IPADS
INSTITUTE OF PARALLEL
AND DISTRIBUTED SYSTEMS

上海交通大學
SHANGHAI JIAO TONG UNIVERSITY

上海人工智能实验室
Shanghai Artificial Intelligence Laboratory

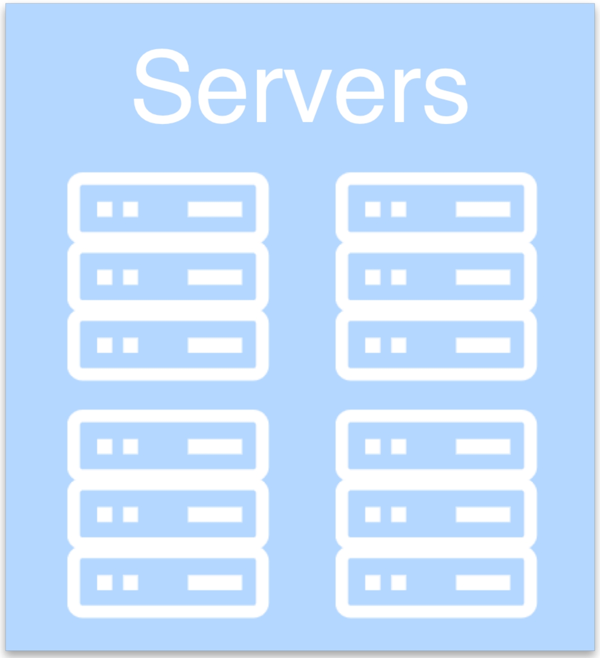
No Provisioned Concurrency: Fast RDMA-codesigned Remote Fork for Serverless Computing

Xingda Wei, Fangming Lu, Tianxia Wang,
Jinyu Gu, Yuhan Yang, Rong Chen, and Haibo Chen

Serverless computing: a new paradigm

Developer's view

w/o serverless

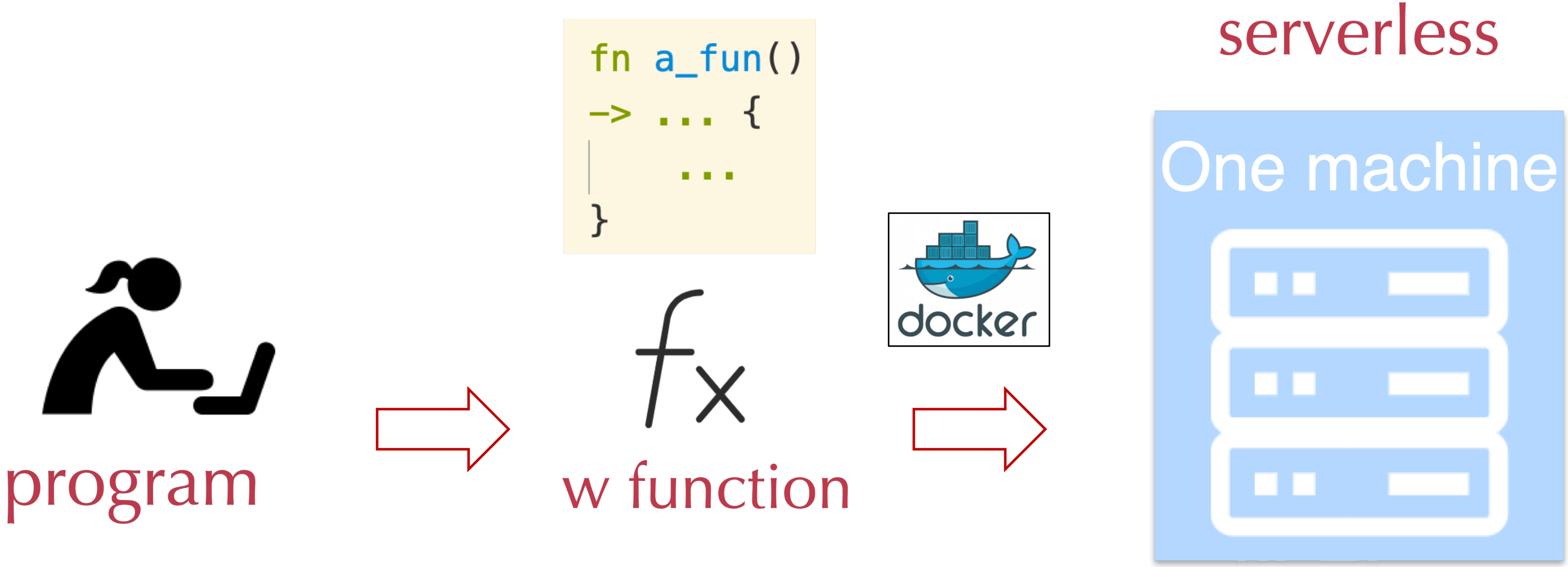


serverless

How to abstract the user code? **Function (FaaS)**

The developers package their code into **function**

- ❑ Functions are encapsulated into **containers**



```
fn a_fun()
-> ... {
      ...
}
```

serverless

One machine

program

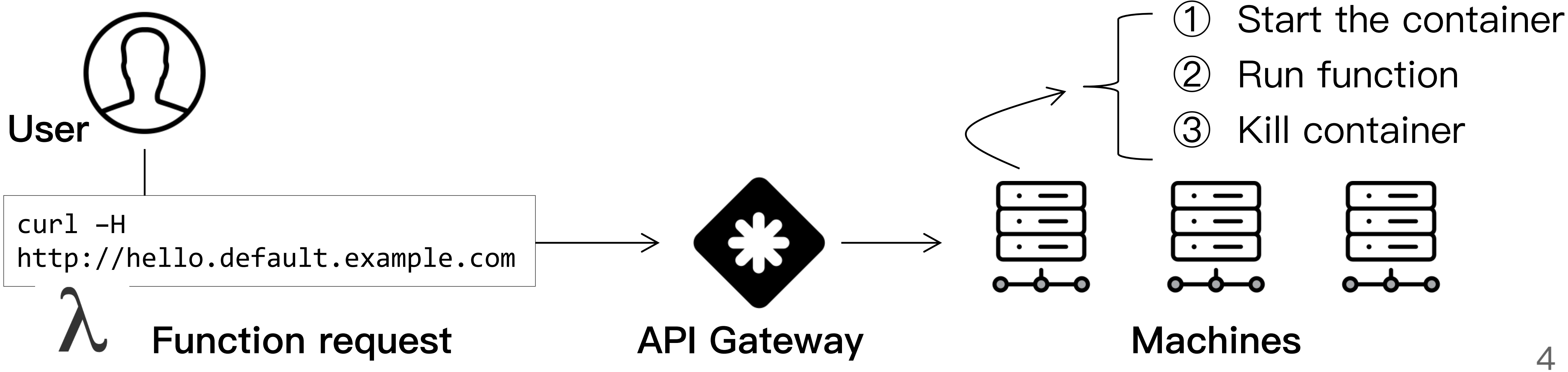
w function

Functions are executed via auto-scaled **containers**

Containers are spawned **on-demand** based on requests

- ❑ The platform will assign a unique URL for each function for the request naming

Name	Function URL
`hello`	`http://hello.default.example.com`



① Start the container
② Run function
③ Kill container

User

```
curl -H
http://hello.default.example.com
```

Function request

API Gateway

Machines

The benefits

Ease of development

- No need to worry how to deploy applications on servers

High resource utilizations

- Containers only run when there is workloads

Economical efficiency

- Less paid by the user, more resource used by the platform

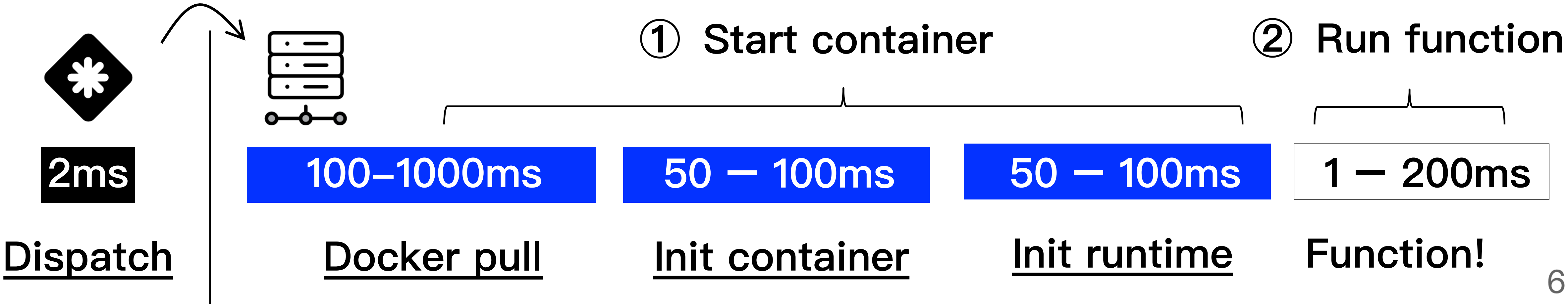
The benefits **at what cost? Function coldstart**

Function invocation requires **booting a container from scratch**

- Contains multiple steps to prepare the function executing environment

Unfortunately, serverless functions are **ephemeral**

- E.g., 67% of the functions execute within 20ms [Lambda@Edge]



① Start container

② Run function

2ms
Dispatch

100–1000ms
Docker pull

50 — 100ms
Init container

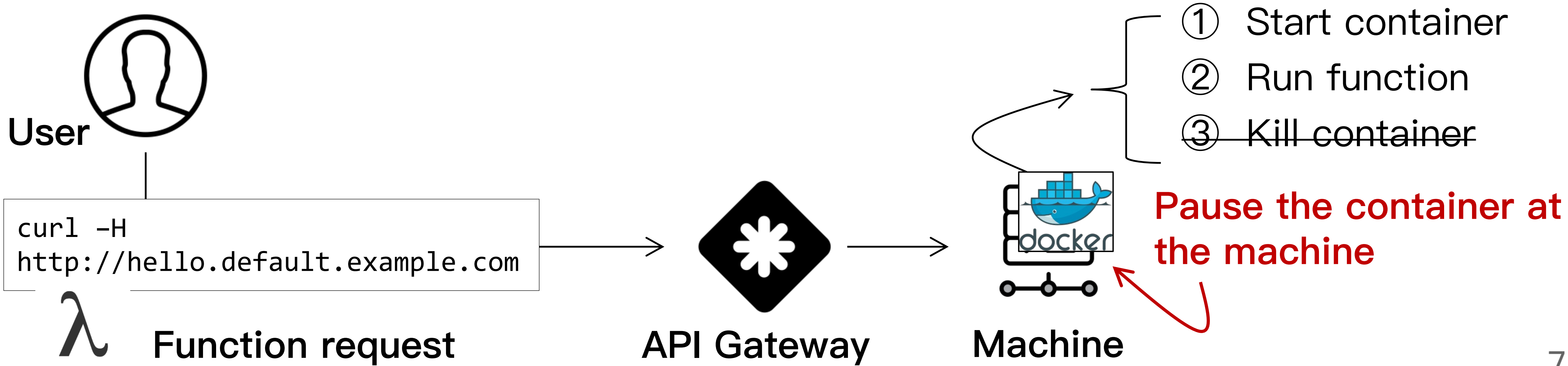
50 — 100ms
Init runtime

1 — 200ms
Function!

Solution#1: **Cache** runned containers (**warmstart**)

After running the functions, don't kill the runned containers

- Cache it in the machine's memory for future usage
- E.g., via **`docker pause`**



```
curl -H
http://hello.default.example.com
```

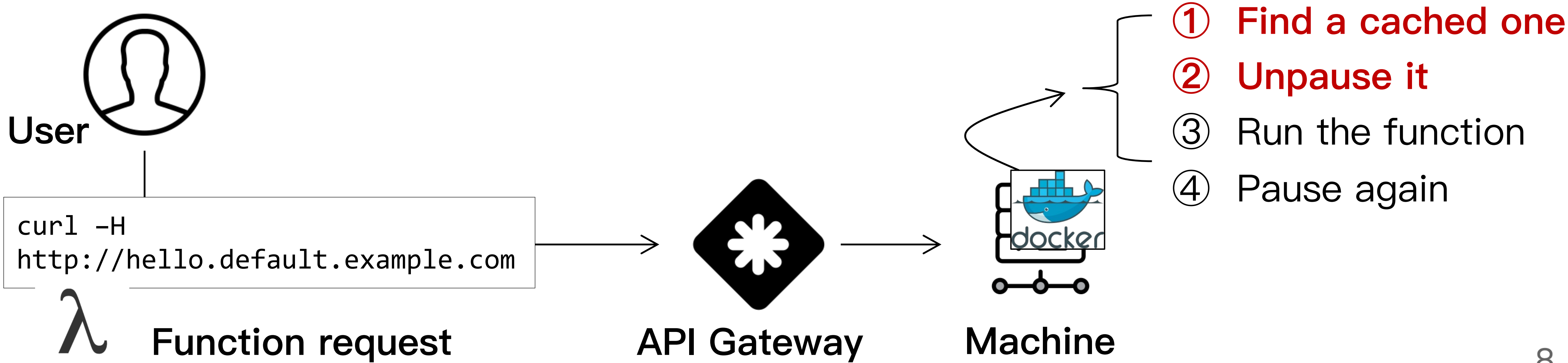
① Start container
② Run function
~~③ Kill container~~

Pause the container at the machine

Solution#1: **Cache** runned containers (**warmstart**)

Future invocations can reuse cached instance

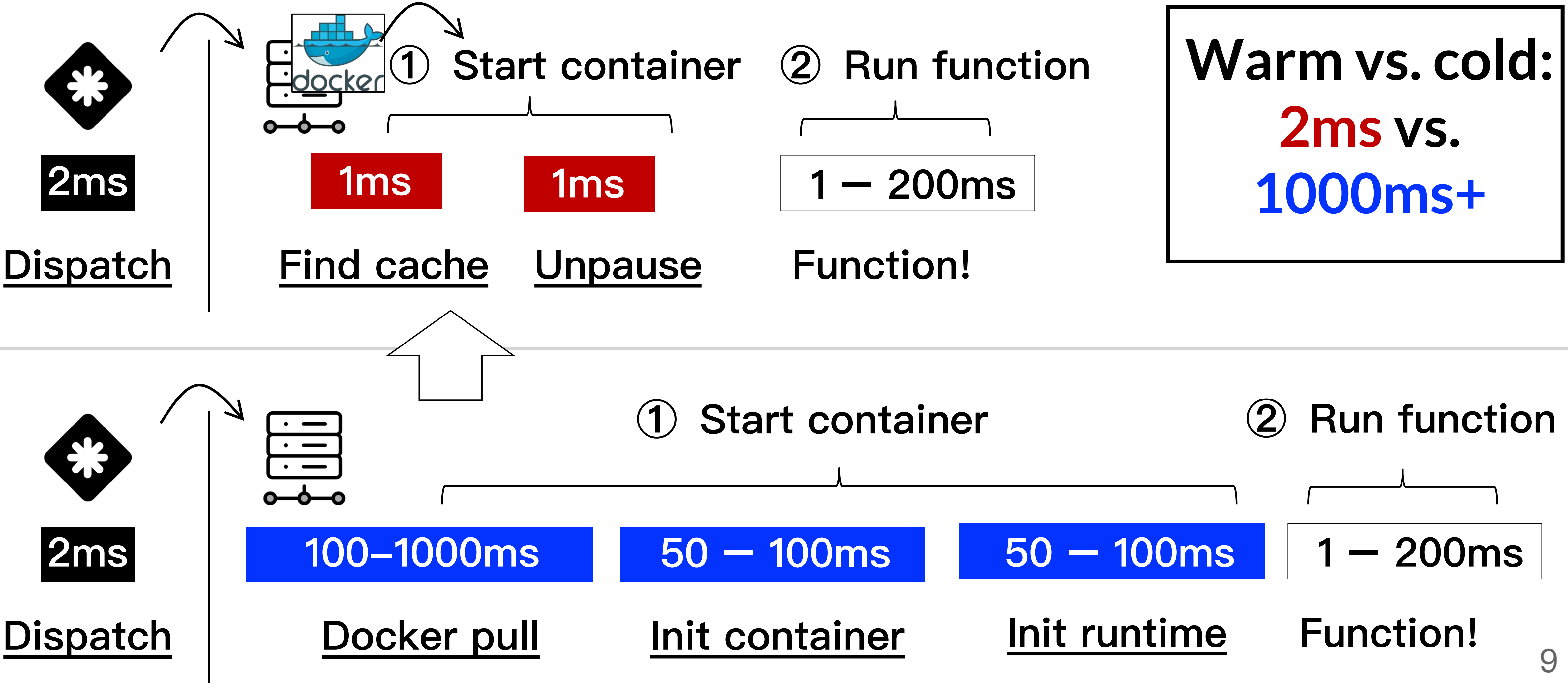
- If a container for the function is cached, then start function from it
- E.g., via **`docker unpause`**



```
curl -H
http://hello.default.example.com
```

① **Find a cached one**
② **Unpause it**
③ Run the function
④ Pause again

Warmstart is fast (near **optimal in performance**)



Warm vs. cold:
2ms vs.
1000ms+

① Start container ② Run function

Dispatch	Find cache	Unpause	Function!
2ms	1ms	1ms	1 — 200ms

① Start container ② Run function

Dispatch	Docker pull	Init container	Init runtime	Function!
2ms	100–1000ms	50 — 100ms	50 — 100ms	1 — 200ms

Challenge of caching: **provisioned concurrency**

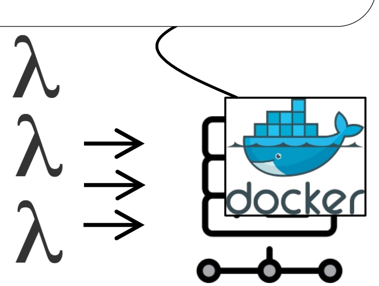
Need cache sufficient (**O(n)**) containers beforehand

- One cached container **can only be unpaused for one invocation**

Meanwhile, real workload exists **concurrent function invocations**

- E.g., loadspikes may appear in real workloads [Serverless in the wild@ATC'20]

What if the one has been used? Fallback to cold



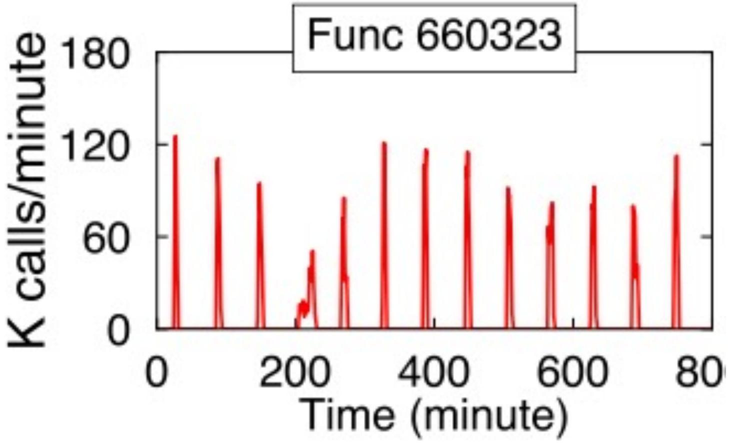
① **Find a cached one**

② Unpause it

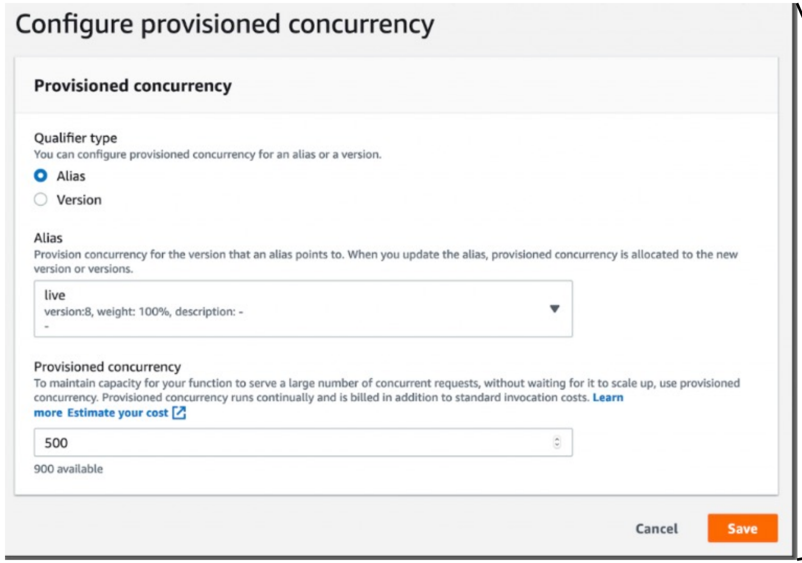
③ Run the function

④ Pause again

Real-world function traces from Azure function.
Source: Serverless in the wild@ATC'20



Goal: warmstart + **no provisioned concurrency**



Configure provisioned concurrency

Provisioned concurrency

Qualifier type
You can configure provisioned concurrency for an alias or a version.

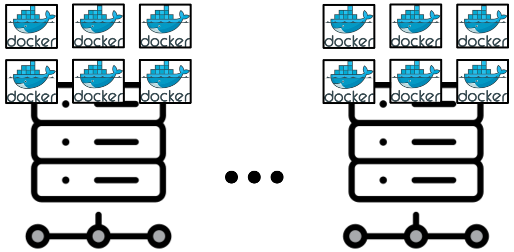
● Alias
○ Version

Alias
Provision concurrency for the version that an alias points to. When you update the alias, provisioned concurrency is allocated to the new version or versions.

live
version:8, weight: 100%, description: -
-

Provisioned concurrency
To maintain capacity for your function to serve a large number of concurrent requests, without waiting for it to scale up, use provisioned concurrency. Provisioned concurrency runs continually and is billed in addition to standard invocation costs. **Learn more** **Estimate your cost**

500
900 available



How many containers to cache?

Existing platforms **require users** to specific the number of cached instances (provisioned concurrency) to improve performance.

Source: https://aws.amazon.com/cn/blogs/aws/new-provisioned-concurrency-for-lambda-functions/

Insight: No provisioned concurrency

- ❑ Users only need to tell the platform **whether** it needs **to prevent coldstart**
- ❑ **Not** tell **how many,** e.g., we only need **O(1)** resource provisioned all the time

Solution#2: Caching **+ Fork**

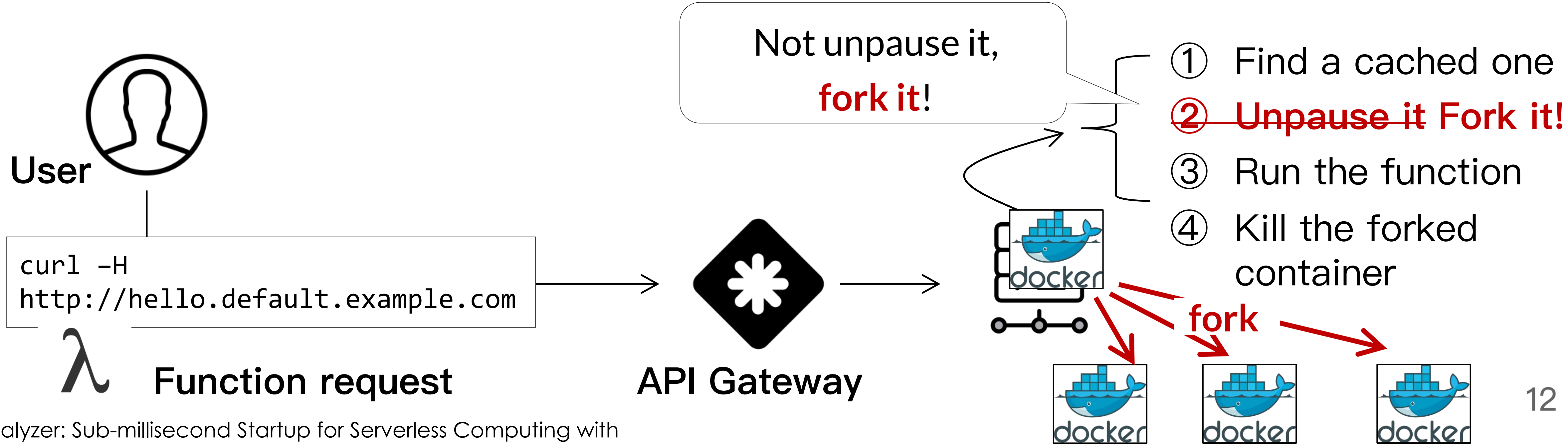
FORK

NAME

fork – create a new process

Use **OS Fork** to start new containers from one cached containers [1]

- One container (parent) can be forking many times
- Reduce the provisioned concurrency from **O(n) -> O(1) on a single machine**
- Still reduce many steps in coldstart



Not unpause it,
fork it!

① Find a cached one
~~② Unpause it~~ **Fork it!**
③ Run the function
④ Kill the forked container

```
curl -H
http://hello.default.example.com
```

---

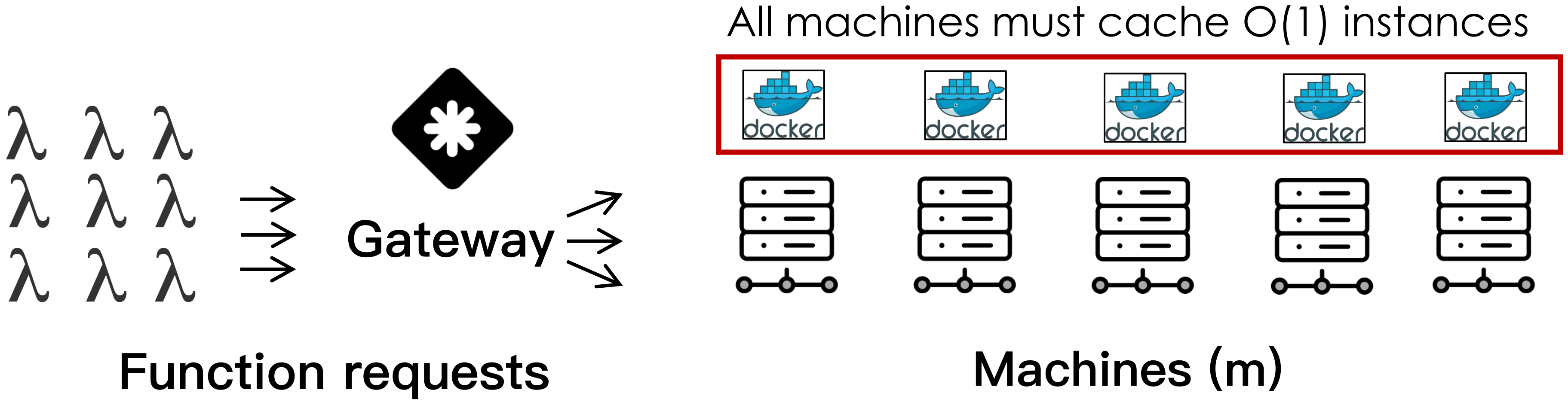
[1] Catalyzer: Sub-millisecond Startup for Serverless Computing with Initialization-less Booting, ASPLOS'20

Limitation of Fork: **cannot scale out**

Question: what if functions need to run across machines?

- ❑ Fork can only achieve O(1) resource provisioned on a single machine
- ❑ Still requires **O(m)** resource provisioned considering scale out!

m: number of machines to run functions

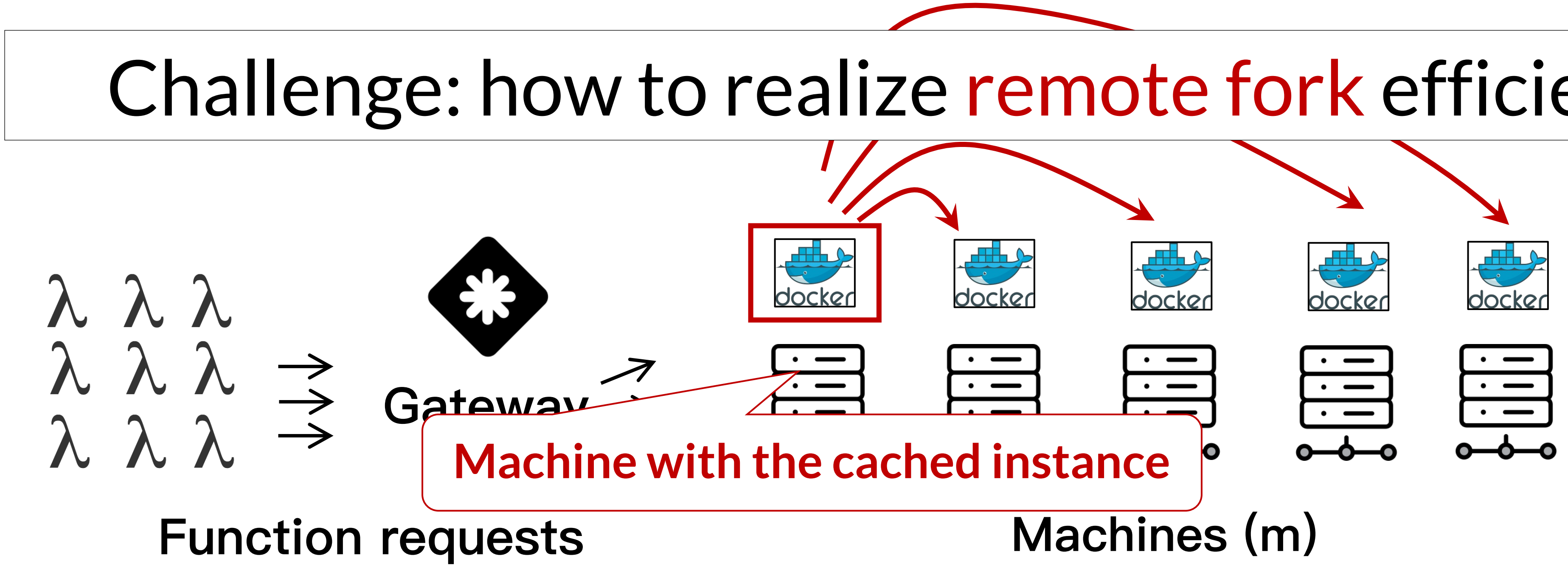


This work: Fast **remote fork** for serverless computing

Observation: remote fork achieves true “no provisioned concurrency”

- One cached container (**O(1)**) can fork multiple instances even across machines

Challenge: how to realize remote fork efficiently?



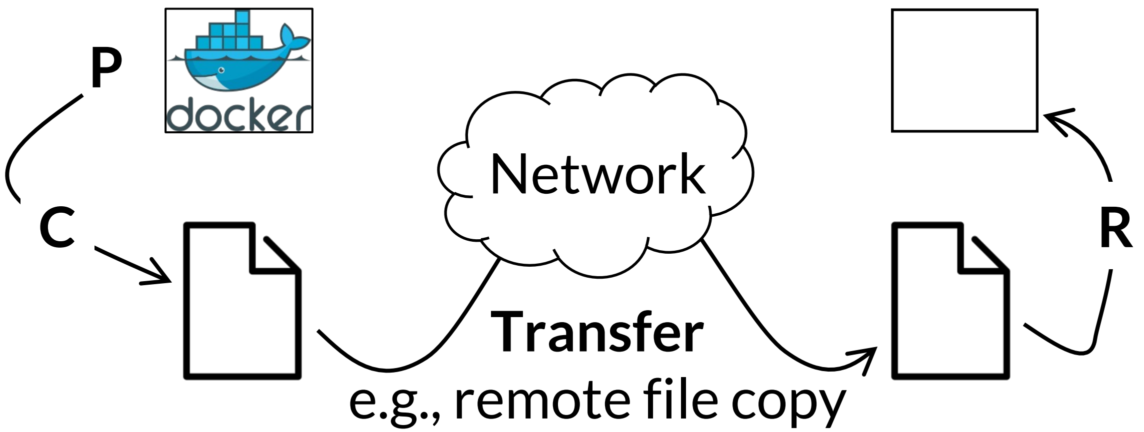
Existing remote fork: Checkpoint & Restore (C/R)

Checkpoint (C)

- Checkpoint the parent (P) states to a file

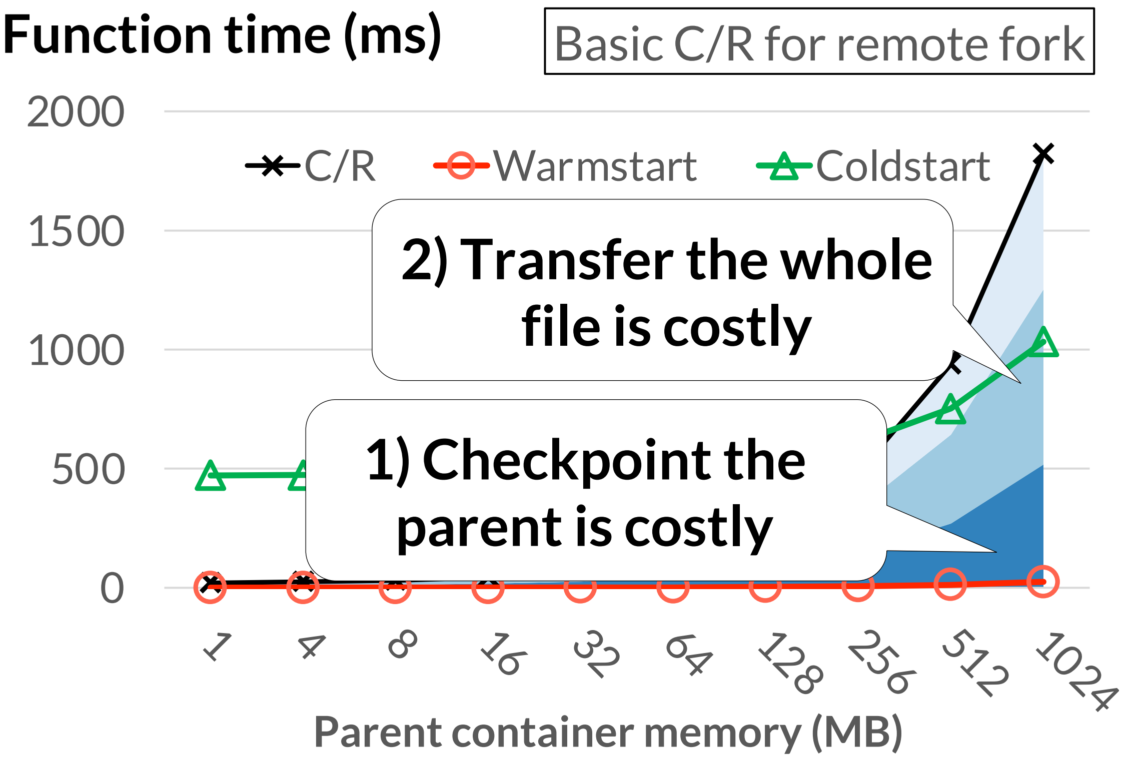
Restore (R)

- Transfer the file to the child machine
- Restore the parent from the file



Transfer

e.g., remote file copy



Basic C/R for remote fork

2) Transfer the whole file is costly

1) Checkpoint the parent is costly

Evaluation setup: CRIU for C/R, file is transferred via RDMA and is stored in-memory

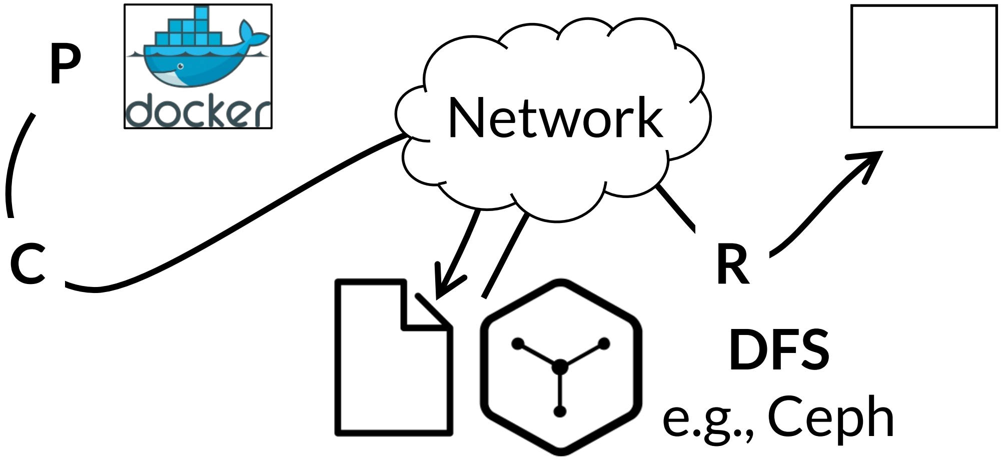
Optimization: Using C/R + distributed file system (DFS)

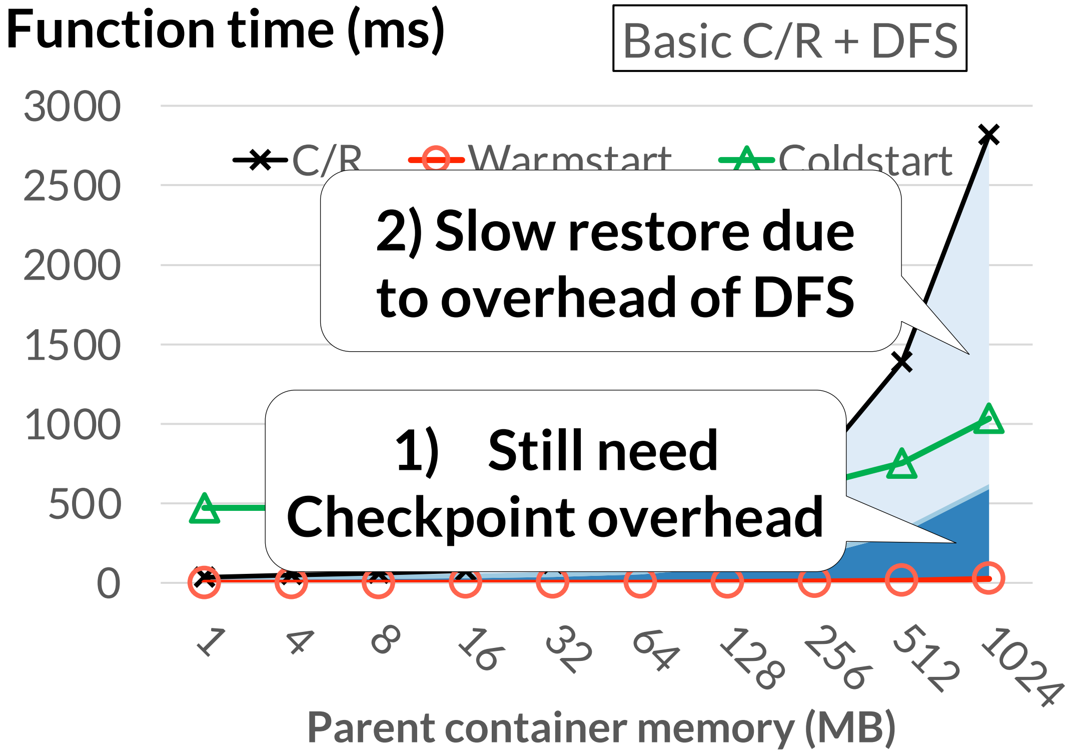
Checkpoint (C)

- Checkpoint the parent (P) states to a file

Restore (R)

- ~~Transfer the file to the child machine~~
- Restore the parent from the file





2) Slow restore due to overhead of DFS

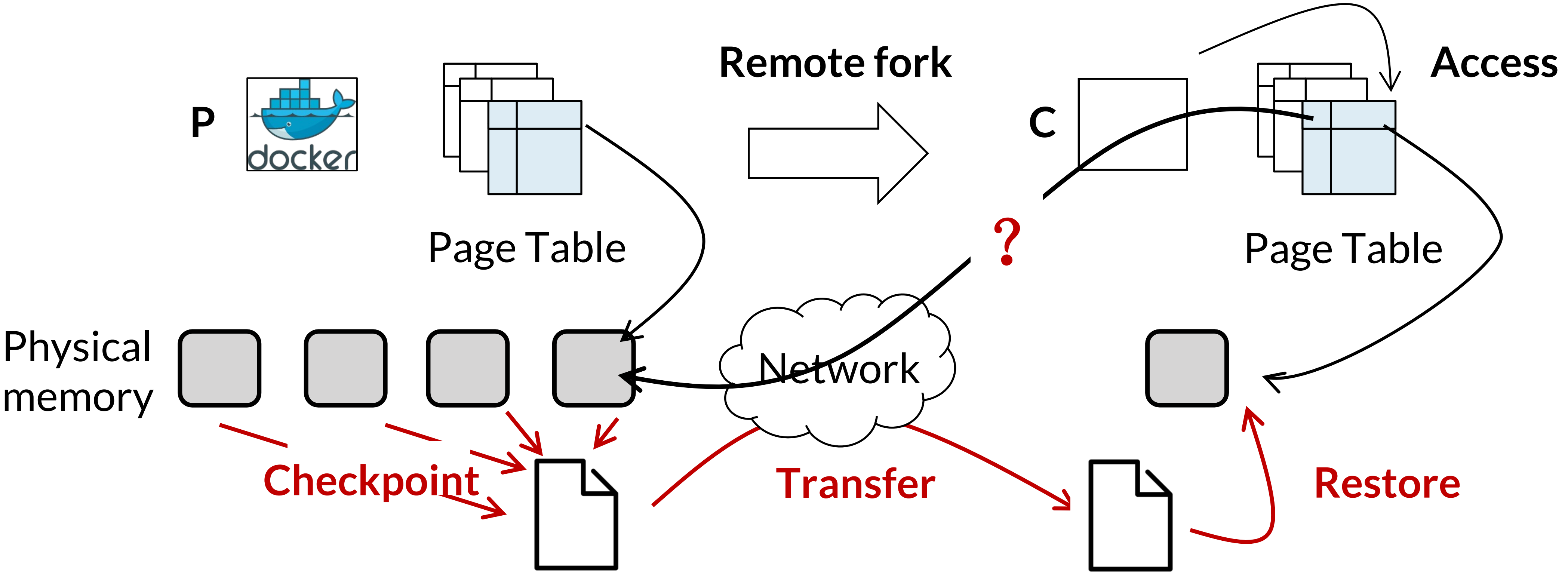
1) Still need Checkpoint overhead

Evaluation setup: CRIU for C/R, file is stored in Ceph, a state-of-the-art RDMA-enabled DFS, file is in-memory

Key problem: OS cannot access remote memory

Thus, the child needs the filesystem to access the remote memory

- ❑ Filesystem essentially pays the overhead of checkpoint & file accesses



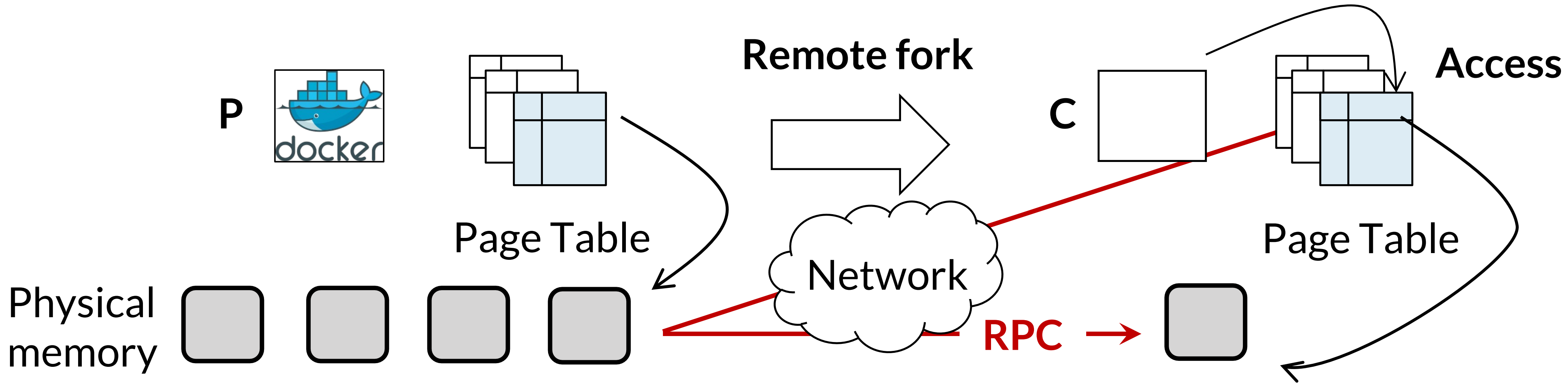
This work: **OS remote memory** for fast remote fork

The OS provides **remote memory abstraction** for remote fork

- ❑ E.g., directly use kernel-space RPC to fetch the memory bypassing the filesystem

Benefits

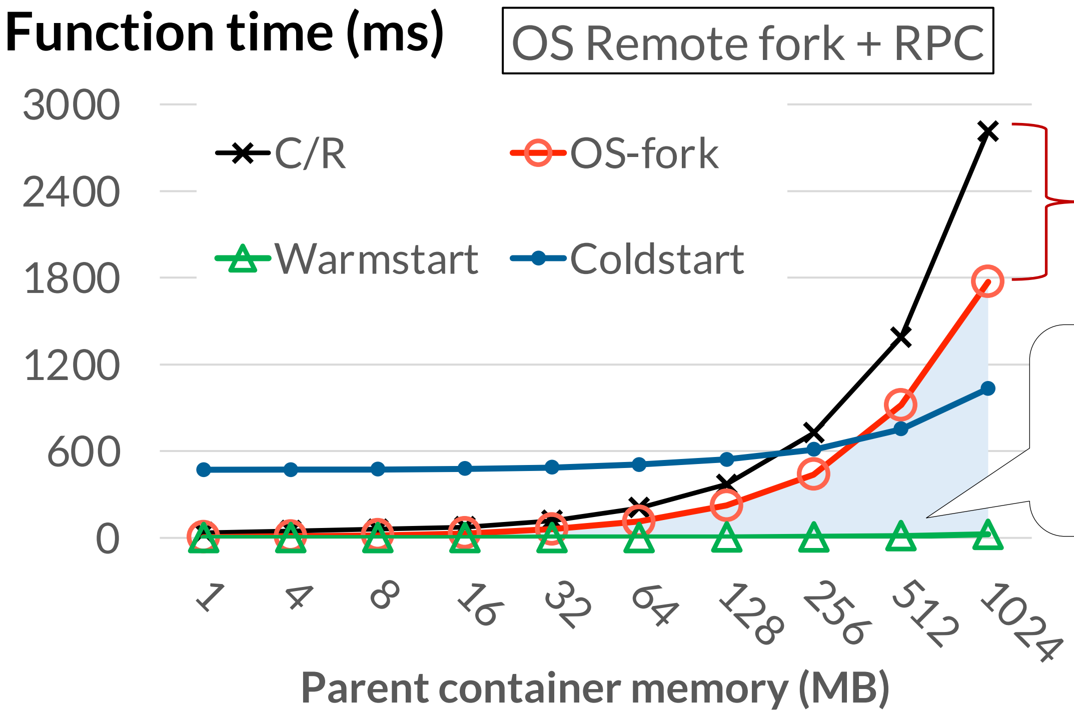
- ❑ Avoid the costly checkpoint phase
- ❑ Avoid the filesystem overhead in reading the remote data



This work: **OS remote memory** for fast remote fork

Extend the OS to directly access the memory of remote machines

- Then implement remote fork by **imitating local fork w/ remote memory accesses**



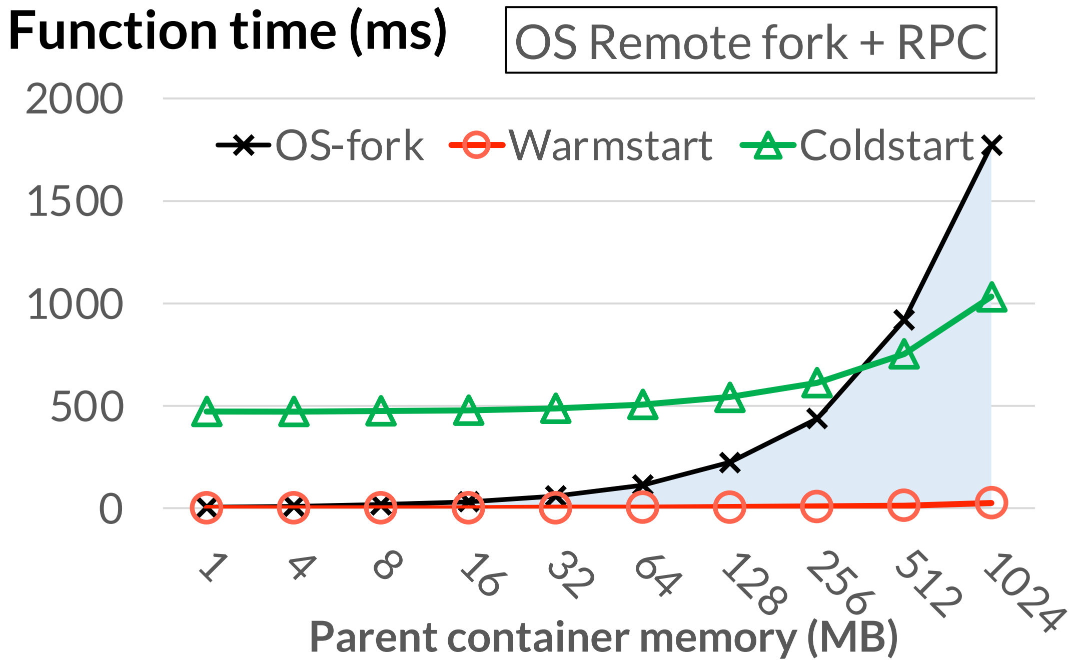
37% improvements

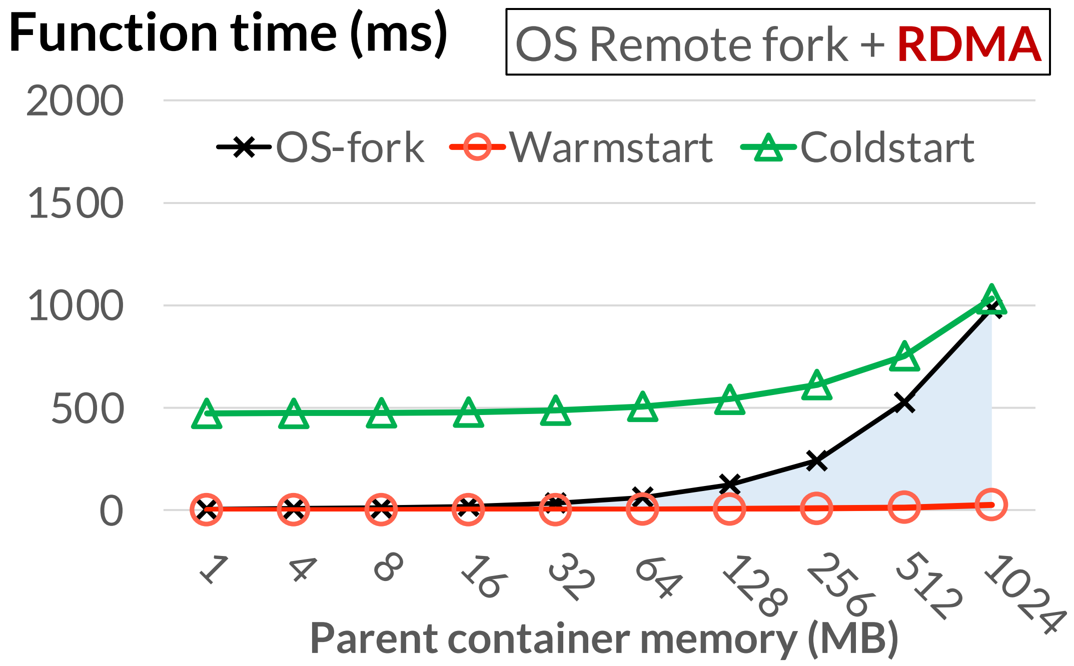
Avoid checkpoint, but still pays high overhead when using RPC to read the memory

Further accelerate OS remote memory: **RDMA**!

Observation: modern datacenter interconnects RDMA

- RDMA provides high bandwidth (400Gbps) & low latency (2μs) remote memory access
- The OS can **directly access the physical memory** of the others **via RDMA** [1]

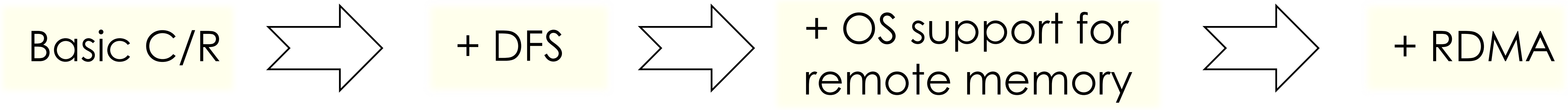




---

[1] LITE Kernel RDMA Support for Datacenter Applications, SOSP'17

Roadmap to **the fast remote fork**

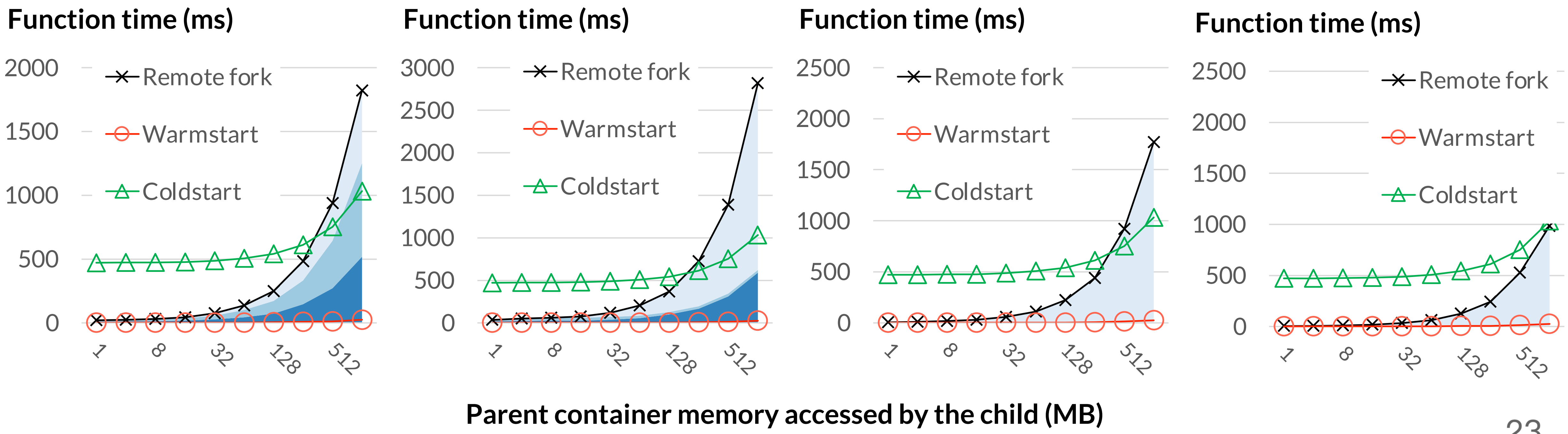


Basic C/R

+ DFS

+ OS support for remote memory

+ RDMA



MITOSIS fast RDMA-OS codesigned remote fork

OS support remote fork + RDMA for accelerating data accesses

- OS fork -> reduce overhead to checkpoint the memory into files
- RDMA -> optimal performance in accessing child memory

Compatible w/
containers
e.g., **runC**

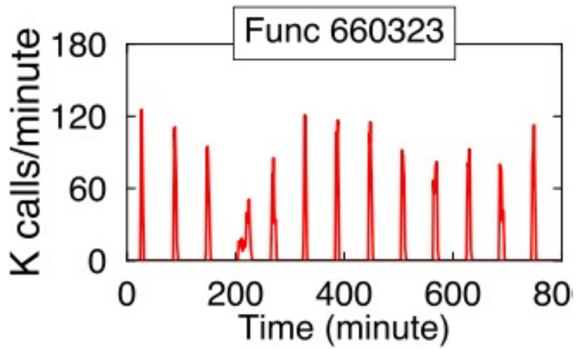
Forking **10,000+** containers
within one second

Accelerate serverless coldstart

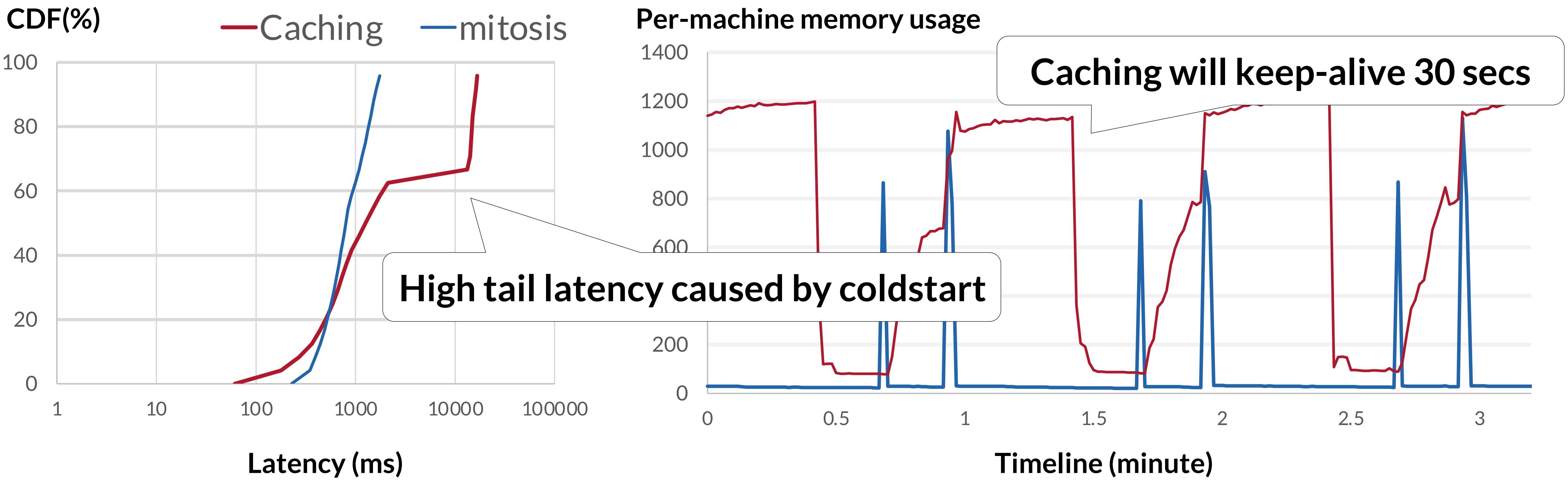
Reduce tail latency by 90% under loads pikes

Efficiency under **loads pikes**

Evaluate platform: Fn



Requests generated from real-world traces[1]



High tail latency caused by coldstart

Caching will keep-alive 30 secs

[1] Serverless in the Wild: Characterizing and Optimizing the Serverless Workload at a Large Cloud Provider@ATC'20

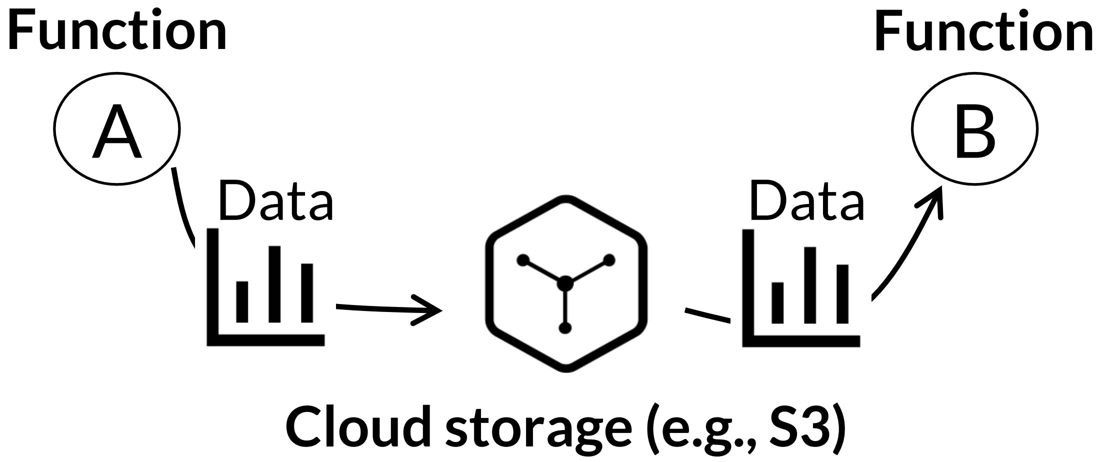
One more thing: fork for **serialization-free** state transfer

Serverless workflow can compose multiple functions together

- ❑ While functions use **message passing (MP)** or **cloud storage (CS)** for state transfer

Both MP & CS have serialization & memory copy overhead

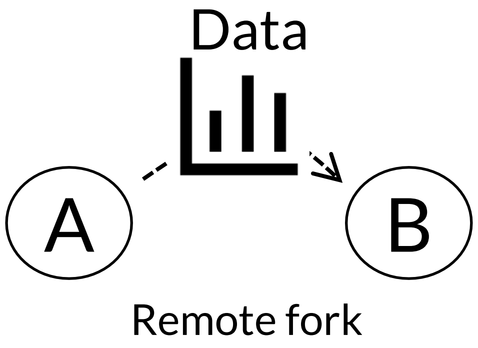
- ❑ Can attributes to **95%** of the total function execute time[1]



Cloud storage (e.g., S3)

[1] Faastlane: Accelerating function-as-a-service workflows, ATC'21

Fork for serialization-free state transfer


Data

A

B

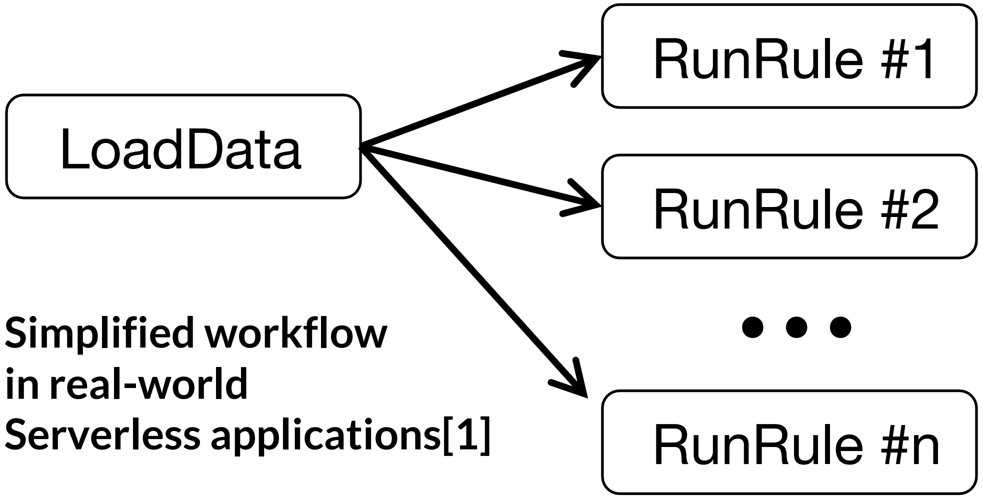
Remote fork

Suppose we want to run functions A & B

- Where B accesses data generated from A

If we fork B from A using MITOSIS, B can

- Transparently inherit A's data w/o serialization and memory copy!


LoadData

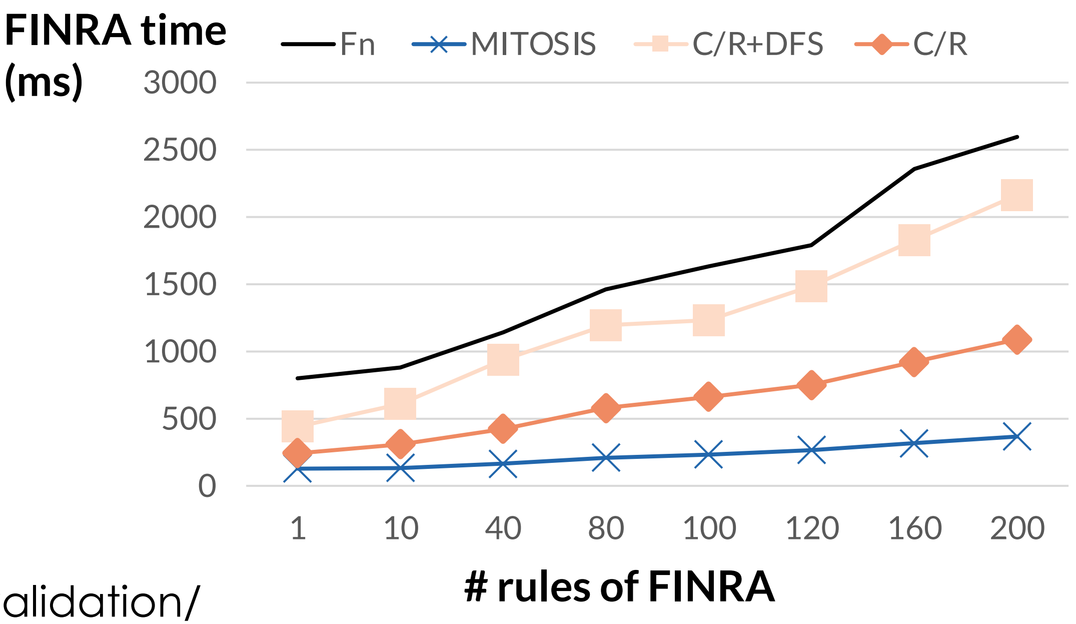
RunRule #1

RunRule #2

• • •

RunRule #n

**Simplified workflow
in real-world
Serverless applications[1]**



[1] https://aws.amazon.com/cn/solutions/case-studies/finra-data-validation/

More about MITOSIS, check our pre-print!

Detailed remote fork design & implementations

- ❑ Various tricks to make the fork fast, e.g. ,prefetch, generalized lean container, etc.

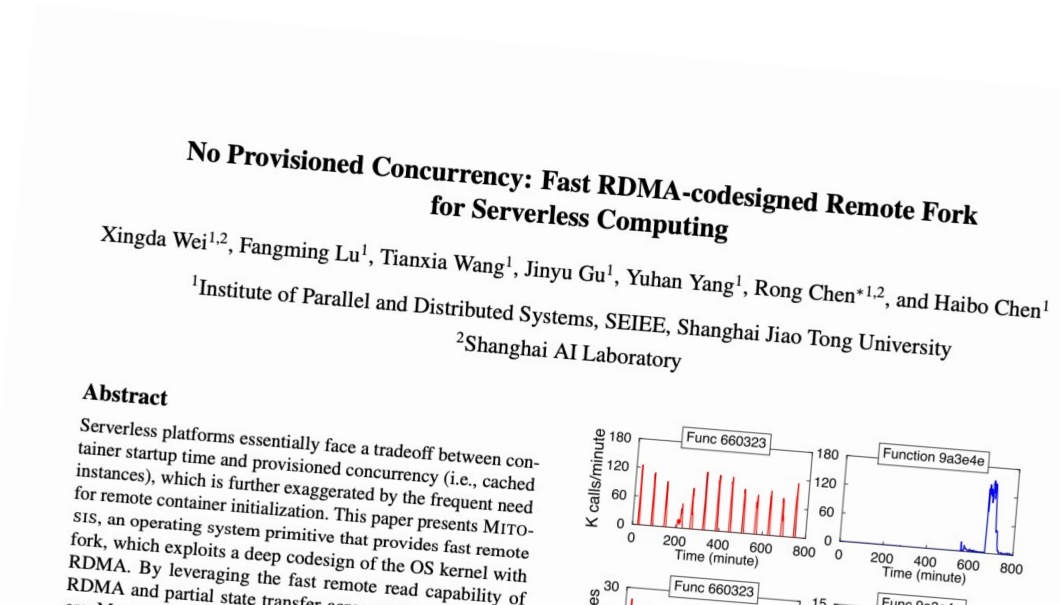
Memory protection & parent resource management

Integrations w/ serverless platforms

- ❑ For fast autoscale & state transfer

Limitations & future work

Pre-print available at:
https://arxiv.org/abs/2203.10225



No Provisioned Concurrency: Fast RDMA-codesigned Remote Fork for Serverless Computing

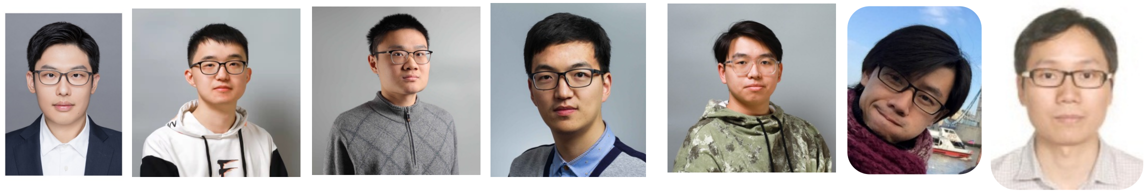
Xingda Wei[1,2], Fangming Lu[1], Tianxia Wang[1], Jinyu Gu[1], Yuhan Yang[1], Rong Chen*[1,2], and Haibo Chen[1]

[1]Institute of Parallel and Distributed Systems, SEIEE, Shanghai Jiao Tong University
[2]Shanghai AI Laboratory

Abstract

Serverless platforms essentially face a tradeoff between container startup time and provisioned concurrency (i.e., cached instances), which is further exaggerated by the frequent need for remote container initialization. This paper presents MITOSIS, an operating system primitive that provides fast remote fork, which exploits a deep codesign of the OS kernel with RDMA. By leveraging the fast remote read capability of RDMA and partial state transfer

Conclusion, Thanks & QA



MITOSIS: Fast remote fork design & implementation

- With a codesign between OS and RDMA

Achieve **no provisioned concurrency** for serverless functions

- Fork 10,000+ containers within one second across 5 machines

Achieve **fast state transfer** between functions

- With no memory copy & data serialization & deserialization overhead

Publicly available at:

https://github.com/ProjectMitosisOS/ProjectMitosisOS