in σOS because the realm abstraction allows `procs` to communicate with each other and access the realm's resoruces even as resource allocations change.

Cafarella et al. argue for a database OS in which all operating system state is represented uniformly as database tables on which stateless tasks operate using queries [6]. σOS instead focuses on multi-tenant elasticity. σOS takes a file-oriented approach and allows state to be stored in a decentralized way, and exposes access to the decentralized state via the realm's namespace.

Pemberton et al. argue for new API for cloud computing [38] that is RESTless. σOS is a proposal for such a new RESTless API.

Isolation The σOS API, σP, which `procs` use to communicate with system services is narrow, and simplifies security policy enforcement in σOS realms. Applications within a single realm can enforce mandatory access control policies using standard filesystem-like permissions to restrict access, and self-certifying pathnames and sessions serve as the basis for authentication in σOS. Since `procs` communicate excluslively via σP, σOS is able to use sandboxing via seccomp filters [17] to provide isolation between `procs`. Kubernetes, Borg, and Omega provide container-based isolation.

Scheduling σOS's hybrid push-pull-style decentralized scheduling within a realm is inspired by Sparrow [37]. Similarly to Borg [54] and Caladan [22], the σOS scheduler allows users to mark jobs as Latency Critical (LC) or Best Effort (BE). σOS use techniques from Borg, such as overcommitment and eviction, to drive up cluster-wide utilization during lulls in load while prioritizing LC tasks during load spikes. Caladan ensures strict performance isolation between LC and BE tasks and adjusts intra-server resource allocations in order to minimize interference.

RackSched [58] is a microsecond-scale scheduler which preserves low latency and high throughput for both high- and low-dispersion workloads. RackSched gives the abstraction of a rack-scale computer by using a two-level scheduling scheme, in which a programmable Top-of-Rack (ToR) switch routes request in order to achieve inter-server load balancing and intra-server schedulers ensure high resource utilization and avoid head-of-line blocking. σOS also seeks to give a tenant the abstraction of a logical, rack-scale computer within a realm. However, σOS focuses on millisecond-scale tasks, and takes a decentralized approach to scheduling.

Chapter 8

Conclusion

The past decade has been marked by a series of major shifts in how developers write and deploy applications: in-house computing has slowly given way to cloud computing; manual application deployment has been supplanted by cluster management frameworks; and manual resource provisioning is being replaced by new programming models, like serverless, which offload the burden of scaling and resource provisioning to the cloud provider.

σOS seeks to drive this last shift by providing a better interface and set of abstractions for tenants and cloud providers to collaboratively deploy and run applications. σOS shifts the burden of resource provisioning from developers to cloud providers using realms. The realm abstraction allows developers to write and run applications with a Unix-like API without worrying about how and where they run, or how to predict their resource usage. The realm abstraction also gives cloud providers the ability to introspect into applications' load when making global resource allocation decisions, and use this information to drive up resource utilization while preserving application performance.

Through the design, implementation, and evaluation of σOS, this thesis takes a step towards making data centers more efficient, and making cloud applications easier to write.

Bibliography

[1] Amazon. Aws lambda. `https://aws.amazon.com/lambda/`.

[2] P. Ambati, I. Goiri, F. Frujeri, A. Gun, K. Wang, B. Dolan, B. Corell, S. Pasupuleti, T. Moscibroda, S. Elnikety, M. Fontoura, and R. Bianchini. Providing SLOs for Resource-Harvesting VMs in cloud platforms. In *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*, pages 735–751. USENIX Association, Nov. 2020.

[3] Apache OpenWhisk. Open source serverless cloud platform. `https://openwhisk.apache.org/`.

[4] B. Burns, B. Grant, D. Oppenheimer, E. Brewer, and J. Wilkes. Borg, omega, and kubernetes. *ACM Queue*, 14(1), 2016.

[5] S. Bykov, A. Geller, G. Kliot, J. R. Larus, R. Pandya, and J. 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.

[6] M. J. Cafarella, D. J. DeWitt, V. Gadepally, J. Kepner, C. Kozyrakis, T. Kraska, M. Stonebraker, and M. 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.

[7] Y. Cheng, Z. Chai, and A. Anwar. Characterizing co-located datacenter workloads: An alibaba case study. In *Proceedings of the 9th Asia-Pacific Workshop on Systems*, APSys '18, New York, NY, USA, 2018. Association for Computing Machinery.

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

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

[10] Corral. Ben congdon. `https://github.com/bcongdon/corral`.

[11] R. Cox, R. Griesemer, R. Pike, I. L. Taylor, and K. Thompson. The go programming language and environment. *Commun. ACM*, 65(5):70–78, apr 2022.

[12] R. Danilak. Why energy is a big and rapidly growing problem for data centers, 2017.

[13] P. Dasgupta, R. J. LeBlanc, M. Ahamad, and U. Ramachandran. The clouds distributed operating system. *Computer*, 24(11):34–44, Nov. 1991.

[14] J. Dean and S. Ghemawat. Mapreduce: simplified data processing on large clusters. *Communications of the ACM*, 51(1):107–113, Jan. 2008.

[15] Docker. Docker swarms. https://docs.docker.com/engine/swarm/.

[16] D. Duplyakin, R. Ricci, A. Maricq, G. Wong, J. Duerig, E. Eide, L. Stoller, M. Hibler, D. Johnson, K. Webb, A. Akella, K. Wang, G. Ricart, L. Landweber, C. Elliott, M. Zink, E. Cecchet, S. Kar, and P. Mishra. The design and operation of CloudLab. In *Proceedings of the USENIX Annual Technical Conference (ATC)*, pages 1–14, July 2019.

[17] J. Edge. A seccomp overview. https://lwn.net/Articles/656307/, Sept. 2015.

[18] etcd.io. Etcd raft. https://github.com/etcd-io/etcd/.

[19] X. Feng, J. Shen, and Y. Fan. Rest: An alternative to rpc for web services architecture. In *2009 First International Conference on Future Information Networks*, pages 7–10, 2009.

[20] S. Fouladi, F. Romero, D. Iter, Q. Li, S. Chatterjee, C. Kozyrakis, M. Zaharia, and K. 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.

[21] S. Fouladi, R. S. Wahby, B. Shacklett, K. V. Balasubramaniam, W. Zeng, R. Bhalerao, A. Sivaraman, G. Porter, and K. 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, Mar. 2017. USENIX Association.

[22] J. Fried, Z. Ruan, A. Ousterhout, and A. Belay. Caladan: Mitigating interference at microsecond timescales. In *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*, pages 281–297. USENIX Association, Nov. 2020.

[23] Google. Cloud functions. https://cloud.google.com/functions.

[24] Google. Kubernetes. http://kubernetes.io/.

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

[26] P. Hunt, M. Konar, F. P. Junqueira, and B. 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.

[27] E. Jonas, Q. Pu, S. Venkataraman, I. Stoica, and B. Recht. Occupy the cloud: Distributed computing for the 99In *Proceedings of the 2017 Symposium on Cloud Computing*, SoCC '17, page 445–451, New York, NY, USA, 2017. Association for Computing Machinery.

[28] E. Jonas, J. Schleier-Smith, V. Sreekanti, C.-C. Tsai, A. Khandelwal, Q. Pu, V. Shankar, J. Carreira, K. Krauth, N. Yadwadkar, J. E. Gonzalez, R. A. Popa, I. Stoica, and D. A. Patterson. Cloud programming simplified: A berkeley view on serverless computing, 2019.

[29] M. Krohn. Building secure high-performance web services with OKWS. In *Proceedings of the 2004 USENIX Annual Technical Conference*, pages 185–198, Boston, MA, June–July 2004.

[30] C. Lee and J. Ousterhout. Granular computing. In *Proceedings of the Workshop on Hot Topics in Operating Systems*, HotOS '19, page 149–154, New York, NY, USA, 2019. Association for Computing Machinery.

[31] C. Lu, K. Ye, G. Xu, C.-Z. Xu, and T. Bai. Imbalance in the cloud: An analysis on alibaba cluster trace. In *2017 IEEE International Conference on Big Data (Big Data)*, pages 2884–2892, 2017.

[32] D. Mazières. *Self-certifying File System*. PhD thesis, Massachusetts Institute of Technology, May 2000.

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

[34] P. Moritz, R. Nishihara, S. Wang, A. Tumanov, R. Liaw, E. Liang, M. Elibol, Z. Yang, W. Paul, M. I. Jordan, and I. 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, Oct. 2018. USENIX Association.

[35] R. M. Needham and A. J. Herbert. *The Cambridge Distributed Computing System*. Addison Wesley, 1983.

[36] J. K. Ousterhout, A. R. Cherenson, F. Douglis, M. N. Nelson, and B. B. Welch. The sprite network operating system. *Computer*, 21(2):23–36, Feb. 1988.

[37] K. Ousterhout, P. Wendell, M. Zaharia, and I. 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.

[38] N. Pemberton, J. Schleier-Smith, and J. E. Gonzalez. The restless cloud. In *Proceedings of the Workshop on Hot Topics in Operating Systems*, HotOS '21, page 49–57, 2021.

[39] M. Perron, R. Castro Fernandez, D. DeWitt, and S. 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.

[40] R. Pike, D. Presotto, S. Dorward, B. Flandrena, K. Thompson, H. Trickey, and P. Winterbottom. Plan 9 from Bell Labs. *Computing Systems*, 8(3):221–254, Summer 1995.

[41] Q. Pu, S. Venkataraman, and I. 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, Feb. 2019. USENIX Association.

[42] D. M. Ritchie and K. Thompson. The UNIX time-sharing system. *Communications of the ACM*, 17(7):365–375, July 1974.

[43] Z. Ruan, M. Schwarzkopf, M. K. Aguilera, and A. Belay. AIFM: High-Performance, Application-Integrated far memory. In *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*, pages 315–332. USENIX Association, Nov. 2020.

[44] M. Schwarzkopf, M. P. Grosvenor, and S. Hand. New wine in old skins: The case for distributed operating systems in the data center. In *Proceedings of the*

4th Asia-Pacific Workshop on Systems, APSys '13, New York, NY, USA, 2013. Association for Computing Machinery.

[45] A. W. Services. Aws elastic beanstalk. `https://aws.amazon.com/elasticbeanstalk/`.

[46] A. W. Services. Aws fargate. `https://aws.amazon.com/fargate/`.

[47] C. Severance. Roy t. fielding: Understanding the rest style. *Computer*, 48(06):7–9, jun 2015.

[48] Y. Shan, Y. Huang, Y. Chen, and Y. Zhang. Legoos: A disseminated, distributed os for hardware resource disaggregation. In *Proceedings of the 13th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, page 69–87, Carlsbad, CA, Oct. 2018.

[49] A. Shebabi, S. Smith, D. Sartor, R.Brown, M. Herrlin, J. Koomey, E. Masanet, N. Horner, I. Azevedo, and W. Lintner. United united states data center energy usage report, 2016.

[50] S. Shillaker and P. 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.

[51] N. S onnischsen. Global data centers energy demand by type 2015-2021, 2021.

[52] A. S. Tanenbaum, R. van Renesse, H. van Staveren, G. J. Sharp, and S. J. Mullender. Experiences with the Amoeba distributed operating system. *Commun. ACM*, 33(12):46–63, Dec. 1990.

[53] M. Tirmazi, A. Barker, N. Deng, M. E. Haque, Z. G. Qin, S. Hand, M. Harchol-Balter, and J. Wilkes. Borg: The next generation. In *Proceedings of the Fifteenth European Conference on Computer Systems*, EuroSys '20, New York, NY, USA, 2020. Association for Computing Machinery.

[54] A. Verma, L. Pedrosa, M. Korupolu, D. Oppenheimer, E. Tune, and J. Wilkes. Large-scale cluster management at Google with Borg. In *Proceedings of the 10th ACM EuroSys Conference*, pages 18:1–18:17, Bordeaux, France, Apr. 2015.

[55] H. Zhang, A. Cardoza, P. B. Chen, S. Angel, and V. 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, Nov. 2020.

[56] W. Zhang, V. Fang, A. Panda, and S. 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.

[57] Y. Zhang, I. n. Goiri, G. I. Chaudhry, R. Fonseca, S. Elnikety, C. Delimitrou, and R. Bianchini. Faster and cheaper serverless computing on harvested resources. In *Proceedings of the ACM SIGOPS 28th Symposium on Operating Systems Principles*, SOSP '21, page 724–739, New York, NY, USA, 2021. Association for Computing Machinery.

[58] H. Zhu, K. Kaffes, Z. Chen, Z. Liu, C. Kozyrakis, I. Stoica, and X. Jin. RackSched: A Microsecond-Scale scheduler for Rack-Scale computers. In *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*, pages 1225–1240. USENIX Association, Nov. 2020.