SOCK: Rapid Task Provisioning with Serverless-Optimized Containers

Edward Oakes, Leon Yang, Dennis Zhou, Kevin Houck, Tyler Harter*, Andrea C. Arpaci-Dusseau, Remzi H. Arpaci-Dusseau

*Microsoft Gray Systems Lab
Increasing Developer Velocity
Trend 1: Rise of High-Level Languages

Growth of major programming languages
Based on Stack Overflow question views in World Bank high-income countries

https://stackoverflow.blog/2017/09/06/incredible-growth-python/
Trend 2: Greater Reliance on Packages

https://www.modulecounts.com
Trend 3: Microservice Decomposition

- Applications are decoupled into modular “services”
- Each service is lightweight, deployed independently
Serverless Computing

- “Functions as a Service”
- Pay-as-you-go, fine-grained billing
Serverless Computing

Benefits:

● True auto scaling
● Massive parallelism
● Cost savings
Serverless Computing

Benefits:
- True auto scaling
- Massive parallelism
- Cost savings

Challenge:
- Deploy, isolate, and start in milliseconds
Serverless Runtime

A_1 \quad A_2 \quad \ldots \quad A_N

Server

Operating System

Hardware

deployment bundles
Serverless Runtime

Docker container:
- 400ms

Python interpreter:
- 30ms

deployment bundles
Serverless Runtime

Docker container:
- 400ms

Python interpreter:
- 30ms

deployment bundles
Serverless Runtime

Docker container:
- 400ms

Python interpreter:
- 30ms

scipy:
- 2700ms download
- 8200ms install
- 88ms import

Server

Operating System

Hardware
Lean serverless-optimized containers (SOCK)

- Precise usage of Linux isolation mechanisms
- 18x faster container lifecycle over Docker
Lean serverless-optimized containers (SOCK)

- Precise usage of Linux isolation mechanisms
- **18x** faster container lifecycle over Docker

Provision from secure Zygote processes

- Fork from initialized runtime to prevent cold start
- **3x** faster provisioning than SOCK alone
Lean serverless-optimized containers (SOCK)

- Precise usage of Linux isolation mechanisms
- \textbf{18x} faster container lifecycle over Docker

Provision from secure Zygote processes

- Fork from initialized runtime to prevent cold start
- \textbf{3x} faster provisioning than SOCK alone

Execution caching across 3 tiers

- Securely reuse initialization work across customers
- \textbf{3-16x} lower platform cost in image-processing case study
Outline

Motivation

Serverless-optimized Containers
- Design
- Evaluation

Generalized Zygotes
- Design
- Evaluation

Serverless Caching
- Design
- Evaluation

Conclusion
Linux Containers
Linux Containers

...they’re just cheaper VMs, right?
Linux Containers

...they’re just cheaper VMs, right?

Not virtualizing hardware, but access

- File system
- Namespaces
- Cgroups
Linux Containers

...they’re just cheaper VMs, right?

Not virtualizing hardware, but access

- File system
- Namespaces
- Cgroups
Container File System

FROM ubuntu:16

- Orange = read only
- Green = read/write
Container File System

FROM ubuntu:16

sudo apt-get install

= read only

= read/write

FROM ubuntu:16
Container File System

- `sudo apt-get install` = read/write
- `= read only`
- `= read/write`

- `make install` (green)
- `sudo apt-get install` (orange)
- `FROM ubuntu:16` (orange)
Container File System

FROM ubuntu:16

sudo apt-get install

make install

sudo apt-get install

FROM ubuntu:16

= read only

= read/write
Container File System

C1 “/”

C2 “/”

C3 “/”

my-image

= read only

= read/write
Linux Containers

...they’re just cheaper VMs, right?

Not virtualizing hardware, but access

- File system
- Namespaces
- Cgroups
Namespaces

- Partition resource access in the kernel
- 7 individual namespaces
  - Mount
  - Network
  - User
  - UTS
  - IPC
  - PID
  - Cgroup
Mount Namespace

```
/ 
/var 
/tmp 
/my-image 
/my-var 
/my-tmp
```
Mount Namespace

P1 → fork() → P2

/  
  /var  
  /tmp  
  /my-image  
  /my-var  
  /my-tmp
Mount Namespace

P1

P2

/
/var
/tmp
/my-image
/my-var
/my-tmp
Mount Namespace

/  
  /var
  /my-image
  /my-var
  /my-tmp

P1

P2

unshare()
Mount Namespace

P1

/  
  /var  
  /tmp  
  /my-image  
  /my-var  
  /my-tmp

P2

/  
  /var  
  /tmp  
  /my-image  
  /my-var  
  /my-tmp

unshare()
Mount Namespace

P1

/ 
/var
/tmp
/my-image
/my-var
/my-tmp

P2

/ 
/var
/tmp
/my-image
/my-var
/my-tmp
Mount Namespace

P1

/ 
/var 
/tmp 
/my-image 
/my-var 
/my-tmp

P2

/ 
/var 
/tmp 
/my-image 
/my-var 
/my-tmp

switch root
- unmount()
- mount()
- pivot_root()
Mount Namespace

P1

/ 
/var 
/tmp 
/my-image 
/my-var 
/my-tmp

P2

/ 
/my-var 
/my-tmp

switch root
- unmount()
- mount()
- pivot_root()
Linux Containers

...they’re just cheaper VMs, right?

Not virtualizing hardware, but access

- File system
- Namespaces
- Cgroups
Cgroups

- Control resource usage
- Limiting, prioritization, accounting, control
  - oom-killer for a container
At runtime:

- Fork init, unshare() into new namespaces
- Create cgroups
- Relocate init into cgroups
- Stitch together root file system
- switch_root() to container root
- Create veth
- Connect veth to virtual bridge
At runtime:

- Fork init, unshare() into new namespaces
- Create cgroups
- Relocate init into cgroups
- Stitch together root file system
- switch_root() to container root
- Create veth
- Connect veth to virtual bridge

...all before running any user code
SOCK: Serverless-optimized Containers

- Containers aren’t a single cohesive abstraction
SOCK: Serverless-optimized Containers

- Containers aren’t a single cohesive abstraction

What are the performance costs of container components?
SOCK: Serverless-optimized Containers

- Containers aren’t a single cohesive abstraction

  What are the performance costs of container components?

  What are the isolation requirements of serverless workloads?
SOCK: Serverless-optimized Containers

- Containers aren’t a single cohesive abstraction

What are the performance costs of container components?

What are the isolation requirements of serverless workloads?
Mount Performance

Mount and unmount as quickly as possible

- Varying levels of parallelism
- Single AUFS layer vs. bind mount
Mount Performance

![Graph showing Mount Performance with lines for Bind and AUFS, depicting Ops/Second against Concurrent Ops.]

- Bind
- AUFS
Mount Performance

Bind mounts are 3x faster than AUFS
SOCK: Serverless-optimized Containers

- Containers aren’t a single cohesive abstraction

What are the performance costs of container components?

What are the isolation requirements of serverless workloads?
File System Requirements

Serverless application containers:

- Don’t need a fully writable OS view
- Do need scratch space and access to libraries
File System Requirements

Serverless application containers:

- Don’t need a fully writable OS view
- Do need scratch space and access to libraries

 Flexible, expensive AUFS + mount namespace

 Simple, cheap bind mounts + chroot
Serverless-optimized Containers

Replace flexible, costly mechanisms with simple, cheap alternatives

- Leverage constraints of the serverless runtime
Serverless-optimized Containers

Replace flexible, costly mechanisms with simple, cheap alternatives

- Leverage constraints of the serverless runtime

  AUFS + mount NS -> bind mounts + chroot

  network NS -> domain socket + outbound access

  user NS -> unprivileged execution
Outline

Motivation

Serverless-optimized Containers
- Design
- Evaluation

Generalized Zygotes
- Design
- Evaluation

Serverless Caching
- Design
- Evaluation

Conclusion
Experiment

Requests to “no-op” handlers as quickly as possible

- Varying numbers of requesting threads
- Docker vs. SOCK
SOCK Container Performance

![Graph showing SOCK Container Performance](image)

- **Latency (ms)** on the y-axis.
- **Throughput (ops/s)** on the x-axis.
- Data points for 1 concurrent, 10 concurrent, and 20 concurrent are marked.
- The graph compares Docker and SOCK container performance.

---

**Legend**:  
- ■ 1 concurrent  
- ● 10 concurrent  
- ♦ 20 concurrent
SOCK Container Performance

18x faster container lifecycle with SOCK
Outline

Motivation

Serverless-optimized Containers
  ● Design
  ● Evaluation

Generalized Zygotes
  ● Design
  ● Evaluation

Serverless Caching
  ● Design
  ● Evaluation

Conclusion
Zygotes

- Used in Android OS
  - Many apps depend on common system libraries
- Start a Zygote at init, importing libraries
  - New processes fork from the Zygote
Generalized Zygotes

Benefits:

- Eliminate interpreter & package initialization cost
- Pack more handlers into memory
Generalized Zygotes

Benefits:
- Eliminate interpreter & package initialization cost
- Pack more handlers into memory

Challenges:
- Cannot trust the libraries we import
- Want to create new Zygotes on the fly
More details in the paper...
Outline

Motivation

Serverless-optimized Containers
- Design
- Evaluation

Generalized Zygotes
- Design
- Evaluation

Serverless Caching
- Design
- Evaluation

Conclusion
Experiment

Create and destroy handler runtimes as quickly as possible

- New container & interpreter
- Varying levels of parallelism
Zygote Provisioning Performance

3x faster provisioning using general Zygotes
Outline

Motivation

Serverless-optimized Containers
- Design
- Evaluation

Generalized Zygotes
- Design
- Evaluation

Serverless Caching
- Design
- Evaluation

Conclusion
SOCK Caching
SOCK Caching

Handler Cache

- Reuse initialized runtimes within a lambda
SOCK Caching

Handler Cache
  ● Reuse initialized runtimes *within* a lambda

Import Cache
  ● Reuse initialized Zygotes *between* lambdas
SOCK Caching

Handler Cache
- Reuse initialized runtimes \textit{within} a lambda

Import Cache
- Reuse initialized Zygotes \textit{between} lambdas

Install Cache
- Reuse installed packages \textit{between} lambdas
SOCK Caching

Handler Cache
● Reuse initialized runtimes *within* a lambda

Import Cache
● Reuse initialized Zygotes *between* lambdas

Install Cache
● Reuse installed packages *between* lambdas
$H_1(A)$

Import Cache

{}
Import Cache

Handler Cache

$H_1(A)$

{A}

{}
Import Cache

{A} \rightarrow \{ \}

Handler Cache

H_1(A)
Import Cache

\[ H_2(A,B) \]

Handler Cache

\[ H_1(A) \]
Import Cache

$H_2(A, B)$

{A}

{} →

{A, B}

Handler Cache

$H_1(A)$
Import Cache

\[ H_2(A,B) \]

{A}

{A,B}

Handler Cache

\[ H_1(A) \]

\[ H_2(A,B) \]
Import Cache

{A}

{A,B}

Handler Cache

H₁(A)

H₂(A,B)
$$H_3(B)$$

**Import Cache**

- $\{A\}$
- $\{A, B\}$

**Handler Cache**

- $H_1(A)$
- $H_2(A, B)$
Import Cache

\[ H_3(B) \]

\{A\} \rightarrow \{\}

\{A,B\}

Handler Cache

\[ H_1(A) \]

\[ H_2(A,B) \]
What if package ‘A’ is malicious?
What if package ‘A’ is malicious?

- “Subset only” rule
What if package ‘A’ is malicious?

- “Subset only” rule
Cache Interaction

handler cache

working set

- blue = django memory
- white = handler-specific memory
Cache Interaction

handler cache

working set

django Zygote

= django memory

= handler-specific memory
Cache Interaction

- handler cache
- working set
- django Zygote

Legend:
- Blue = django memory
- White = handler-specific memory
Cache Interaction

Handler cache misses are:
- Rarer

- = django memory
- = handler-specific memory

working set

django Zygote
Cache Interaction

Handler cache misses are:
- Rarer
- Faster

- = django memory
- = handler-specific memory

Handler cache misses are:
- Rarer
- Faster
Outline

Motivation

Serverless-optimized Containers
  ● Design
  ● Evaluation

Generalized Zygotes
  ● Design
  ● Evaluation

Serverless Caching
  ● Design
  ● Evaluation

Conclusion
Microbenchmark

Not a stress test, want to examine differences in caching

Experimental Setup:

- 1 OpenLambda worker machine
- 2 random requests per second
- 100 distinct lambdas, all importing django
Caching Performance

- **both**
- **import**
- **handler**
- **no cache**

Latency (ms) vs. Percent of Reqs
Caching Performance

- **both**
- **import**
- **handler**
- **no cache**
Caching Performance

![Graph showing caching performance with latency on the x-axis and percent of requests on the y-axis. The graph compares different caching strategies: both, import, handler, and no cache.](image-url)
Outline

Motivation

Serverless-optimized Containers
  ● Design
  ● Evaluation

Generalized Zygotes
  ● Design
  ● Evaluation

Serverless Caching
  ● Design
  ● Evaluation

Conclusion
Evolution of Applications

- PC running many diverse processes
- VMs running monolithic applications
- Containers running small pieces of applications
Evolution of Applications

PC running many diverse processes

VMs running monolithic applications

Containers running small pieces of applications

???
Modern Virtualization

How can we run small, distributed pieces of code faster, more easily, and more securely?
Modern Virtualization

How can we run small, distributed pieces of code faster, more easily, and more securely?

SOCK:
- Carefully measure and use existing abstractions
Modern Virtualization

How can we run small, distributed pieces of code faster, more easily, and more securely?

SOCK:
- Carefully measure and use existing abstractions developed for long-running applications

Future Systems:
- Need to fundamentally rethink design
Questions?