Evaluating SigmaOS with Kubernetes for Orchestrating Microservice and Serverless Applications

by

Yizheng He

B.S. Electrical Engineering, University of Pennsylvania, 2015
B.S. Finance, University of Pennsylvania, 2015

Submitted to the System Design & Management Program
in partial fulfillment of the requirements for the degree of

MASTER OF SCIENCE IN ENGINEERING AND MANAGEMENT

at the

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

September 2023

Authored by:	Yizheng He System Design & Management Program August 18, 2023
Certified by:	M. Frans Kaashoek Charles Piper Professor of Electrical Engineering and Computer Science Thesis Supervisor
Accepted by:	Joan S. Rubin Executive Director, System Design & Management Program

Evaluating SigmaOS with Kubernetes for Orchestrating Microservice and Serverless Applications

by

Yizheng He

Submitted to the System Design & Management Program on August 18, 2023

in Partial Fulfillment of the Requirements for the Degree of

MASTER OF SCIENCE IN ENGINEERING AND MANAGEMENT

ABSTRACT

SigmaOS is a new multi-tenant cloud operating system that simplifies distributed application development. Its design centers around the novel concepts of `realms` and `procs`. A `realm` presents a tenant with a shared global namespace that hides the machine boundaries. Tenants structure their applications as process-like `procs` interacting through the `realm`'s namespace. `Procs` are lightweight, stateful, and can communicate. SigmaOS manages the scheduling and execution of `procs` to achieve high resource utilization and performance isolation.

This thesis compares SigmaOS with Kubernetes, a mainstream cloud operating system, using a microservice-style social network website and a serverless image resizing program. It measures their performances on a small-scale cluster in CloudLab. The SigmaOS version of the social network is easier to build (30% fewer lines), and its image resizing starts faster (25% - 89%). SigmaOS performs comparably to Kubernetes regarding latency and resource consumption when running a single application but provides better performance isolation when running multiple applications in separate `realms`: latency increases by 4-11% with concurrent applications in SigmaOS versus over 150% in Kubernetes.

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

Acknowledgments

I sincerely thank everyone who has guided and supported me throughout my journey towards a master's degree. The SDM faculties and staff created a fantastic program that allows me to explore and continue learning after years in the industry. This work only exists because of Professor Kaashoek's patient guidance, profound dedication, and Ariel Szekely's impressive prior efforts.

I want to thank my family for loving and caring for me during my MIT journey, as you always have. I feel incredibly fortunate to be the kid of Xiulan Ma and Qing He and to have met and married Mengxi Tan. You are the source of my strength and motivation.

I want to thank all my friends who inspired, listened, and encouraged me. I also want to thank my colleagues, mentors, and friends at BlackRock for supporting me in pursuing this degree.

Biographical Sketch

Yizheng He was born in Shenyang, China. He received dual B.S. degrees in electrical engineering and finance from the University of Pennsylvania's Jerome Fisher Program in 2015. Then, he joined BlackRock, Inc.'s Financial Modeling Group as a quantitative engineer and became senior engineer in 2019 and lead engineer in 2022. In 2021, while working at BlackRock, he joined MIT's System Design & Management program as a master's student.

At Penn, Yizheng researched robotics and competed in the RoboCup Standard Platform League, aka the World Cup for humanoid robots. At BlackRock, he developed and managed an optimization software system that solves some of the world's most complex portfolio construction problems. At MIT, he explored multiple domains, including operations research, distributed systems, and engineering management. He worked with the Parallel and Distributed Operating System group for this thesis.

Yizheng is a history reader, tennis player, and avid soccer fan in his free time. In December 2020, Yizheng married Mengxi Tan, an artist, gamer, product manager, and Penn alum.

Contents

List of Figures

Chapter 1

Introduction

Despite the prevalence of cloud computing, programming and deploying cloud applications remains challenging. Developers must orchestrate multiple machines through systems like Kubernetes or use frameworks like AWS Lambda that restrict functionality. SigmaOS, a new cloud operating system under development by MIT's Parallel and Distributed Operating Systems (PDOS) group, hopes to eliminate this trade-off. This thesis, submitted to MIT's System Design and Management Program, evaluates SigmaOS by comparing it with Kubernetes, an existing solution, using two applications of different types. Specifically, this thesis summarizes the motivation and related work in Chapter One, reviews SigmaOS's design and implementation in Chapter Two, introduces the two applications in Chapter Three, presents the experiment results in Chapter Four, and concludes the findings in Chapter Five.

1.1 Motivation

A typical cloud tenant executes various software on multiple machines: long-running analytics, short background job queues, fleets of stateless and stateful servers, sharded database servers, and more. Running multiple applications on a shared cluster of machines presents challenges, including initial configuration and start-up, coordination and communication among a changing set of entities, and re-assigning machine resources (CPU and memory) from long-running batch applications to latency-critical applications in response to increasing loads. In an ideal world, a cloud operating system would hide such complexity from applications, much like single-machine operating systems hide details of RAM, disk, and CPU management under easy-to-use abstractions.

One route is to use a cloud orchestration system like Kubernetes [1], [2]. Kubernetes takes on many burdens of managing a cluster, from deciding where to run each program to adjusting resources in response to load. However, much of the underlying hardware details leak through Kubernetes' abstractions. For example, a developer must set up a naming system for inter-service communication and configure a container image with custom mount points to launch a `pod`, Kubernetes' basic deployment unit. As a result, a multi-service deployment unusually requires a complex specification with many machine-level details. A more abstract model would be welcome.

An alternative is to rely on a serverless computation framework like AWS Lambda [3].

This model simplifies deployment by requiring as little as writing an ordinary function call, and the cloud provider hides all distribution aspects. However, The serverless model places tight restrictions on how functions can interact with each other and how long they can execute. A more flexible model would be welcome.

SigmaOS [4] is a novel cloud operating system under development at MIT PDOS that simplifies application development and deployment by combining the benefits of the above approaches. Programs start process-like `procs` without being aware of where they execute, and `procs` can communicate, keep state, and provide services without location knowledge. `Procs` of a tenant run in a `realm`, which provides a separate single system image with a hierarchical namespace that mediates `procs` activities. SigmaOS allocates physical machine resources to `realms`, isolates `realms` from each other, and decides where to execute each `proc`. The result is flexibility and ignorance of distribution reminiscent of a single-machine operating system.

This thesis evaluates SigmaOS using Kubernetes to validate whether SigmaOS meets its design expectations. We build two applications in the two systems. One is a communication-heavy social network website formed by many microservices derived from DeathStarBench [5], and another is a computation-heavy image resizing program with parallel serverless tasks based on an open-source algorithm [6]. We benchmark the SigmaOS applications against comparable versions in Kubernetes on CloudLab [7] and measure lines of code, start times, performance, and ability to time-share. We analyze performance differences and investigate whether they result from design choices or implementation approaches.

1.2 Contributions

The contributions of this thesis are:

1. Implementations of a social network application consisting of microservices in SigmaOS and Kubernetes based on DeathStarBench.

2. Comparison of the lines of codes, start times, single and multi-application performances of two systems deployed on CloudLab, and analyses of the impact of SigmaOS's design and implementation choices.

1.3 Related Work

SigmaOS combines the benefits of cloud orchestration systems and serverless systems. It places few restrictions on workloads like the former and abstracts away machine-level details like the latter. SigmaOS achieves these benefits through `realms` and `procs`. This section discusses related works to SigmaOS.

Orchestration Systems:

The most basic way to manage a cluster of machines in a cloud environment is to statically provision virtual machines (VMs) and then configure and maintain them by hand. As cloud

systems scale, however, this approach becomes unwieldy. This difficulty has led to the widespread adoption of cloud orchestration systems to automatically manage machines and their resources, including Borg [8], Mesos [9], Omega [10], Kubernetes [1], [2], and Docker swarms [11].

Most orchestration systems operate at a machine level of abstraction and expose containers [12] or VMs to users. This property preserves compatibility with existing software but makes deploying applications hard because developers must maintain low-level details such as container volumes, IP addresses, and network ports. Furthermore, creating a container or VM is an expensive operation that limits how fast and well such systems can respond to changes in load.

Serverless Systems:

Serverless systems offer an opposite trade-off by hiding low-level details at the expense of reduced compatibility. Platform as a service (PaaS) systems execute code repositories on behalf of customers and automatically scale them through an HTTP web frontend and load balancer [13]. Function as a service (FaaS) like AWS Lambda [3] can execute arbitrary compiled code as functions and supports a richer set of event triggers, such as HTTP, timeouts, task queues, and storage modifications [14], [15].

Serverless systems impose severe constraints on tasks that they can handle. For example, PaaS requires customers to use specific programming languages or runtime environments. FaaS limits the execution time of each function. Both approaches require tasks to be stateless, do not allow direct communication between tasks, and generally assume tasks will run on a constrained set of memory and CPU resources.

Improving Serverless

Researchers have explored ways to address the limitations of serverless systems. For example, ExCamera [16] and gg [17] use a proxy virtual machine for communication between lambdas. Several systems, like gg, Starling [18], and Locus [19]), introduce efficient techniques to shuffle data between lambdas. Faasm supports stateful lambdas by sharing memory and a distributed object store [20] between lambdas. MXFaaS [21] multiplexes machine resources among serverless functions to reduce overheads and improve image resizing performance. The Actor framework [22]–[24] structures applications as short functions that store long-lived states in a persistent storage service.

Performance Benchmarks

Researchers and developers have built benchmark applications to test the hardware and operating system performances in cloud environments. This thesis refers to DeathStarBench [5], an open-source suite of cloud microservices, for its design of the social network website.

Chapter 2

SigmaOS Design and Implementation

SigmaOS aims to simplify cloud computing by allowing cloud providers to manage resources on behalf of tenants. The dual concepts of `procs` and `realms` abstract away machine boundaries. The SigmaOS kernel, which includes distributed schedulers, namespace managers, and external resource proxies, dynamically allocates resources to `realms` and their `procs`. This chapter explains SigmaOS's design and provides a brief overview of its implementation.

2.1 Design of `procs` and `realms`

`procs`: Process-like Deployment Units

SigmaOS uses a lightweight and location-agnostic deployment unit called `procs` to free applications from container- or virtual machine-level details. `Procs` are analogies of processes in single-machine environments. Since processes are less complicated than containers or VMs, `procs` take less effort to start.

```
1    proc := &Proc{"do-task", args, env}
2    sigmaOS.Spawn(proc)
3    sigmaOS.WaitStart(proc.Pid())
4
```

Figure 2.1: Spawning a Child `proc`

Applications can create, destroy, and manage `procs` through a group of SigmaOS APIs[1]. Each `proc` has a unique process identifier `pid` and a corresponding descriptor with the binary location, arguments to the binary, and environment variables. Figure 2.1 shows an example of spawning and waiting for a `proc` in the Go programming language [25].

Developers specify whether a `proc` performs latency-critical (LC) or best-effort (BE) work, following Borg [8]. For LC `procs`, the developer specifies the number of CPU cores it may utilize at peak load. SigmaOS prioritizes LC `procs` when resources are insufficient.

[1] Figure A.1 in the Appendix provides the full list of `proc` APIs.

`realms`: Per-tenant Single System Image

SigmaOS `procs` must be able to handle various types of tasks like containers or VMs do. For instance, they may need to maintain state, send requests to peers, or query external databases. SigmaOS introduces `realms` to facilitate the orchestration of such activities. Every `proc` resides in a `realm`, and each `realm` provides a shared namespace to its `procs` like a system map. Figure 2.2 is an example namespace of a `realm`. The pathnames have universal meanings to the `realm`'s `procs` regardless of their physical locations.

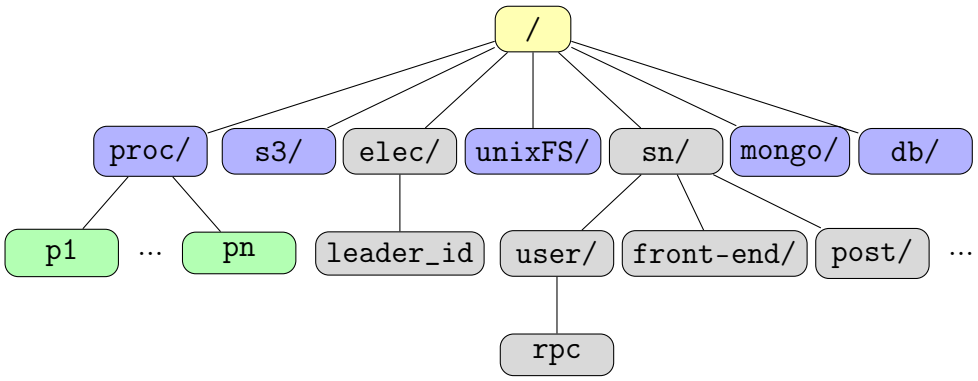


Figure 2.2: Example Namespace of a `realm`

```
1    mnt := MkMountServer(MY_ADDR)
2    sigmaOS.Create("/sn/user", WRITE, OWNER)
3    sigmaOS.Mount("/sn/user", mnt)
4    sigmaOS.MkRPCDev("/sn/user/rpc", service)
5
```

(a) Server Side

```
1    var req UserLoginRequest
2    fd := sigmaOS.Open("/sn/user/rpc", OWNER)
3    res := sigmaOS.RPC(fd, &req)
4
```

(b) Client Side

Figure 2.3: `procs` Performing an RPC

`procs` use the namespace for many purposes. For instance, they can check the status of others by accessing `/proc/` or coordinate leader election through a small custom file `/elec/leader_id`.

To extend the namespace with new services, a `proc` provides remote procedure call (RPC) based services and advertises its existence by mounting itself in the namespace. Figure 2.3a shows an example user service in Go. It creates a symlink `/sn/user/` that points to an in-memory file system and then creates an RPC device in this file system at `/sn/user/RPC`. This device can encode or decode messages and invoke RPC handlers. Figure 2.3b shows how a client `proc`, such as an HTTP front-end, sends a request to the user service. It opens the

RPC device at `/sn/user/RPC`, which auto-mounts the user service, and then sends requests by writing to the file descriptor of the RPC device. The client and server perform the RPC without knowing their physical locations.

`Procs` can advertise utility services using the same mechanism. For example, `/db` and `/mongo` in Figure 2.2 are mount points of proxies that communicate with an SQL database and a MongoDB instance. `/s3` and `/unixFS` expose Amazon's S3 storage and the host machine's file system. By combining `procs` and `realms`, SigmaOS enables cloud programs to interact and explore resources as if they run on top of a single machine's operating system.[2]

2.2 SigmaOS Kernel

The SigmaOS kernel comprises `procs` supporting essential system functions. As shown in Figure 2.4, one kernel runs on top of the Linux operating system in each physical machine of the cluster.

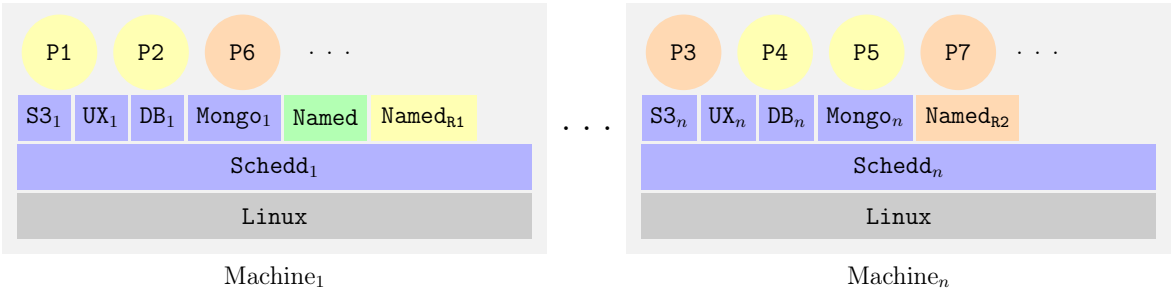


Figure 2.4: The SigmaOS Kernel. User `procs` (circles) run on top of kernel `procs` (rectangles) on top of Linux. `Procs` P1, P2, P4, P5 belong to `realm` R1 (yellow). `Proc` P3, P6, P7 belong to `realm` R2 (orange).

The most essential function of the kernel is to schedule different `realms`' `procs`. Specifically, the kernel must decide which `procs` to execute next and where and how to execute them. There are three high-level scheduling goals. First, prioritize LC `procs` over BE `procs`. The former should experience minimal interference from the latter. Second, achieve high resource utilization. Third, achieve fairness so that `realms` can properly time-share limited resources.

A group of distributed `Schedd` `procs`, one on each machine, fulfill the scheduling function. When a user `proc`'s life cycle starts, it first enters a scheduling queue on one of the machines and waits for running. The `Schedd` on each machine monitors the local queue and executes queuing `procs` if there is enough local CPU and memory. `Schedds` also jointly maintain a global queue so that machines with idle resources can steal `procs` waiting for execution on busy machines. The distributed nature of `Schedds` improves scalability and reduces latency: tenant programs interact only with a local `Schedd` to spawn `procs`, and `Schedds` interact only with remote counterparts in case of inadequate local resources.

When starting a `proc`, `Schedds` arrange the underlying Linux' cgroups [26] to label BE `procs` as `SCHED_IDLE` [27], Linux's lowest-priority scheduling class. As a result, LC `procs`

[2]Figure A.2 in the Appendix provides the full list of namespace APIs.

receive almost all CPU cycles when busy, and BE ones take CPU cycles only when the former are idle. To fully utilize resources, `Schedds` slightly oversubscribe CPU resources on each machine by running additional BE `procs`.

For fairness, `Schedds` divide resources evenly among any `realms` that have LC `procs` and then divide the remaining resources evenly among `realms` with BE `procs`. Within each step, they choose which `realm`'s `proc` to start next through a round-robin[3].

A centralized `Named` `proc` implements the root namespace. When a tenant creates a new `realm`, the kernel starts a sub-`Named` `proc` on one of the machines, which builds the `realm`'s namespace on top of the root one and serves it to the `realm`'s `procs`. In addition, the kernel provides access to file systems and data storage through proxy `procs` on all machines, including `S3`, `Ux`, `Db`, and `Mongo`. The combination of `Named` and proxy `procs` enable user `procs` in SigmaOS to navigate services and resources regardless of their locations.

2.3 SigmaOS Implementation

`procs` take the form of Linux processes, but they are limited to a few system calls: allocating and freeing memory, sending and receiving messages on sockets, and creating threads. Tenants provide a statically linked binary for each `proc` they start.[4]

`procs` of different `realms` run in separate Docker [28] containers to ensure isolation between `realms`. Since `procs` have no dependency on file system image, on each machine, SigmaOS co-locates `procs` from the same `realm` in one image-less container.

From a `proc`'s perspective, it goes through a four-step start sequence after `Schedds` decide to run it on a particular machine. First, SigmaOS instantiates a `proc` object on the target machine and sets up its environment variables. Second, if the `proc` has not previously run on the machine, SigmaOS downloads its binary from designated locations. Third, if no other `proc` of the same `realm` has previously run on the machine, SigmaOS starts a `proc` manager for this `realm` and initializes a container for the `proc`. Fourth, the `proc` manager starts the `proc`'s binary in the container as a Linux process. Starting a `proc` cold requires all steps, but a warm start, in contrast, needs only the first and the last. Therefore, warm starts in SigmaOS are as fast as starting processes on a local Linux machine.

[3]SigmaOS currently pursues this simple fairness policy for research purposes. For commercial usage, it may divide resources by tenant's payments.

[4]This section primarily focuses on the implementation of `procs`. A previous M.S. thesis [4] provides details on other aspects of SigmaOS's implementation.

Chapter 3

Applications

We evaluate SigmaOS by comparing it with Kubernetes, a widely adopted cloud orchestration system, as a benchmark. The evaluation process includes constructing mirrored versions of applications for both systems and comparing their performances under similar conditions. This chapter covers the applications, and the next chapter presents the evaluation results.

Since SigmaOS supports microservice- and serverless-style applications, we choose to build a social network website and an image resizing program. The social network adopts a microservice structure with multiple small servers, each supporting one type of function. It involves heavy communication among its components, and its load on the cloud operating system scales with the number of requests. In contrast, resizing images is a classical serverless and computation-heavy task. It requires little configuration or communication but may quickly claim CPU resources when processing inputs of considerable size.

3.1 A Microservice-based Social Network

We port an open-source C++ implementation of the social network [29] in DeathStarBench to SigmaOS and Kubernetes. We rewrite the application in Go and maintain its architecture and functions.

Architecture

Figure 3.1 shows the application's structure: a group of microservices interact through RPCs to support the features of a social network jointly. An HTTP front-end on top receives client requests from external and routes them to corresponding internal service providers. The microservices store application data in a persistent MongoDB instance, including user credentials, post timelines, and post contents. They rely on a sharded key-value storage to cache intermediate results.

Figure 3.2 lists the microservices' individual functionality. Processing most requests requires internal chitchat. For instance, when a user posts a status update, the `compose` service queries the `text` service to process text contents, and the latter queries `url` and `user` services to shorten URLs in the text and resolve notifications like "@user". The `compose` service then asks the `post` service to store the status update and informs the `home` service to add this update to followers' and notify users' homepages. In this process, the `compose`

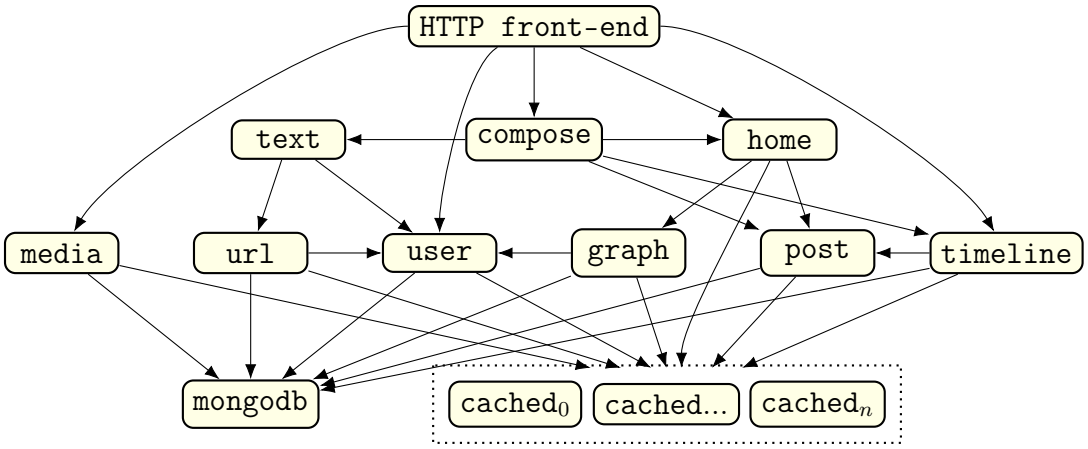


Figure 3.1: Social Network Architecture. Each text box represents a microservice. An arrow from A to B means A is an RPC client of B.

service spends more time waiting for responses from others than conducting local operations. As a result, when the application experiences heavy loads, communication overheads may dominate end-to-end performance.

Service Name	Functions
User	Register new users; Check user existence; Process logins
Graph	Follow and un-follow; Check followers and followees
Post	Store and read posts
Timeline	Update and read a user's timelines
Home	Update and read a user's home page
Url	Shorten URLs from post texts; Store and read URLs
Media	Store and read media contents
Text	Recognize URLs and "@user" in post texts
Compose	Process new posts from user
Front End	Route client HTTP requests to services

Figure 3.2: Social Network Microservice Functions

Implementations

In SigmaOS, we structure each microservice as a `proc` in the same `realm`. They advertise themselves by creating self-pointing symlinks in the `realm`'s namespace and perform RPCs by opening the destination services' symlinks, as in Figure 2.3. In Kubernetes, we place each service in a `pod`, communicating through gRPC [30]. Although the two implementations use different RPC frameworks, they adopt identical communication interfaces in Protocol Buffer [31].

The Kubernetes implementation includes an additional Registry `pod`, which fulfills the functions of a `realm`'s namespace in SigmaOS: when a microservice first starts, it must send messages to the registry to expose itself and queries the registry to initialize connection channels to peers.

Figure 3.3 shows simplified application configurations in the two systems. In SigmaOS, to start the social network, we use a Go program that spawns service `procs` in an order corresponding to their dependencies. In Kubernetes, we use a YAML file to configure the container details for each `pod` and `kubectl` commands to invoke the YAML files to start the application.

```
    servers := []Server{usersrv, graphsrv, ...}
    for _, srv := range servers {
        p := &Proc(srv.Bin, srv.Args, srv.Envs, srv.CPU)
        sigmaOS.Spawn(p)
        sigmaOS.WaitStart(p.Pid())
    }

```

(a) SigmaOS

```
    apiVersion: apps/v1
    kind: Deployment
    spec:
        replicas: 1
        template:
            spec:
                containers:
                    image: XXXXX
                    name: socialnetwork-user
                    command: user
                ports:
                    containerPort: 9999
                resources:
                    requests:
                        cpu: XXXm

```

(b) Kubernetes

Figure 3.3: Social Network Configurations. Top: SigmaOS starts all `procs` in a loop. Bottom: YAML configuration file one `pod` in Kubernetes.

3.2 Serverless Image Resizing

We port an open-source image resizing algorithm [6] in Go to to build a resizing service. The service loads input images from local file systems or AWS S3 buckets and conducts multiple resizing tasks in parallel. The tasks do not interact with the controller and run for a limited time, forming a classic example of a serverless application.

In SigmaOS, we use a manager program in Go to orchestrate concurrent tasks. Upon receiving resizing requests, it creates a worker `proc` for each request and monitors their progress. In Kubernetes, we use a YAML file to place resizing tasks into individual `pods`. The file also regulates the total number of tasks, level of parallelism, and input locations. Figure 3.4 shows simplified configurations in both systems.

```
    taskIds := make([]Pid, 0)
    for _, req := resizeRequests {
        p := &Proc("resize-image", req.InputPath, req.Envs)
        sigmaOS.Spawn(p)
        taskIds = append(taskIds, p.Pid())
    }
    go monitorJobs(taskIds)

```

(a) SigmaOS

```
    apiVersion: batch/v1
    kind: Job
    spec:
        completions: XXX
        parallelism: XXX
        template:
            spec:
                containers:
                    name: img-resize
                    image: XXXXX
                    env:
                        <AWS credentials>
                    args:
                        "s3://XXXXX"

```

(b) Kubernetes

Figure 3.4: Image Resizing Configurations

Chapter 4

Evaluation

4.1 Evaluation Questions

Our evaluation approach is comparison experiments. We provide Kubernetes, the existing player, and SigmaOS, the novel challenger, with the same hardware resources and measure their performances when running the same applications and conducting the same tasks. We then conduct quantitative and qualitative analyses on performance differences and determine whether they result from design mechanisms, implementation choices, or external factors. In addition, we review application codes in both systems to gain insights from a cloud developer's perspective. Following the design objectives of SigmaOS, we aim to answer four questions:

1. **Is SigmaOS easier to program?** We measure the total lines of code of the social network excluding comments. We expect the SigmaOS version of the social network to have fewer lines. We skip the image resizing program due to its small size.

2. **Does SigmaOS start `pods` faster?** We measure the time it requires to start an image resizing task in Kubernetes `pod` or a SigmaOS `proc`, in either cold or warm conditions. We expect the start time of `procs` to be smaller. We skip the social network because start times have little impact on its long-running microservices.

3. **How does SigmaOS perform running one application?** We measure the request throughput, latency, and CPU consumption of the social network and the completion time and CPU consumption of the image resizing program.[1] We expect the SigmaOS applications to perform at least on the same level as those in Kubernetes.

4. **How does SigmaOS perform running multiple applications?** We conduct measurements similar to the previous question but with both applications running simultaneously. We also measure performance isolation, defined as the scale of performance drop when an application runs concurrently with other programs. We expect SigmaOS to perform at least similarly to Kubernetes in both measures.

[1] Performance evaluations in this thesis do not cover the applications' memory profiles as they are not memory-bound, but this is an interesting domain for future studies.

4.2 Programmability

To evaluate whether `realms` and `procs` simplify cloud programming, we calculate the lines of code of the social network in both Kubernetes and SigmaOS using cloc [32], an open-source counter. Figure 4.1 shows that while achieving the same functionalities, the SigmaOS version of the application has 30% fewer lines (2,138 vs. 3,094) than the Kubernetes version.

Application Section	Language	K8s LoC	σOS LoC
Interfaces			
RPC Definitions	Protobuf	222	206
Implementations			
User Service	Go	219	200
Graph Service	Go	266	243
URL Service	Go	176	147
Text Service	Go	163	132
Media Service	Go	176	145
Post Service	Go	190	163
Home Service	Go	179	153
Timeline Service	Go	183	156
Compose Service	Go	200	163
Front-end	Go	324	321
Service Registry	Go	118	–
Total		2,194	1,823
Configurations			
Service IP & Ports	JSON	18	–
Service Deployment	YAML	660	–
Start Up Code	Go	–	109
Total		678	109
Total	–	3,094	2,138

Figure 4.1: Social Network Lines of Code

To understand the source of SigmaOS's smaller number of lines of code, we decompose the social network code into three sections: interface definitions, core implementations, and configurations.

The interface definitions cover the communication protocols among the microservices. They entirely depend on the functionalities of microservices and are thus agnostic to the underlying cloud operating systems. We use Protocol Buffer to program this section in both versions and observe similar lines of code (206 vs. 222).

Implementations of the microservices form the essential section of the social network, and the SigmaOS version is 17% (1,823 vs. 2,194) shorter for two reasons. First, the Kubernetes version needs a 118-line registry service to keep track of the services, while the SigmaOS kernel already fulfills this function through a `realm`'s namespace. Second, services in Kubernetes have to send messages to the registry to expose themselves and connect with peers. Meanwhile, in SigmaOS, they achieve a similar goal through a much shorter call of

the SigmaOS API. As a result, most services in Kubernetes require 30-40 more lines than corresponding ones in SigmaOS.

The configuration files regulate the application's start-up procedure in a cloud environment, and in this section, we observe the most significant difference (109 vs. 678) between the two systems. To set up the social network in Kubernetes, we must allocate a unique port number to each microservice and create a YAML file to specify container details of a service's hosting `pod`. In contrast, in SigmaOS, we use only a short program to start microservices as `procs` while the SigmaOS kernel manages network specifications.

While counting lines of code is not a perfect measure of programmability, the decomposition analysis suggests that SigmaOS reduces the burden for cloud developers by allowing them to design fewer modules and maintain fewer settings. By providing a universal namespace in each `realm` and hiding network details when scheduling `procs`, SigmaOS migrates significant configuration efforts from the application level to the cloud operating system level.

4.3 Start Time

By design, process-like `procs` are more lightweight than container-based `pods` and are therefore cheaper to start. To verify, we compare the start time of a `pod` in Kubernetes and that of a `proc` in SigmaOS. Specifically, the `pod` and the `proc` have identical functionalities of resizing an image, and we measure the start times when the systems are cold and warm. For both systems, cold starts happen when they run a `pod` or `proc` for the first time, and warm starts happen when they rerun a previously known `pod` or `proc`. Figure 4.2 shows that SigmaOS `pods` start faster in both scenarios.

Measurements	**Kubernetes** `Pod`	**SigmaOS** `Proc`
Cold Start		
Environment Setup (ms)	–	6.2
Binary Download (ms)	2,877.7	1,749.6
`Proc` Manager Initialization (ms)	–	593.4
`Pod`/`Proc` Start (ms)	282.4	25.7
Total (ms)	3,160.1	2,374.9
Warm Start		
Environment Setup (ms)	–	2.6
Binary Download (ms)	–	0.9
`Pod`/`Proc` Start (ms)	247.8	24.1
Total (ms)	247.8	27.6

Figure 4.2: Image Resizing Task Start Times

We decompose the warm and cold start-up sequences for both systems, and Figure 4.3 summarizes the definitions of each step. During cold starts, binary download times dominate in both systems, and the total time in SigmaOS is 25% (2,375 vs. 3,160) shorter. While the systems download binaries from different storage services in this experiment, SigmaOS downloads a much smaller object than Kubernetes since the `procs` binaries are smaller than

Operations	Definition
Environment Setup*	Initialize a `proc` object and set up environment variables
Binary Download	Download a `pod`'s container image, or a `proc`'s binary
`Proc` Manager Initialization*	Start a local service for `procs` of the same `realm`
`Pod`/`Proc` Start	Start a `pod`'s container or a `proc`'s binary

* Only Applicable to SigmaOS

Figure 4.3: Start-up Sequence Definitions

Properties	Kubernetes `Pod`	SigmaOS `proc`
Binary Type	Docker Image	Unix Binary
Binary Storage	Docker Registry	AWS S3 Bucket
Binary Size (MB)	30.9	10.6

Figure 4.4: Binary Details

containers. Furthermore, the measurement for SigmaOS, though better, may be unfair as it includes the `proc` manager initialization time, which is part of the system start-up instead of the `proc`'s start-up. An equivalent step may exist in Kubernetes, but Kubernetes does not provide a public API to measure it.

Thanks to their caches, both systems spend zero or close-to-zero time downloading binaries during warm starts. SigmaOS also skips the `proc` manager initialization as it only needs to start once per `realm`. As a result, the actual times to start a `pod` and `proc` dominate, and SigmaOS has a clear advantage (27.6 vs. 247.8). This significant edge suggests that SigmaOS is more suitable to run many short-lived serverless tasks like resizing small images for which long start sequences are too expensive.

4.4 Single Application Performance

To evaluate the end-to-end performance of SigmaOS, we compare it against Kubernetes through a series of benchmark experiments running the social network website and the image resizing program. We provision both systems with a five-machine cluster on CloudLab [7], each with four 2.20 GHz cores. We allow applications to consume 16 of the 20 cores and reserve the remaining four for operating systems and databases. This section discusses the results of the systems running a single application at a time, and the next section analyzes their performances running multiple applications concurrently.

Social Network

We deploy the social network to both systems and generate the same loads. We follow DeathStartBench's original implementation for load request composition. Figure 4.5 summarizes the load test parameters. In both systems, the social network microservices may take the cluster's all available resources (16 cores).

Time Period	Load (Req/sec)
1 - 5 seconds	600 requests/sec
6 - 10 seconds	1,200 requests/sec
11 - 15 seconds	1,800 requests/sec
15 - 20 seconds	900 requests/sec

(a) Load by Time

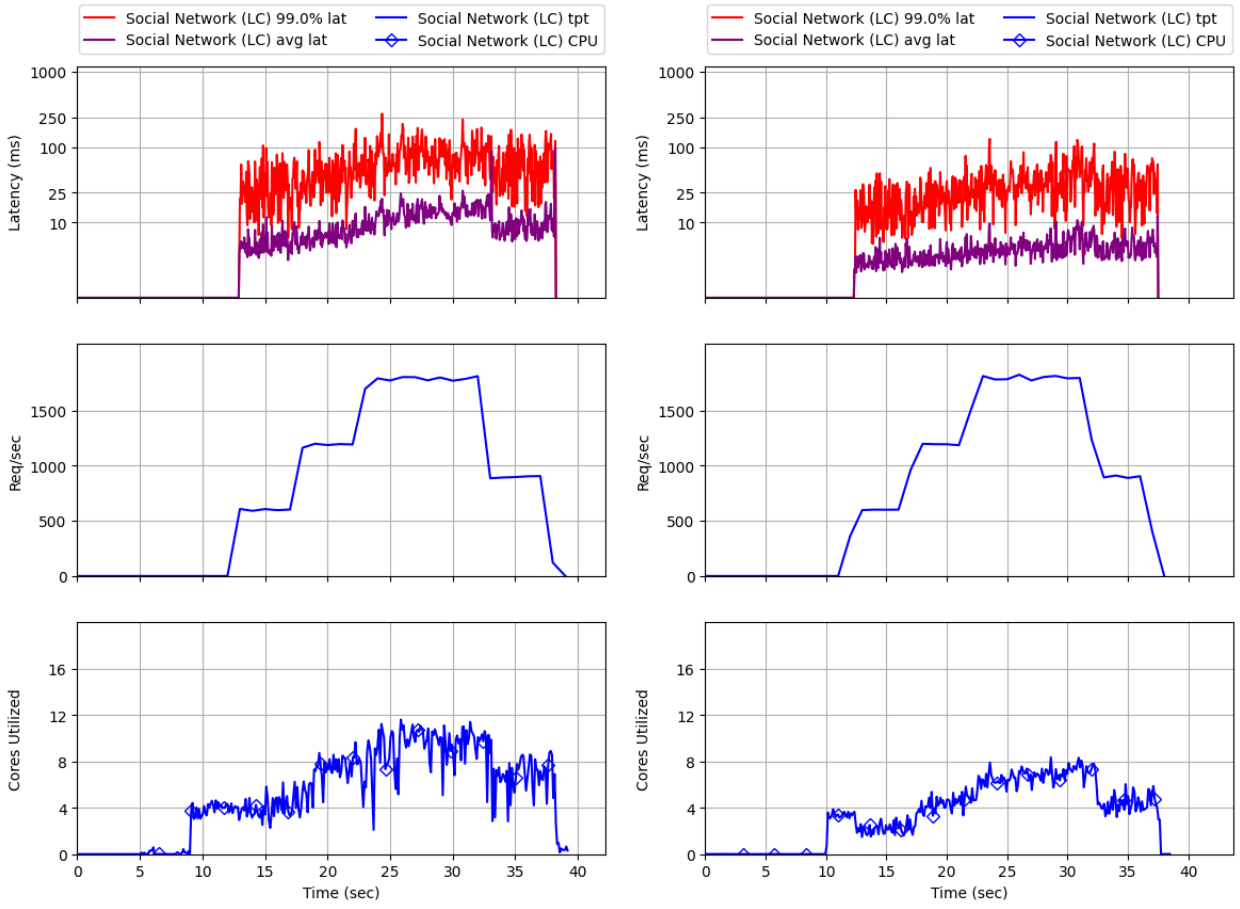
Request	Type	Weight
Compose Message	Write	10%
Read Home Page	Read	60%
Read User Timeline	Read	30%

(b) Request Composition

Figure 4.5: Social Network Test Parameters

Performance Metrics	Kubernetes	SigmaOS
Mean Latency (ms)	10.33	4.36
99% Tail Latency (ms)	84.08	40.84
Peak CPU (cores)	11.62	8.36

Figure 4.6: Social Network Performance Summary



(a) Kubernetes **(b)** SigmaOS

Figure 4.7: Social Network Performance. The top charts plot log-scale mean and tail latency by time. Middle charts plot request throughputs by time. The bottom charts plot CPU consumption by time. All charts share the time scales at the bottom.

Figure 4.6 summarizes that the SigmaOS version has better mean and tail latency while consuming less CPU. Figure 4.7 provides more details: the middle charts suggest that the two social networks achieve similar throughput that matches the input pattern from Figure 4.5a. The top charts show that the latency in Kubernetes is about twice that in SigmaOS throughout the testing period, and the bottom charts show that Kubernetes consumes about 2-3 more cores at all times.

SigmaOS's less CPU consumption aligns with the expectation that process-like `procs` are more lightweight than container-based `pods`. Specifically, SigmaOS runs all `procs` on each machine in one image-less container because they belong to the same `realm`. Kubernetes, however, must start one container for each `pod`, leading to additional CPU overheads.

For the difference in latency, we first observe that RPC times dominate the end-to-end request handling times of the social network. Specifically, for most requests, the processing time of a microservice request is much shorter than the time to create and transmit RPC messages. This observation coincides with the application's structure, where many services communicate with each other but conduct simple tasks each.

We then observe that the lower latency in SigmaOS results from shorter RPC times. To isolate this phenomenon, we measure the time spans of 5000 sequential light RPC calls in Kubernetes and SigmaOS, with RPC client and server residing in two `pods` for the former and in two `procs` for the latter. We configure server-side RPC handling to be trivial (< 1 μs) so that RPC creation and transmissions contribute to almost the entirety of time measures. Figure 4.8 shows that the average latency of a RPC call is about 50% higher in Kubernetes than in SigmaOS, with client and server on either the same or separate machines.

Measurements	**Kubernetes**		**SigmaOS**	
	Local	Network	Local	Network
Mean Latency (μs)	220	295	168	194
Standard Deviation (μs)	27	24	29	63
50% Tail Latency (μs)	221	309	165	158
99% Tail Latency (μs)	286	342	222	296

Figure 4.8: RPC Latency in Kubernetes and SigmaOS. Server and client run on the same machine for local measures and on separate ones for network measures.

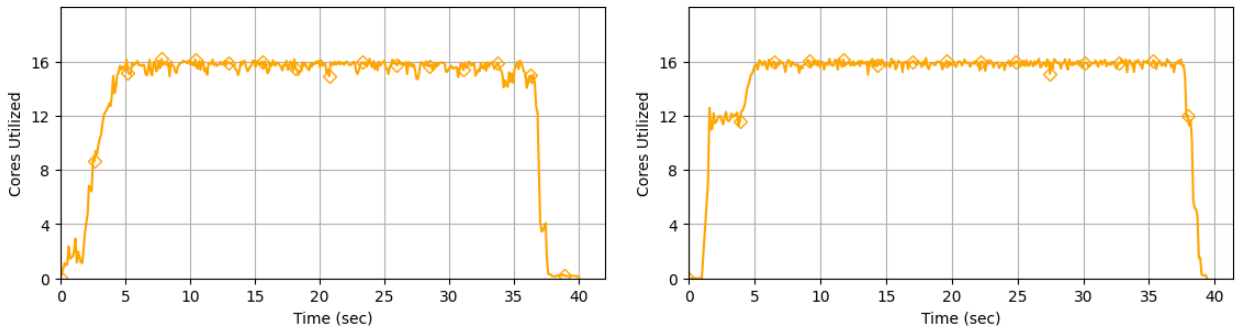
We suspect the root cause of the discrepancies in RPC latency to be the two systems' network configurations. In Kubernetes, each `pod` has its IP address and virtual network device [1], [33]. As a result, RPC messages between two `pods` have to hop three times: from the sender `pod`'s network device to the sender host machine's network device, then to that of the receiver host machine, and finally to that of the receiver `pod`. If both `pods` reside on the same machine, they skip the second step. In comparison, SigmaOS does not create distinct network interfaces for `procs`. Thus, RPC messages between `procs` hop only once between the host machines' network devices or zero times for collocated `procs`.

It is possible to make SigmaOS `realms` run on separate networks, which may level the two systems' RPC latency when the client and the server are on separate machines. However, the difference in local RPC latency will persist as collocated `procs` in the same `realm` can share the same network device. Overall, since network transmission times are sensitive to

choices in machine cluster configuration, network isolation mechanism, and network interface implementations, we do not interpret the SigmaOS's better latency as an artifact of better system design but as a different implementation choice.

Image Resizing

We run 48 concurrent image resizing tasks in Kubernetes and SigmaOS. Each task loads a 6.3MB image and resizes it 20 times. The tasks may take all 16 provisioned CPU cores, and Figure 4.9 shows that the two systems perform very closely. Since image resizing requires heavy computing power, the concurrent tasks quickly saturate CPU cores in both systems. The heavy loads of the tasks enable them to run long enough (over 30s), making the start-up cost (20 - 100ms) insignificant. With few additional bottlenecks other than CPU, the two systems take roughly the same time to complete the tasks.



(a) Kubernetes **(b)** SigmaOS

Figure 4.9: Image Resizing Performance

4.5 Multi-Application Performance

We run the social network and the image resizing program concurrently in Kubernetes and SigmaOS to examine whether they allocate resources swiftly, leaving no idle resources, and properly prioritize latency-critical applications over best-effort ones. In addition, we configure the social network as LC and image resizing as BE. In SigmaOS, we also place the two applications in separate `realms`.

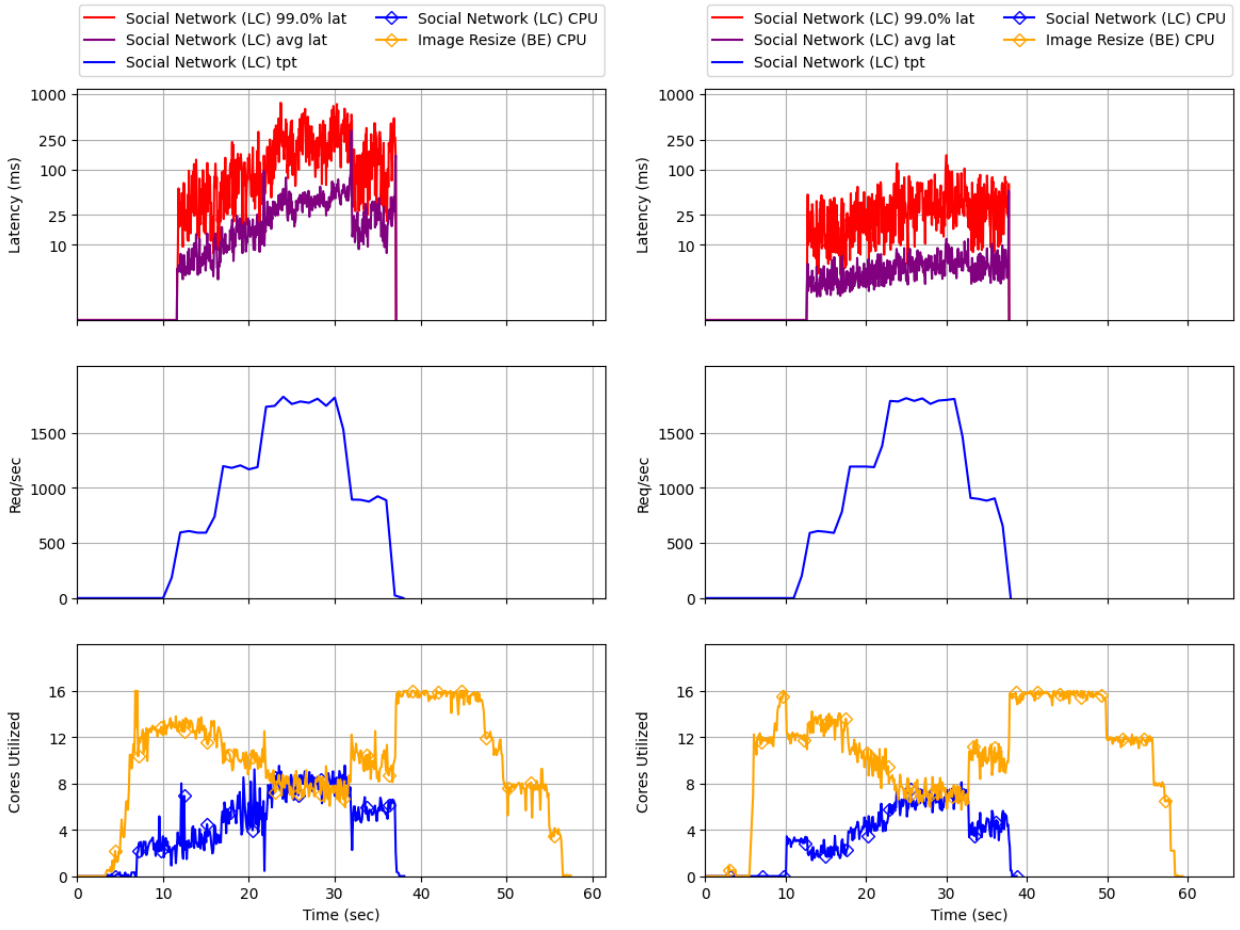
Figure 4.10 shows that in the multi-application scenario, it takes over 40% more time for the BE image resizing tasks to complete because of their low resource priorities. Meanwhile, the LC social network in SigmaOS has significantly lower mean and tail latency. When running with concurrent image resizing tasks, the SigmaOS social network experiences only a 4-11% increase in latency compared to running alone. In contrast, the Kubernetes version suffers from an over 150% increase in latency. Furthermore, with concurrent BE tasks, the Kubernetes social network consumes 17.8% less CPU at peak load moment despite its high priority, and the time it utilizes more than 7 CPU cores decreases by 40.3%. SigmaOS, however, has less than 5% drops for both measures.

Performance Metrics	Kubernetes			SigmaOS		
	Alone	Multi-App	Impact	Alone	Multi-App	Impact
Image resizing (BE)						
Total Time (s)	37.45	52.91	+41.3%	37.52	55.81	+48.7%
Social Network (LC)						
Mean Latency (ms)	10.33	26.32	+154.8%	4.36	4.82	+10.6%
99% Tail Latency (ms)	84.08	254.63	+202.8%	40.84	42.60	+4.3%
Peak CPU (cores)*	11.62	9.55	-17.8%	8.36	8.10	-3.1%
High CPU Time (s)**	15.08	9.00	-40.3%	3.57	3.41	-4.5%

* Maximum amount of cores taken by the application during load test

** Total time that the application consumes more than 7 cores

Figure 4.10: Multi-Application Performance Summary



(a) Kubernetes **(b)** SigmaOS

Figure 4.11: Social Network and Image Resizing Performance. We start the BE image resizing jobs and generate loads for the LC social network afterwards. The top charts plot the social network's log-scale mean and tail latency. Middle charts plot the social network's request throughputs. The bottom charts plot the CPU consumption of the two applications. All charts share the time scales at the bottom.

Figure 4.11 provides additional insights. The middle charts suggest that social networks in both systems achieve similar throughput as in the previous section. The top charts, when compared with those in Figure 4.7, attest to the significant latency increase of Kubernetes in the multi-application scenario. The bottom charts show that the BE image resizing tasks in both systems claim most CPU cores at the beginning of the experiments but gradually yield resources to the LC social network as the latter receives loads. Image resizing tasks reclaim CPU cores only when social network loads decrease, and the dents in their CPU consumption curves (orange) lead to longer completion times.

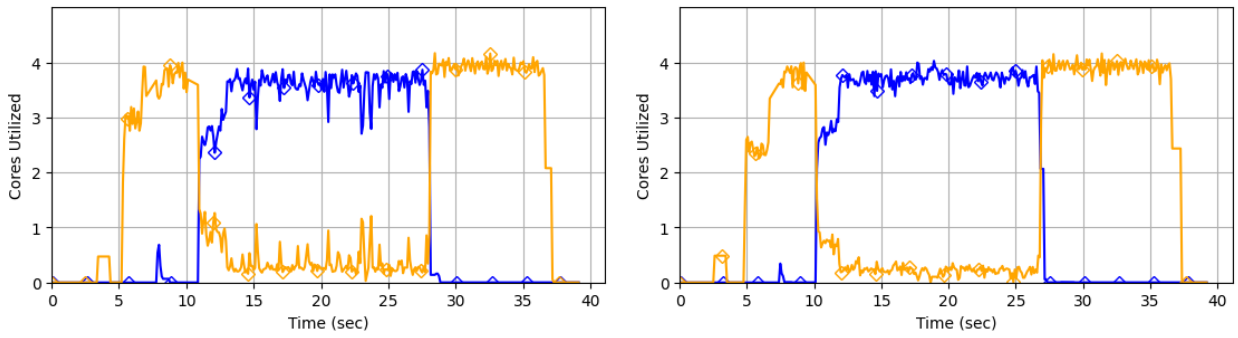
Figure 4.11's bottom charts also suggest that SigmaOS shifts resources between applications more smoothly: the CPU consumption curves in SigmaOS are less spiky than those in Kubernetes. We quantify this effect by measuring the scale of second-order derivatives of the curves in both systems. Figure 4.12 shows that the SigmaOS curves are much smoother.

Measurements	Kubernetes	SigmaOS
Social Network CPU Curve Smoothness	1.29e−4	7.05e−5
Image Resizing CPU Curve Smoothness	9.78e−5	7.78e−5

Figure 4.12: Resource Balancing Smoothness. Measured as the average absolute values of the second-order derivatives of the original curves. Smaller numbers indicate smoother curves, with linear functions having 0 smoothness measure.

We first confirm that Linux cgroup configurations do not cause the difference in performance isolation: the two systems assign similar CPU weights to LC and BE applications, with social network `pods` and `procs` receiving weights of 30 - 40 and image resizing ones receiving 1.

A critical difference in implementations between the two systems that might explain the discrepancy is SigmaOS's adoption of `SCHED_IDLE` for BE `procs`, which allows any process with higher priority to preempt their CPU usage immediately. As a result, the BE image resizing tasks in SigmaOS give up resources properly and swiftly.



(a) W/O `SCHED_IDLE` **(b)** With `SCHED_IDLE`

Figure 4.13: Impact of `SCHED_IDLE` in SigmaOS. CPU consumption of LC and BE image resizing jobs, with and without `SCHED_IDLE`

We conduct further comparison experiments between SigmaOS and a special version that does not use `SCHED_IDLE` to isolate its impact. We first run two groups of image processing

Performance Metrics in SigmaOS	Alone†	**W/O** `SCHED_IDLE` Multi-App	Impact	**With** `SCHED_IDLE` Multi-App†	Impact†
Social Network (LC)					
Mean Latency (ms)	4.36	7.85	+80.0%	4.82	+10.6%
99% Tail Latency (ms)	40.84	88.31	+116.2%	42.60	+4.3%
Peak CPU (cores)*	8.36	7.13	-14.7%	8.10	-3.1%
High CPU Time (s)**	3.57	0.21	-94.1%	3.41	-4.5%
Image resizing (BE)					
Total Time (s)	37.52	54.93	+46.4%	55.81	+48.7%

† Identical to the "SigmaOS" columns in Figure 4.10

* Maximum amount of cores taken by the application during load test

** Total time that the application consumes more than 7 cores

Figure 4.14: Impact of `SCHED_IDLE` in Multi-Application Scenario

tasks on a four-core machine, labeling one group as LC and the other as BE. Figure 4.13 shows that without `SCHED_IDLE`, the BE group consumes more CPU, and both groups have more spikes in CPU consumption curves.

We then measure the special SigmaOS's multi-application performance, using the same setup of running social network and image resizing in parallel. Figure 4.14 shows that the LC social network in SigmaOS without `SCHED_IDLE` experiences similar performance downgrades as in Kubernetes: its mean and tail latency increase by close to 100%, and its CPU consumption drops by close to 15% when running concurrent best effort tasks. These measurements confirm that `SCHED_IDLE` strongly affects SigmaOS's performance isolation.

Chapter 5

Conclusion

The thesis compares SigmaOS, a novel cloud operating system, with Kubernetes, a popular existing solution. SigmaOS hopes to simplify cloud development through `procs`, which are lightweight, and `realms`, which provide single-system images using a global namespace. SigmaOS also aims to provide swift and proper resource allocation and strong performance isolation through a group of distributed schedulers and accompanying kernel services.

Measurements in terms of line of codes and start times of the same applications in Kubernetes and SigmaOS validate that `procs` and `realms` achieve their design objectives: developers construct fewer modules and write 30% less code in SigmaOS, and applications start 25% and 89% faster in SigmaOS respectively in cold and warm conditions. Application performance measurements confirm that SigmaOS manages resources efficiently: latency-critical applications perform comparably in Kubernetes and SigmaOS when running alone but suffer only a 4-11% performance drop in SigmaOS when running concurrently with best-effort tasks, compared to over 150% in Kubernetes.

While a small number of applications and benchmark experiments cannot comprehensively evaluate an operating system's prospect, this thesis' results demonstrate SigmaOS's potential, and we hope they provide insights for future improvements and adoption of the system.

Appendix A

SigmaOS APIs

Method	Description
`Spawn(descriptor)`	Queue `proc` and return its identifier
`WaitStart(pid)`	Wait until `proc` has started
`WaitExit(pid)`	Wait until `proc` has exited
`WaitKill(pid)`	Wait until `proc` is killed
`Started(pid)`	`proc` marks itself as started
`Exited(pid, status)`	`proc` marks itself as exited
`Kill(pid)`	Kill `proc`

Figure A.1: SigmaOS `proc` APIs

Method	Description
`Create(path, perm, mode)`	Create a file, directory, link, or pipe at `path`
`Open(path, mode)`	Open a file, directory, or link at `path`
`Close(fd)`	Close an object `fd`
`Remove(path)`	Remove an object at `path`
`Rename(old, new)`	Rename an object from `old` to `new`
`Stat(path)`	Fetch info about an object at `path`
`Read(fd)`	Reads data from `fd`
`Write(fd, data)`	Write data to `fd`
`RPC(fd, data)`	Send an RPC through `fd` and read reply
`Lseek()`	Change offset of an object `fd`
`Mount(path, service)`	Advertise `service` at a mount point `path`
`ResolveMount(path)`	Resolves ∼`local` or ∼`any` to a global pathname
`OpenWatch(path, func)`	Open file at `path` or wait until file is created
`WatchDir(path, func)`	Call `func` when directory at `path` changes
`WatchRemove(path, func)`	Call `func` when `path` is removed

Figure A.2: SigmaOS Namespace APIs

References

[1] Google, *Kubernetes*, http://kubernetes.io/.

[2] B. Burns, B. Grant, D. Oppenheimer, E. Brewer, and J. Wilkes, "Borg, Omega, and Kubernetes," *ACM Queue*, vol. 14, no. 1, 2016.

[3] Amazon, *AWS Lambda*, https://aws.amazon.com/lambda/.

[4] A. Szekely, "σOS: Elastic realms for multi-tenant cloud computing.," M.S. thesis, Massachusetts Institute of Technology, Cambridge, Massachusetts, 2022.

[5] Y. Gan, Y. Zhang, D. Cheng, *et al.*, "An open-source benchmark suite for microservices and their hardware-software implications for cloud & edge systems," in *Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*, ser. ASPLOS '19, Providence, RI, USA, 2019, pp. 3–18. [Online]. Available: https://github.com/delimitrou/DeathStarBench.

[6] J. Schlicht, *Image resize in go*, https://github.com/nfnt/resize.

[7] D. Duplyakin, R. Ricci, A. Maricq, *et al.*, "The design and operation of CloudLab," in *Proceedings of the USENIX Annual Technical Conference (ATC)*, Jul. 2019, pp. 1–14. [Online]. Available: https://www.flux.utah.edu/paper/duplyakin-atc19.

[8] A. Verma, L. Pedrosa, M. Korupolu, D. Oppenheimer, E. Tune, and J. Wilkes, "Large-scale cluster management at Google with Borg," in *EuroSys '15: Proceedings of the Tenth European Conference on Computer Systems*, Bordeaux, France, Apr. 2015, 18:1–18:17.

[9] B. Hindman, A. Konwinski, M. Zaharia, A. Ghodsi, A. D. Joseph, R. Katz, S. Shenker, and I. Stoica, "Mesos: A platform for fine-grained resource sharing in the data center," in *Proceedings of the 8th USENIX Conference on Networked Systems Design and Implementation*, ser. NSDI'11, Boston, MA: USENIX Association, 2011, pp. 295–308.

[10] M. Schwarzkopf, A. Konwinski, M. Abd-El-Malek, and J. Wilkes, "Omega: Flexible, scalable schedulers for large compute clusters," in *SIGOPS European Conference on Computer Systems (EuroSys)*, Prague, Czech Republic, 2013, pp. 351–364. [Online]. Available: http://eurosys2013.tudos.org/wp-content/uploads/2013/paper/Schwarzkopf.pdf.

[11] Docker, *Docker swarms*, https://docs.docker.com/engine/swarm/.

[12] D. Merkel *et al.*, "Docker: Lightweight Linux containers for consistent development and deployment," *Linux j*, vol. 239, no. 2, p. 2, 2014.

[13] E. Keller and J. Rexford, "The "Platform as a Service" model for networking," *INM/WREN*, vol. 10, pp. 95–108, 2010.

[14] M. Shahrad, R. Fonseca, I. Goiri, G. Chaudhry, P. Batum, J. Cooke, E. Laureano, C. Tresness, M. Russinovich, and R. Bianchini, "Serverless in the wild: Characterizing and optimizing the serverless workload at a large cloud provider," in *2020 USENIX Annual Technical Conference, USENIX ATC*, 2020, pp. 205–218.

[15] A. Agache, M. Brooker, A. Iordache, A. Liguori, R. Neugebauer, P. Piwonka, and D. Popa, "Firecracker: Lightweight virtualization for serverless applications," in *17th USENIX Symposium on Networked Systems Design and Implementation, NSDI 2020, Santa Clara, CA, USA, February 25-27, 2020*, 2020, pp. 419–434.

[16] 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)*, Boston, MA: USENIX Association, Mar. 2017, pp. 363–376, ISBN: 978-1-931971-37-9. [Online]. Available: https://www.usenix.org/conference/nsdi17/technical-sessions/presentation/fouladi.

[17] 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*, ser. USENIX ATC '19, Renton, WA, USA: USENIX Association, 2019, pp. 475–488, ISBN: 9781939133038.

[18] M. Perron, R. Castro Fernandez, D. DeWitt, and S. Madden, "Starling: A scalable query engine on cloud functions," ser. SIGMOD '20, Portland, OR, USA: Association for Computing Machinery, 2020, pp. 131–141, ISBN: 9781450367356. DOI: 10.1145/3318464.3380609. [Online]. Available: https://doi.org/10.1145/3318464.3380609.

[19] 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)*, Boston, MA: USENIX Association, Feb. 2019, pp. 193–206, ISBN: 978-1-931971-49-2. [Online]. Available: https://www.usenix.org/conference/nsdi19/presentation/pu.

[20] S. Shillaker and P. Pietzuch, "Faasm: Lightweight isolation for efficient stateful serverless computing," in *2020 USENIX Annual Technical Conference (USENIX ATC 20)*, USENIX Association, Jul. 2020, pp. 419–433, ISBN: 978-1-939133-14-4. [Online]. Available: https://www.usenix.org/conference/atc20/presentation/shillaker.

[21] J. Stojkovic, T. Xu, H. Franke, and J. Torrellas, "Mxfaas: Resource sharing in serverless environments for parallelism and efficiency," in *Proceedings of the 50th Annual International Symposium on Computer Architecture*, ser. ISCA '23, Orlando, FL, USA: Association for Computing Machinery, 2023. DOI: 10.1145/3579371.3589069. [Online]. Available: https://doi.org/10.1145/3579371.3589069.

[22] 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*, ser. SOCC '11, Cascais, Portugal: Association for Computing Machinery, 2011, ISBN: 9781450309769. DOI: 10.1145/2038916.2038932. [Online]. Available: https://doi.org/10.1145/2038916.2038932.

[23] P. Moritz, R. Nishihara, S. Wang, *et al.*, "Ray: A distributed framework for emerging AI applications," in *13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 18)*, Carlsbad, CA: USENIX Association, Oct. 2018, pp. 561–577, ISBN: 978-1-939133-08-3. [Online]. Available: https://www.usenix.org/conference/osdi18/presentation/moritz.

[24] Cloudflare, *Cloudflare workers*, https://workers.cloudflare.com/.

[25] Google, "The Go Programming Language," https://golang.org/.

[26] N. Brown, *Control groups series*, https://lwn.net/Articles/604609/, Jul. 2014.

[27] Linux.Org, *Fixing sched_ idle*, https://lwn.net/Articles/805317/.

[28] Docker, *Docker*, https://www.docker.com/.

[29] DeathStartBench, *Social network benchmark*, https://github.com/delimitrou/DeathStarBench/tree/master/socialNetwork.

[30] Google, *gRPC*, https:https://grpc.io/.

[31] Google, *Protobuf*, https://developers.google.com/protocol-buffers.

[32] A. Danial, *Cloc: Count line of code*, https://github.com/AlDanial/cloc.

[33] F. Minna, A. Blaise, F. Rebecchi, B. Chandrasekaran, and F. Massacci, "Understanding the security implications of kubernetes networking," *IEEE Security & Privacy*, vol. 19, no. 5, pp. 46–56, 2021. DOI: 10.1109/MSEC.2021.3094726.