National Technical University of Athens
School of Electrical and Computer Engineering
Computer Science Division
Computing Systems Laboratory

Extending storage support for unikernel containers

Fifth International Workshop on Serverless Computing (WoSC) 2019

Orestis Lagkas Nikolos, Konstantinos Papazafeiropoulos, Stratos Psomadakis, Anastasios Nanos, Nectarios Koziris

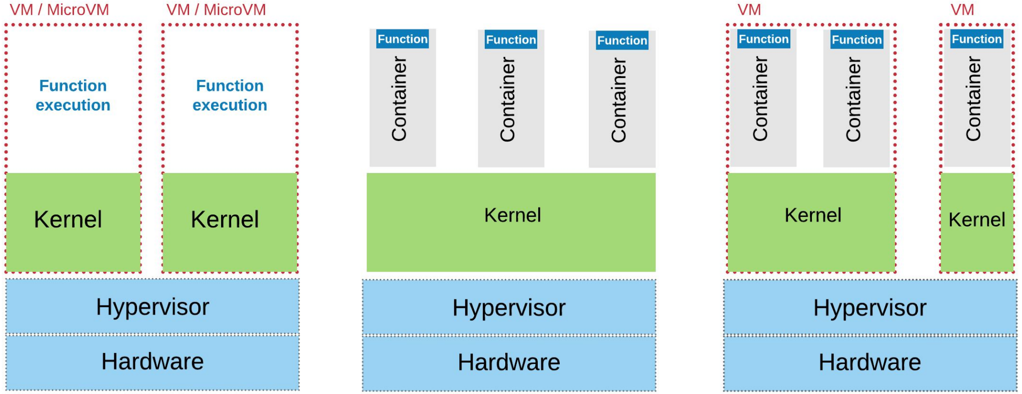
Outline

- Motivation (why we consider unikernels for Serverless)
- Background (Docker & Nabla Containers)
- Enabling storage for Unikernel Containers (classify requirements & desing)
- Experimental Results
- Summary & Conclusions (overview and future directions)

State of practice

Serverless frameworks execute functions on:

- Virtualized guests / micro VMs
  + : strict isolation, generic virtual device interfaces
  - : boot time, OS noise
- Containers on per-tenant VMs:
  + : lightweight function execution
  - : looser isolation, reduced security, footprint/function execution



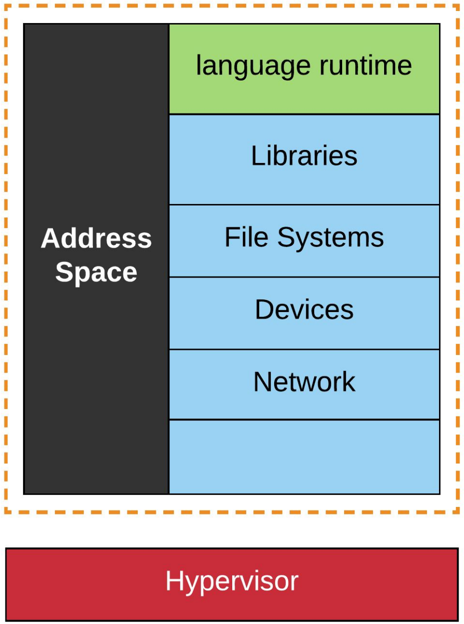
Unikernels

Pick only the absolutely necessary **OS components** for the execution of the **function** on top of the **hypervisor**.

"baked" in a **single address space** **image**
⟶ boot and run directly on the hypervisor.

+ : VM's isolation, minimal footprint, near-instant spawn time

Great fit for short-lived applications ⟶ Serverless



Storage handling in Docker Containers

Container "root" (/)→ a mount point on host.

Docker storage implements the following mechanisms:

- Layers
- Image
- Container

(Layers, Image, Container) → graphdriver

- Bind mounts of files on the host within one or more containers.

Different serverless functions → re-use container image

Storage handling in Unikernels

Unlike containers, Unikernels handle I/O (network and storage) through virtual devices.

In this work, we bridge the gap between containers and unikernels with respect to storage access, in the context of serverless computing.

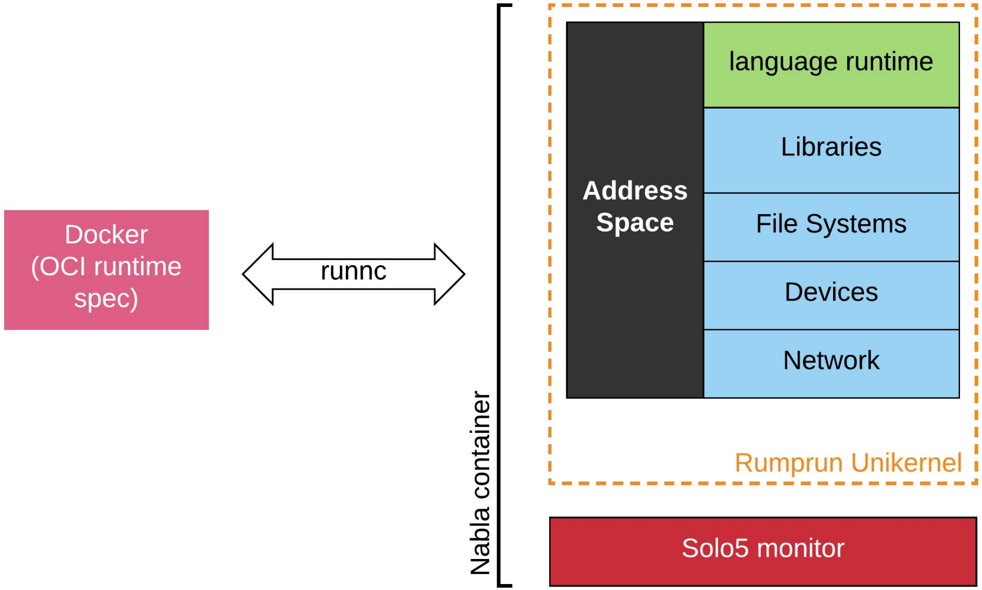
Contributions

Shareable layers among container - unikernel images:
→ reduce storage space required → run more containers per host
→ identical layers → share pages in host's page cache.

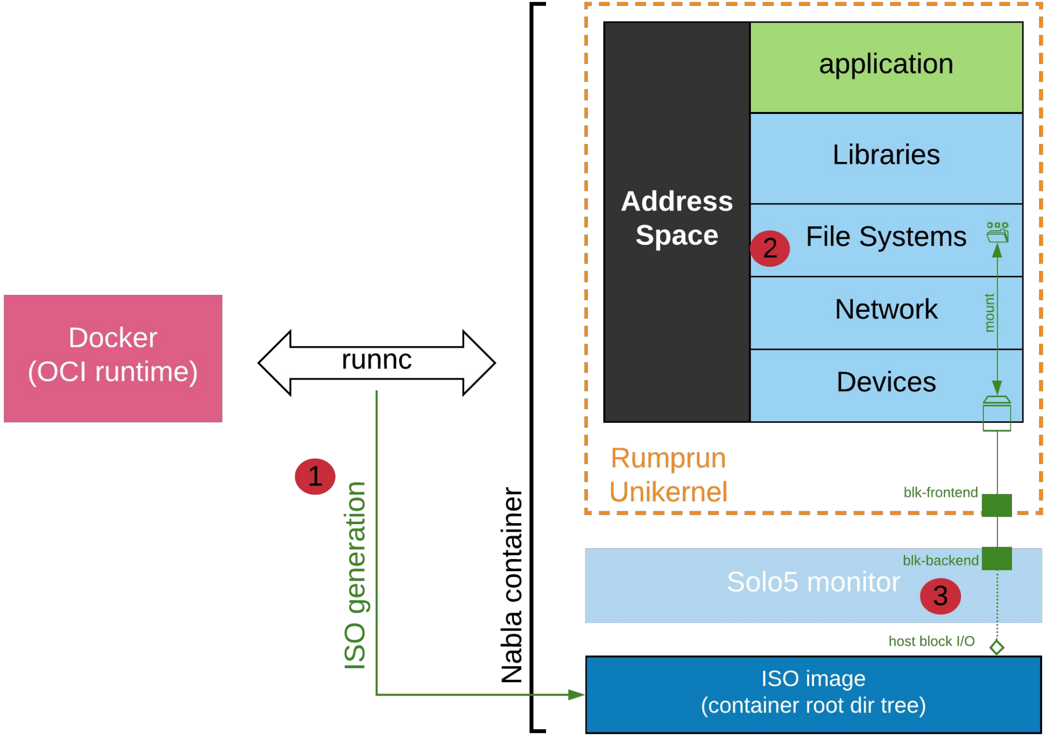
Shift the filesystem images generation in Docker build time:
→ container - unikernel starts faster

Background :: Nabla Containers

Combine the unikernel concept with benefits of the container ecosystem. Components:



Background :: Storage Handling in Nabla Containers



Our approach

Change the traditional Docker workflow to use image files instead of directory trees.

(i.e. convert vanila container's layers to image files → block devices inside the unikernel).

Our Design :: Storage classes for Unikernel Containers

To design a solution for container-unikernel storage handling, we first classify storage access of a unikernel container in four basic categories:

Application binary	*base - layer*
Library dependencies	*shareable, read-only layers*
Configuration	*bind mounts*
I/O data	*N/A*

image

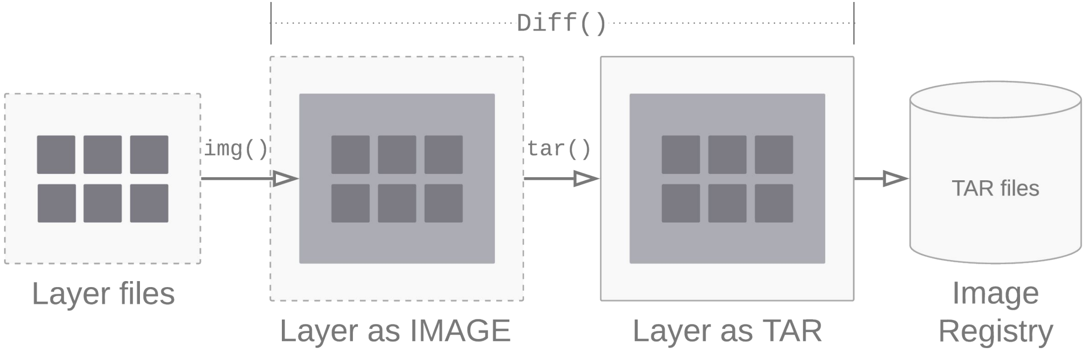
container

Our Design :: Docker graphdriver

We introduce a container - unikernel storage driver, implemented as Docker graphdriver:

- Graphdriver implements two interfaces: (a) ProtoDriver (basic capabilities)
  (b) DiffDriver (push/pull operations)

- Our Diff method implementation converts layers to image files before pushing them.



Implementation :: Extend Nabla Containers

Extend Rumprun

→ multiple virtual blocked devices → solo5 block devices

→ union mount layer image files

→ recreate layer's original directory tree

Extend Nabla runtime (runnc)

→ Docker bind mounts (currently as read-only)

Evaluation :: Spawn Time

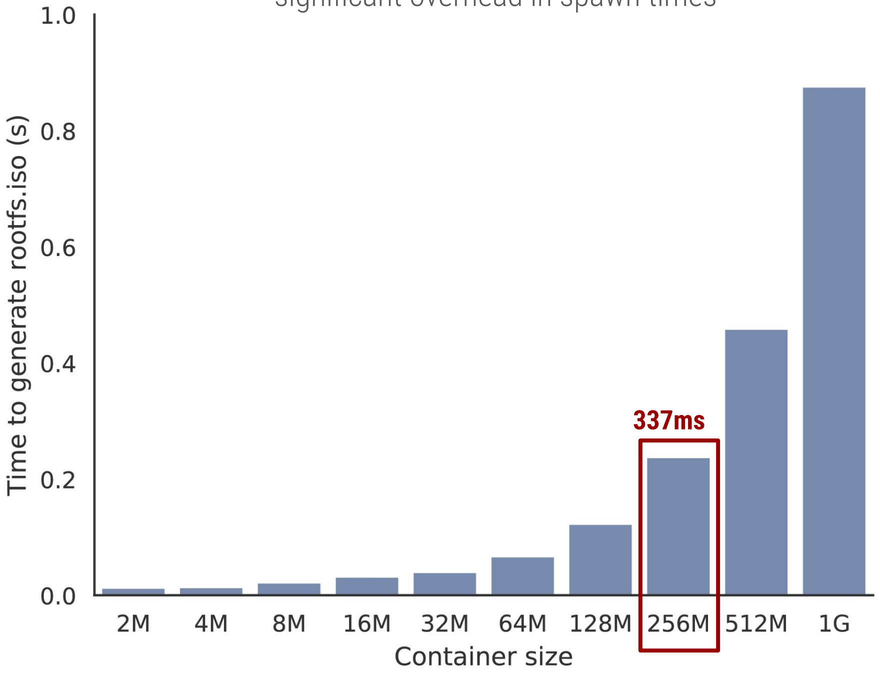
Example: Nabla container (libs + python unikernel) ⟶ function: simple HTTP request

15% of the total request execution time (cold spawn to tear-down!)

We eliminate this overhead from the critical path of the function execution:

⟶ **Faster function instantiation**

Serverless functions are short-lived ⟶ Rootfs generation ⟶ significant overhead in spawn times

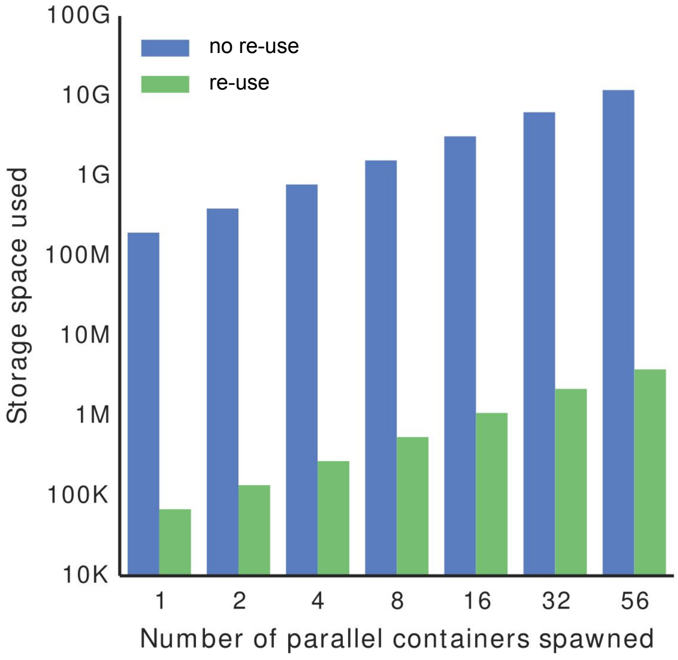


Evaluation :: Increase host intensity

Inject precooked image files (rootfs-X.iso) in each layer at container build time.
⟶ 100% reuse of the layers and the unikernel.

Generic Nabla : 56 containers (system limit)
⟶ over 10GBs

Our approach : Less than 3MB of extra disk space
⟶ storage reuse increases host intensity limit



**IBM cloud hosted Xeon(R) Gold 5120 CPU @ 2.20GHz*

Conclusion

We introduce a mechanism to:

- **Enable docker layers approach**
- **Enable container image layers re-use →increase intensity on host**

Our results show that:

→ **Storage space overhead per container is eliminated**

→ Overhead of image generation at runtime is eliminated, **enabling instant cold boot times**

Thank you

Questions?