IPADS INSTITUTE OF PARALLEL AND DISTRIBUTED SYSTEMS

上海交通大學 SHANGHAI JIAO TONG UNIVERSITY

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

Xingda Wei, Fangming Lu, Tianxia Wang,

Jinyu Gu, Yuhan Yang, Rong Chen, Haibo Chen

Problem: container startup is slow for ephemeral functions

E.g., `docker run` *`SOME_IMG`* `python foobar.py`

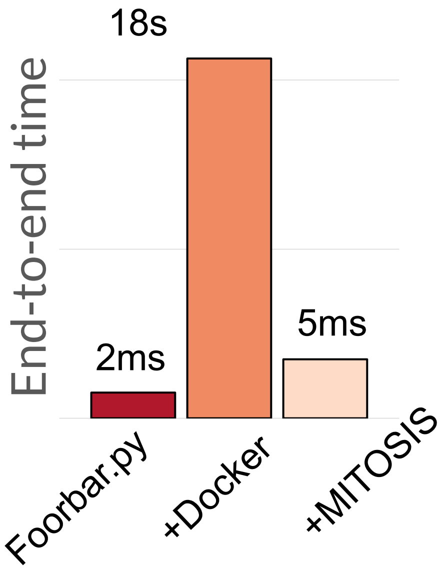
– The foorbar executes a simple program

– However, container startup causes **9,000X slower** to the program's execution (18s)

foobar.py

```
import time

print("hello world")
```



MITOSIS: fast container startup with minimal resource usage

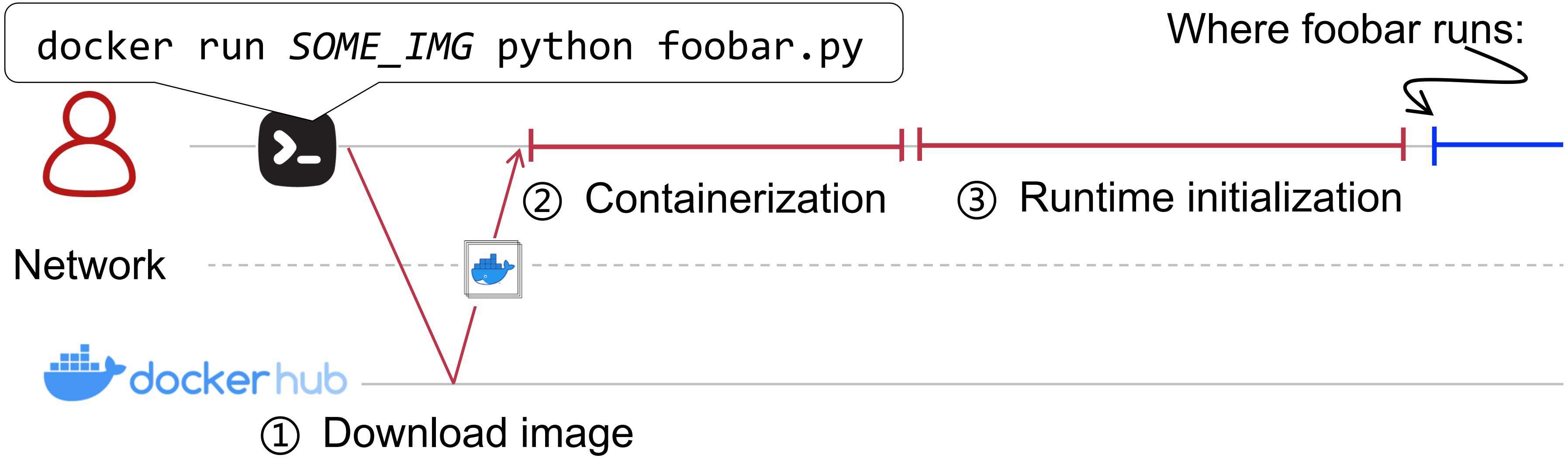
– Container startup **< 5ms** on a clean machine (fastest method)

– Start more than **100,000** containers on 5 machines in one second

Why container (cold) start is slow?

Start containers to run the application code involve many steps:

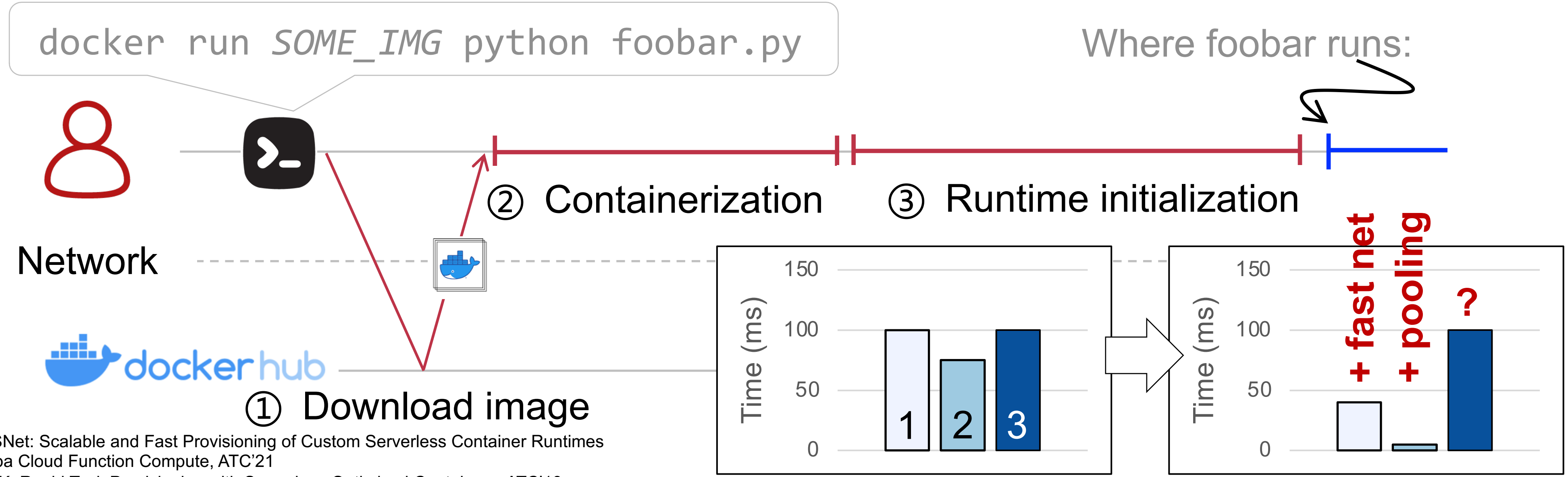
- Download the container image from a registry
- Containerization: setup `cgroup` and `namespaces`
- Runtime initialization: initialize Python runtime, import libraries (e.g., import torch)



How to accelerate the startup?

Potential solutions to accelerate each step:

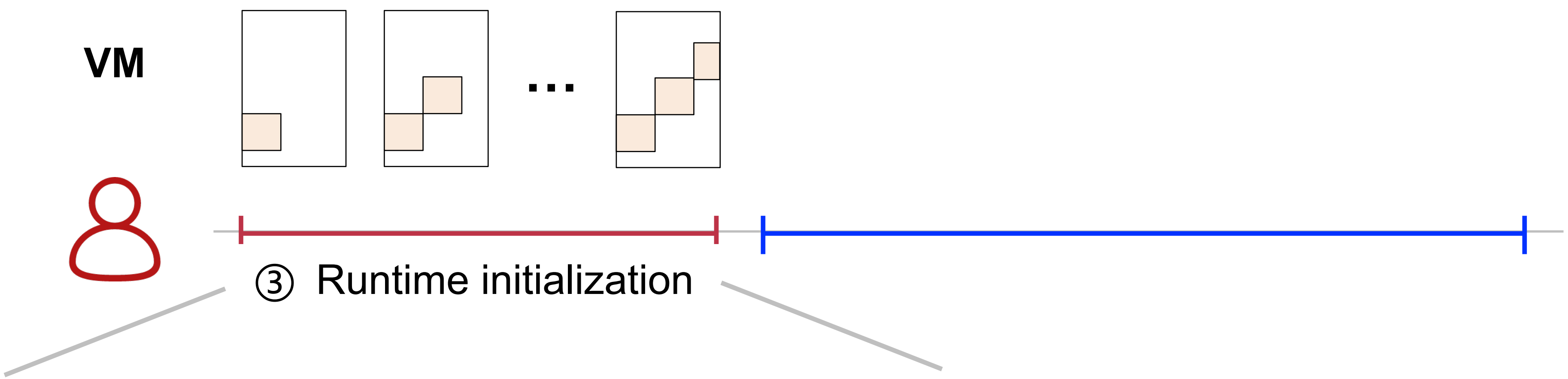
– Download image: optimize the pull, **but still has a cost** ([1])
– Containerization: use `cgroup` and namespace pooling to hide its cost [2]
– Runtime initialization: **?**



[1] FaaSNet: Scalable and Fast Provisioning of Custom Serverless Container Runtimes at Alibaba Cloud Function Compute, ATC'21
[2] SOCK: Rapid Task Provisioning with Serverless-Optimized Containers, ATC'18

Idea: reusing initialized state from other containers

Observation: runtime initialization + image == initialize container virtual memory



3.1 load the downloaded modules

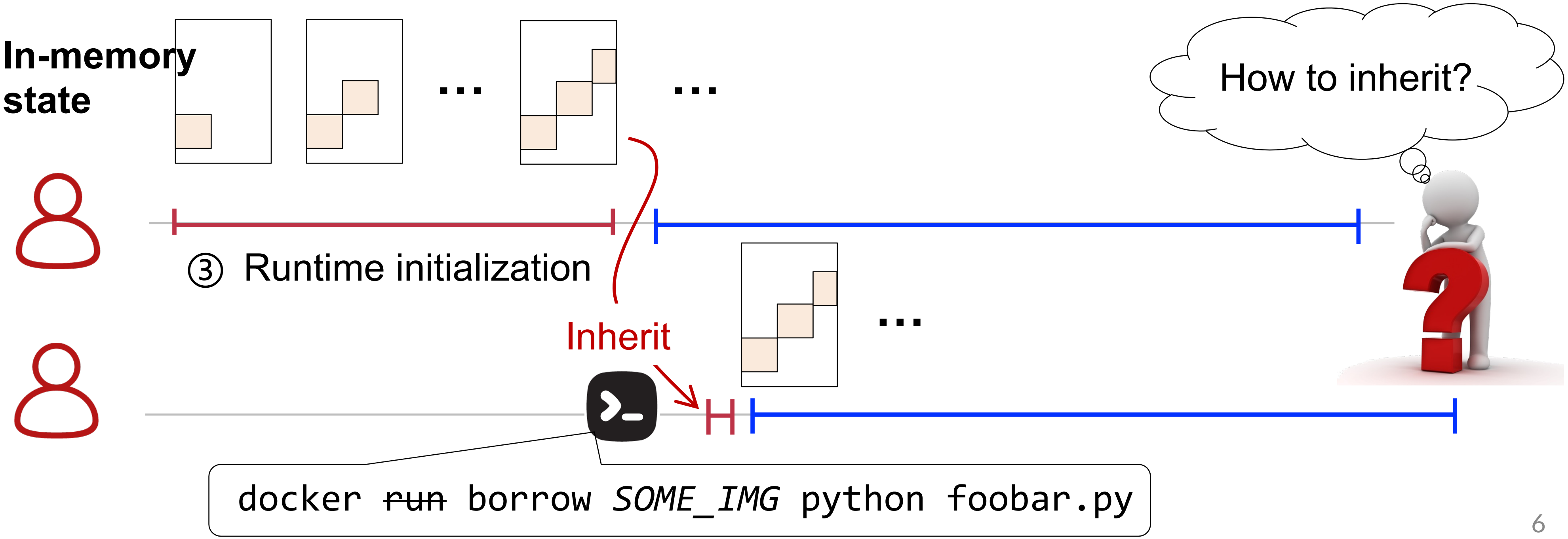
3.2 Translate to byte code

3.3 Execute module init routines

Idea: reusing initialized state from other containers

Observation: runtime initialization + image == initialize container virtual memory

- A new container can inherit the state from another initialized container
- No need to download the image or initialize the runtime



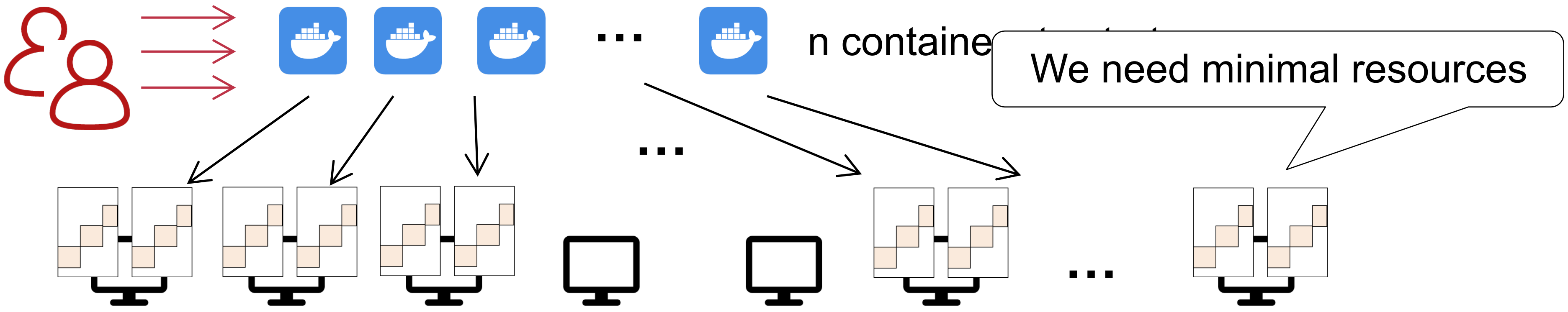
Design requirement: no provisioned concurrency

Suppose we have n containers to start, how many initialized states to store?

– The required number of stored states is usually termed as provisioned concurrency

Ideal case: no provisioned concurrency

– The provisioned case is irrelevant to the started containers, e.g., **O(1)**



Clusters to run the containers

Existing solutions need provisioned concurrency

Approach #1. Caching, a.k.a, warm start

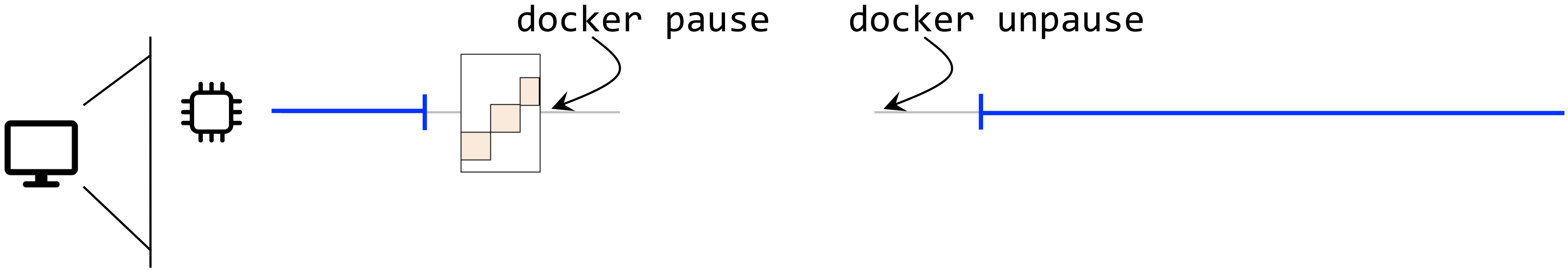
– E.g., `docker pause + docker unpause`

Docker pause

– Stop a container and store its state in DRAM

Docker unpause

– Resume the container for execution



Existing solutions need provisioned concurrency

Approach #1. Caching, a.k.a, warm start

– E.g., `docker pause + docker unpause`

Docker pause

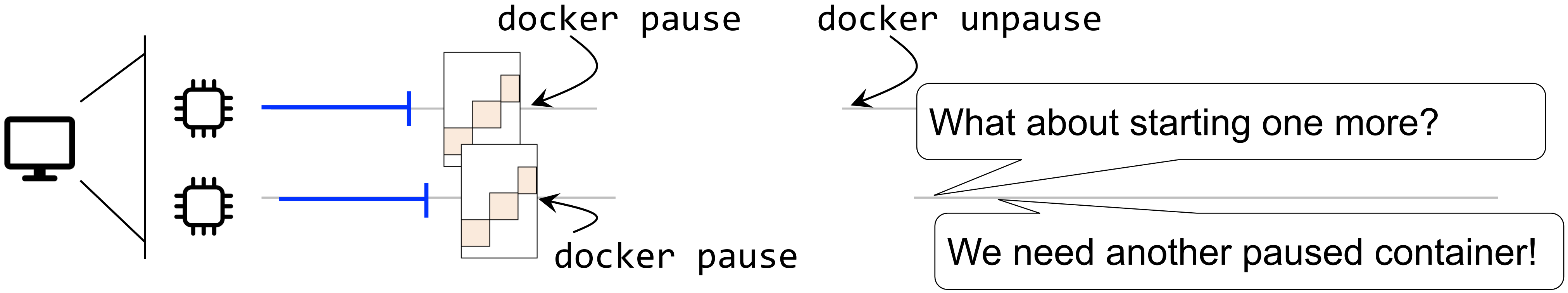
– Stop a container and store its state in DRAM

Docker unpause

– Resume the container for execution

Cons: needs provisioned concurrency!

O(n) containers provisioned, n: the number of concurrent invocations



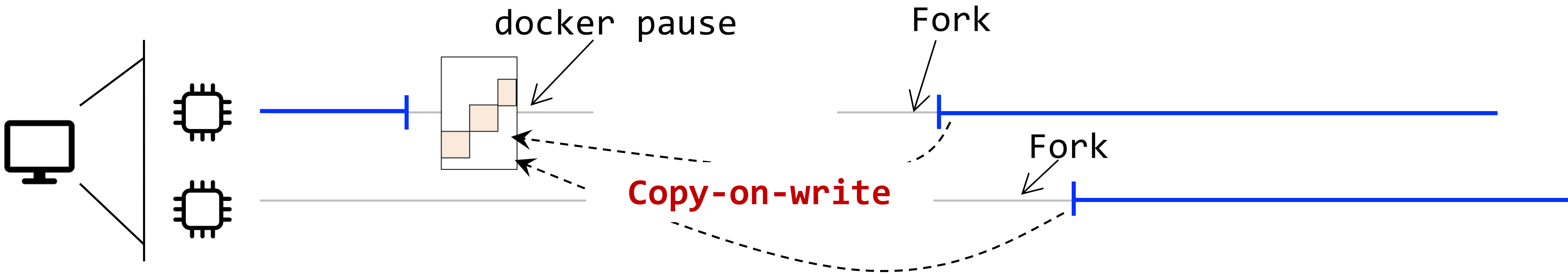
Existing solutions need provisioned concurrency

Approach #2. Fork, a.k.a, start containers in a process forking manner [1,2]

```
Fork --- Create a new process from an existing one
```

Pros:

- Each machine only needs 1 parent to concurrently start many containers
- Achieve O(1) resource provisioned **on a single machine**

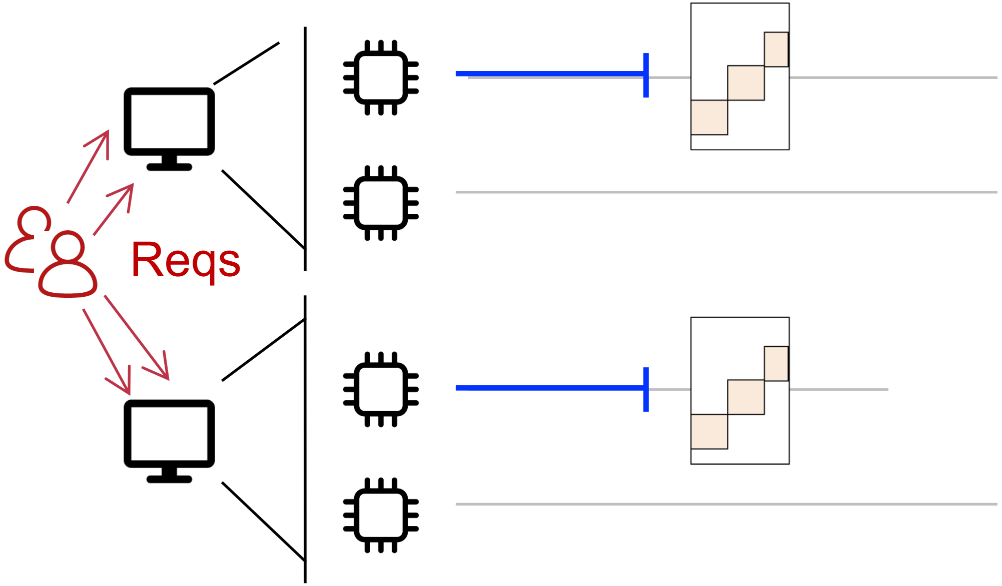


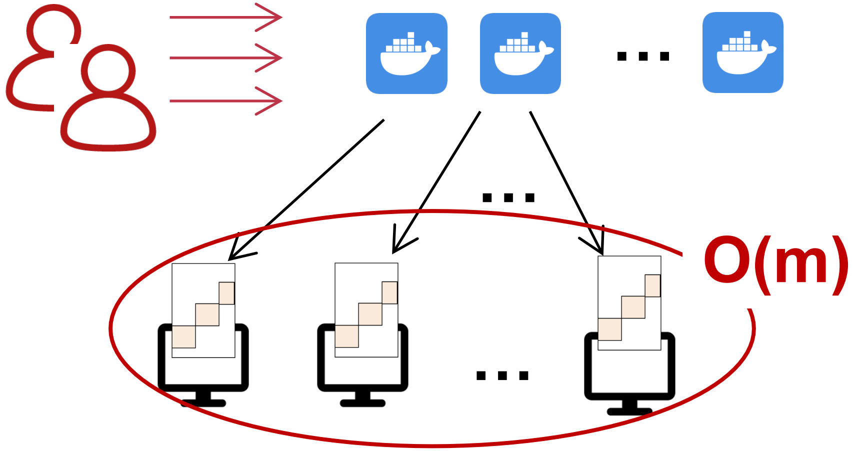
[1] Catalyzer: Sub-millisecond Startup for Serverless Computing with Initialization-less Booting, ASPLOS'20
[2] SOCK: Rapid Task Provisioning with Serverless-Optimized Containers, ATC'18

Existing solutions need provisioned concurrency

What if there is a load spike that applications want start many containers?

- E.g., there is a load spike in the workload
- Fork still need provisioned concurrency (**O(m)**) : deploy one parent on each machine!





Clusters w/ **m** machines to run the containers

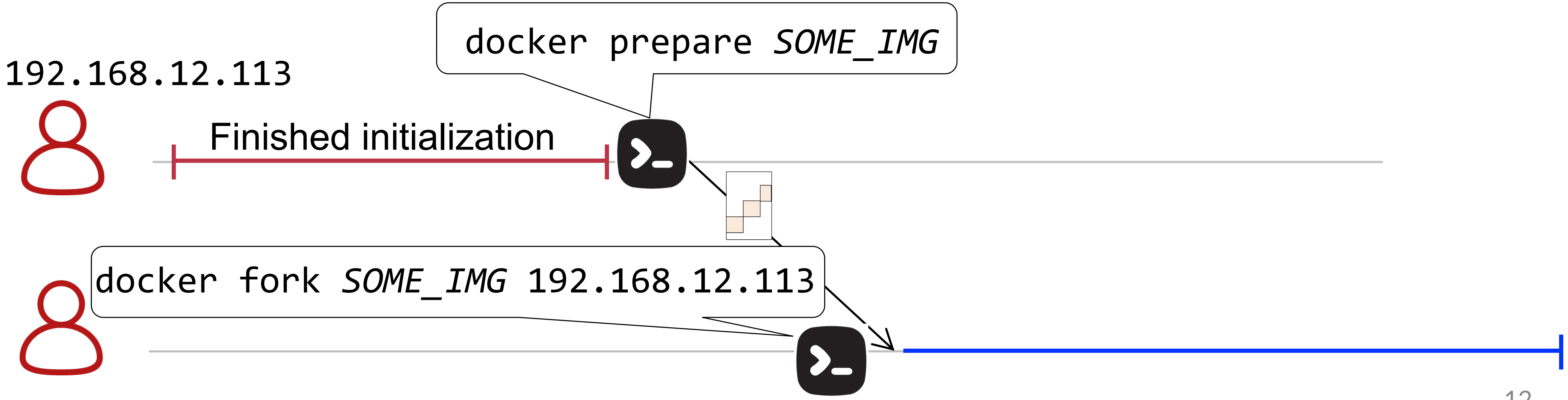
MITOSIS: remote fork ⇨ no provisioned concurrency

```
Fork --- Create a new process from an existing one
```

Remote fork is a primitive for no provisioned concurrency

– Observation: one parent is sufficient for starting containers across machines

– A generalization of fork to remote enabling no provisioned concurrency in a cluster

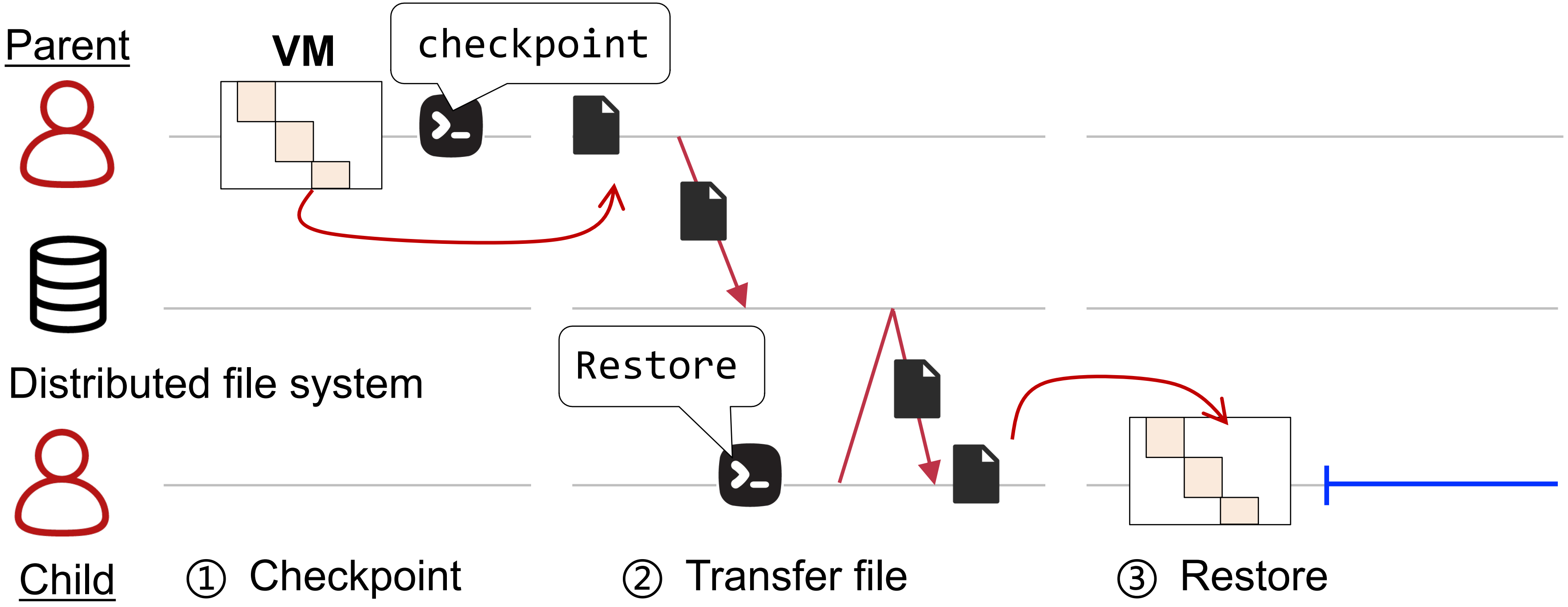


How to implement remote fork efficiently?

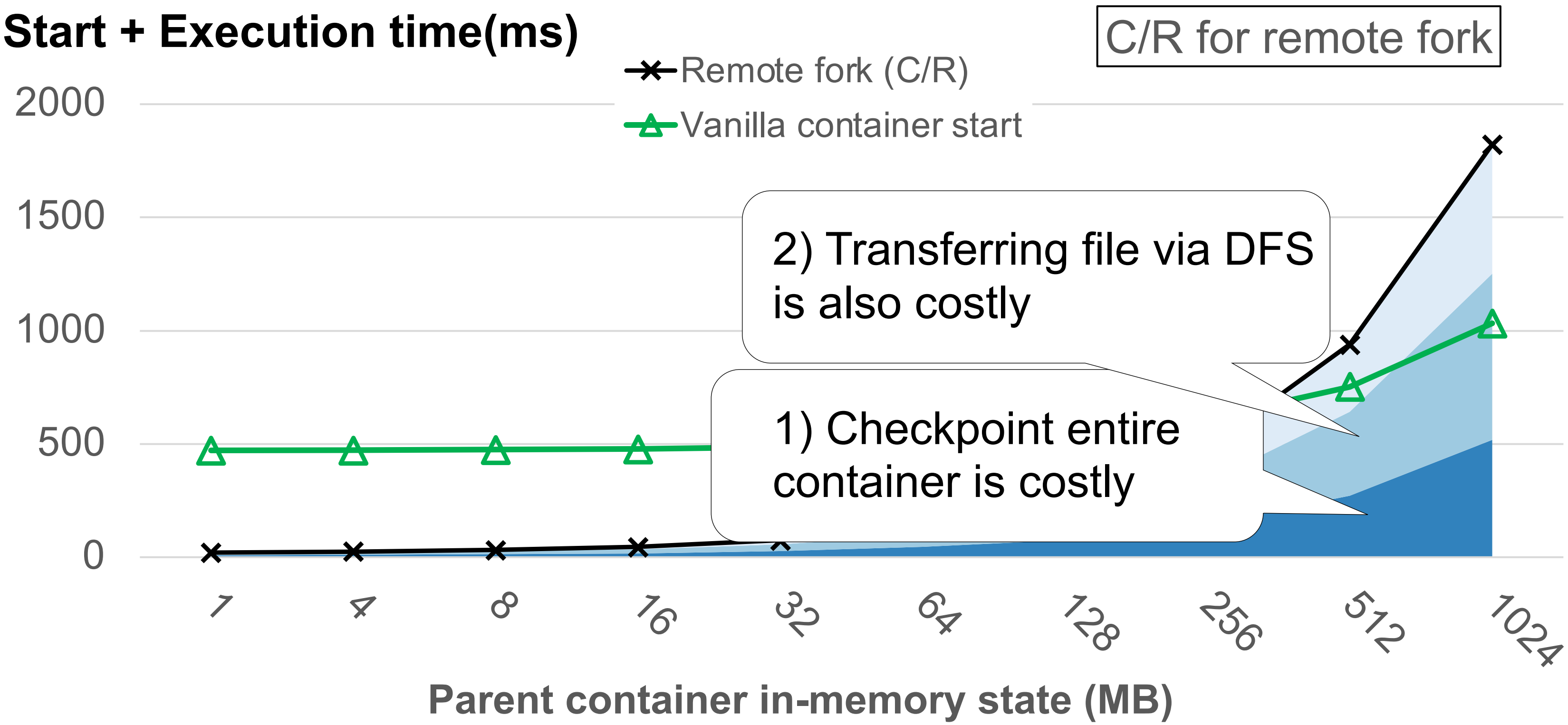
Current solution—Checkpoint & Restore (CRIU) is not efficient enough

– Checkpoint: stop and dump the memory to a file

– Restore: reconstruct the VM according to the file and resume the process



Current remote fork is not designed for RDMA



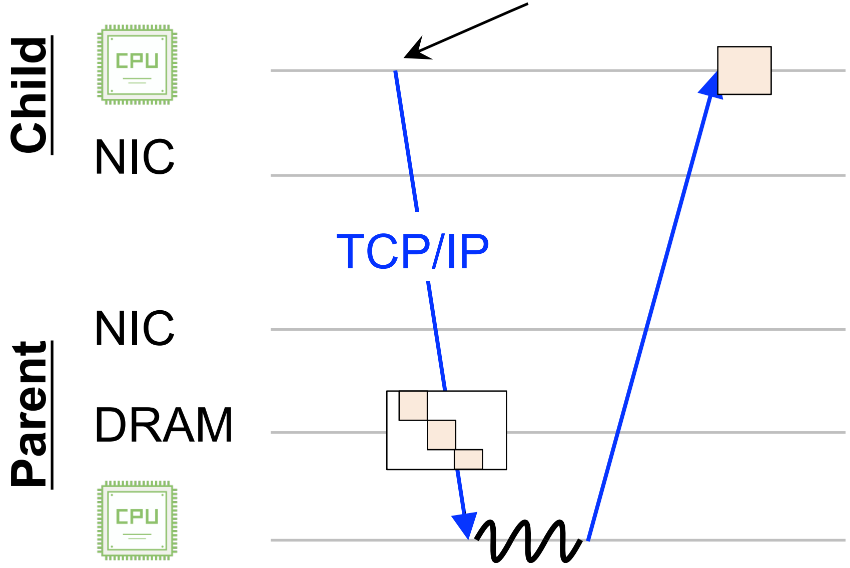
Evaluation setup: CRIU for C/R, file is transferred via RDMA and is stored in-memory

Opportunity: Remote Direct Memory Access (RDMA)

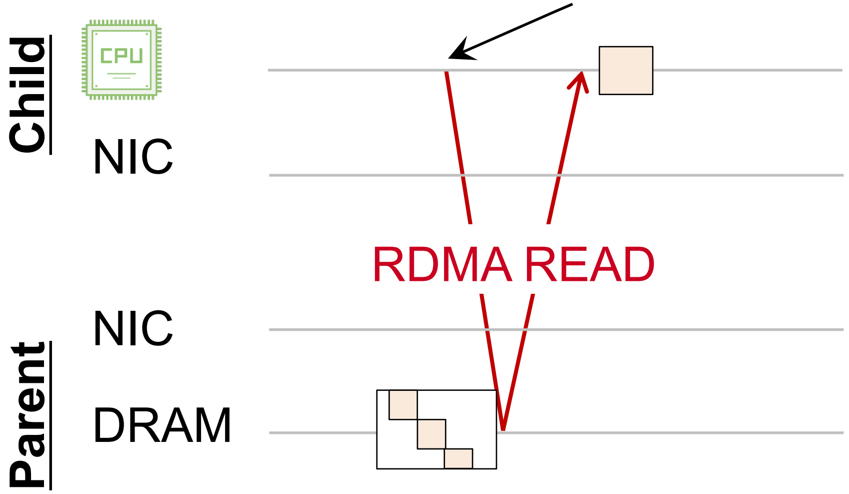
A fast datacenter networking feature that allows direct remote memory access

- High bandwidth (400Gbps) & low latency (600ns)
- CPU bypassing: the memory read/writes are offloaded to the NIC hardware

Read page at `0xdeadbeaf`



Read page at `0xdeadbeaf`



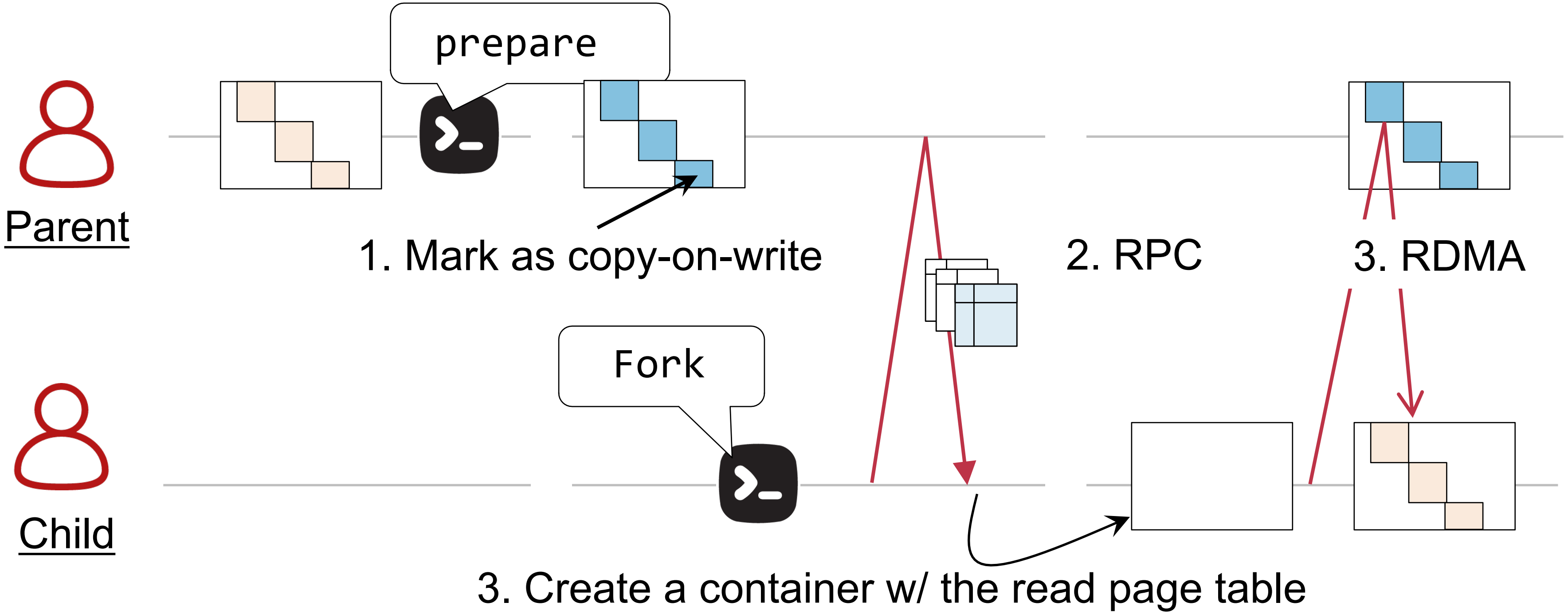
We can imitate local fork w/ RDMA!

MITOSIS co-designs remote fork with RDMA

Upon fork, we first use RDMA-based RPC to read the page table to the child

– One-sided RDMA is not efficient at this step due to network amplification

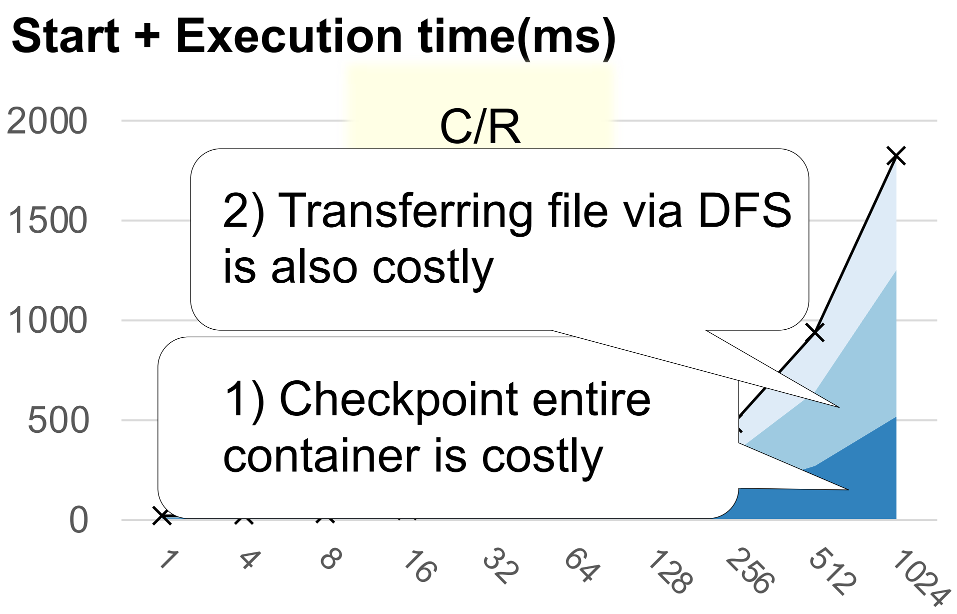
Afterward, the child retrieves memory pages in a RDMA-on-access manner (on demand)

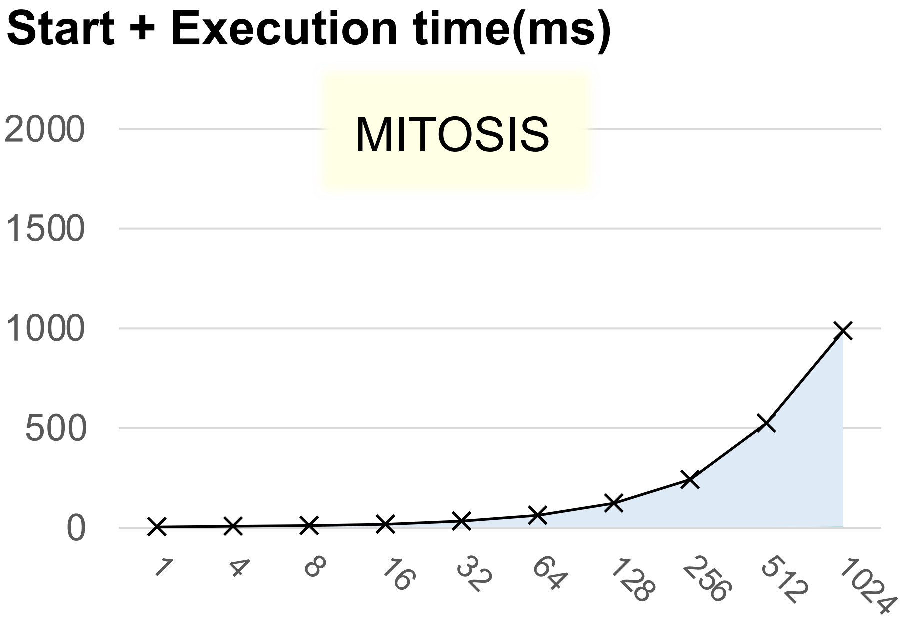


MITOSIS co-designs remote fork with RDMA

44—80% faster than basic C/R[1] not co-designed with RDMA

– The C/R implementation has used RDMA-based DFS to restore states

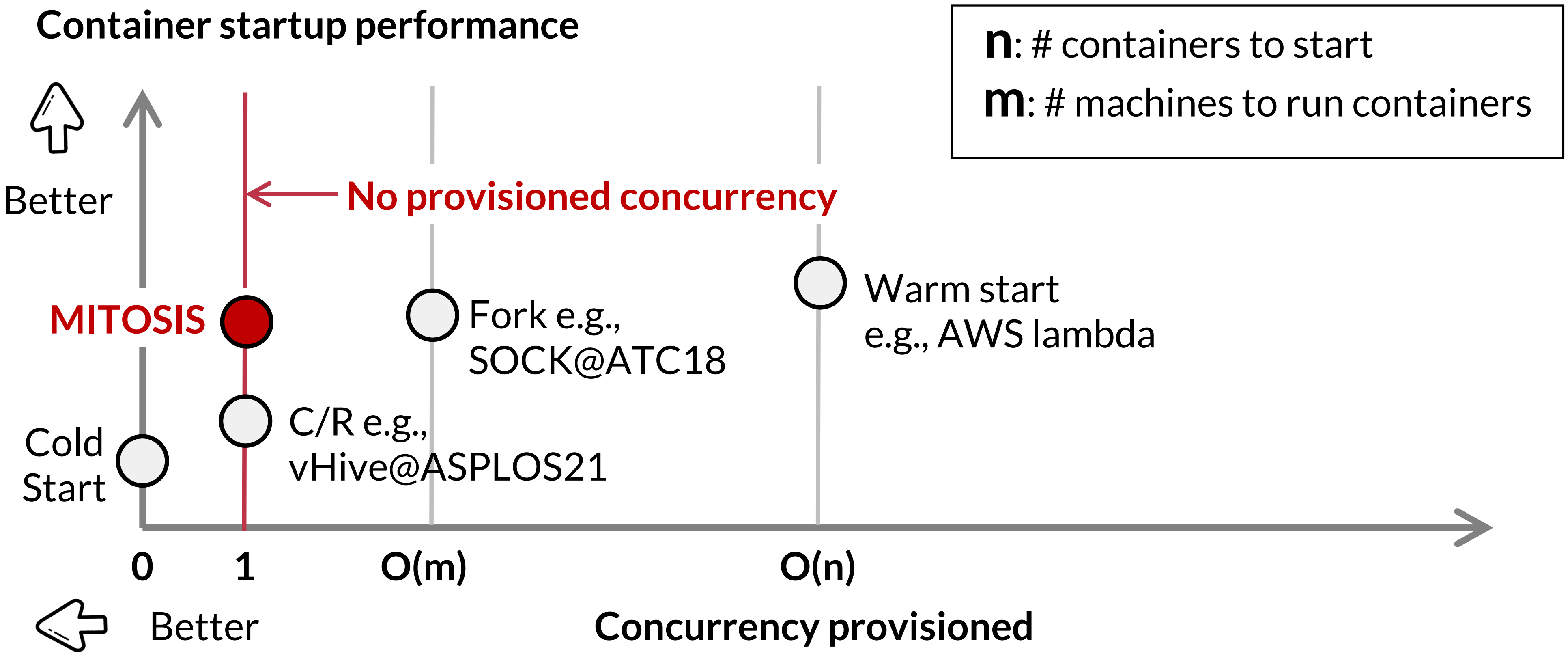




Parent container in-memory state (MB)

[1] CRIU: The state-of-the-art impl of C/R

MITOSIS vs. The state of the arts



Killer application of MITOSIS: Serverless Computing

A new paradigm on building cloud applications

- Users upload application as functions
- Each function is executed in a container for the ease of deployment

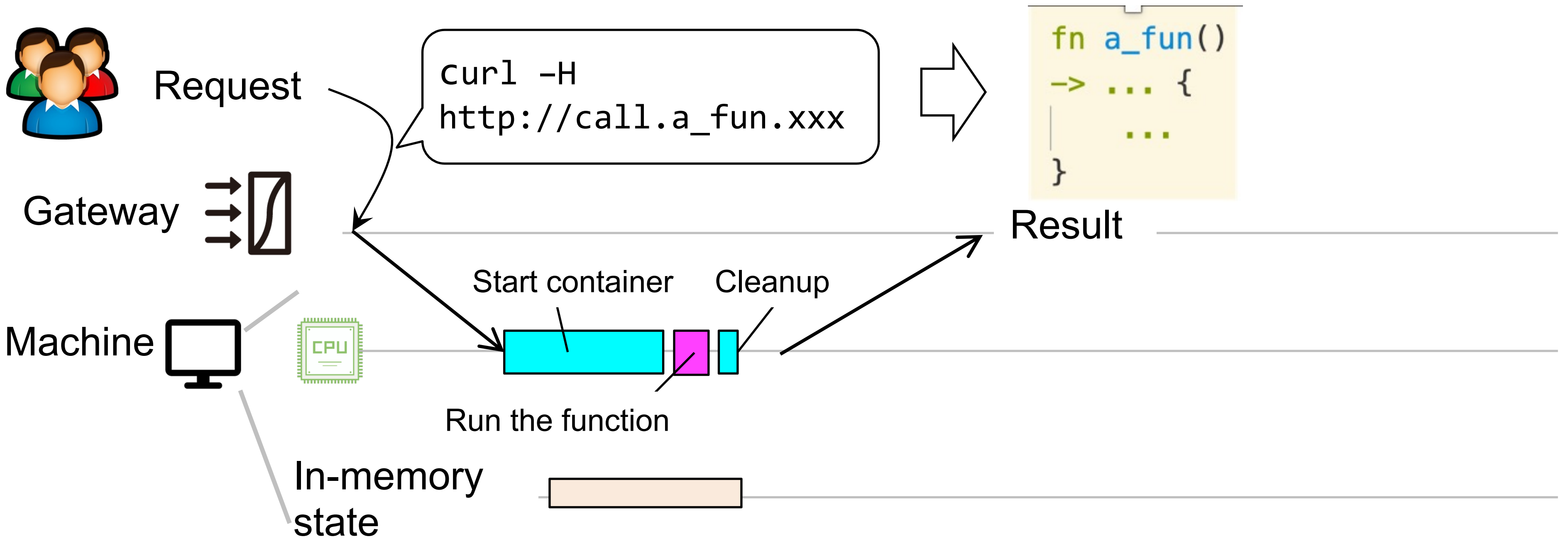
AWS Lambda | Microsoft Azure | Google Functions | Huawei cloud functions | Opensoruce platforms

Two key attributes to serverless computing

1. Fast container startup for **resource-efficient auto-scaling**
2. **Fast state transfer** between serverless functions---no (de)serialization !

Case study #1. Resource-efficient auto-scaling

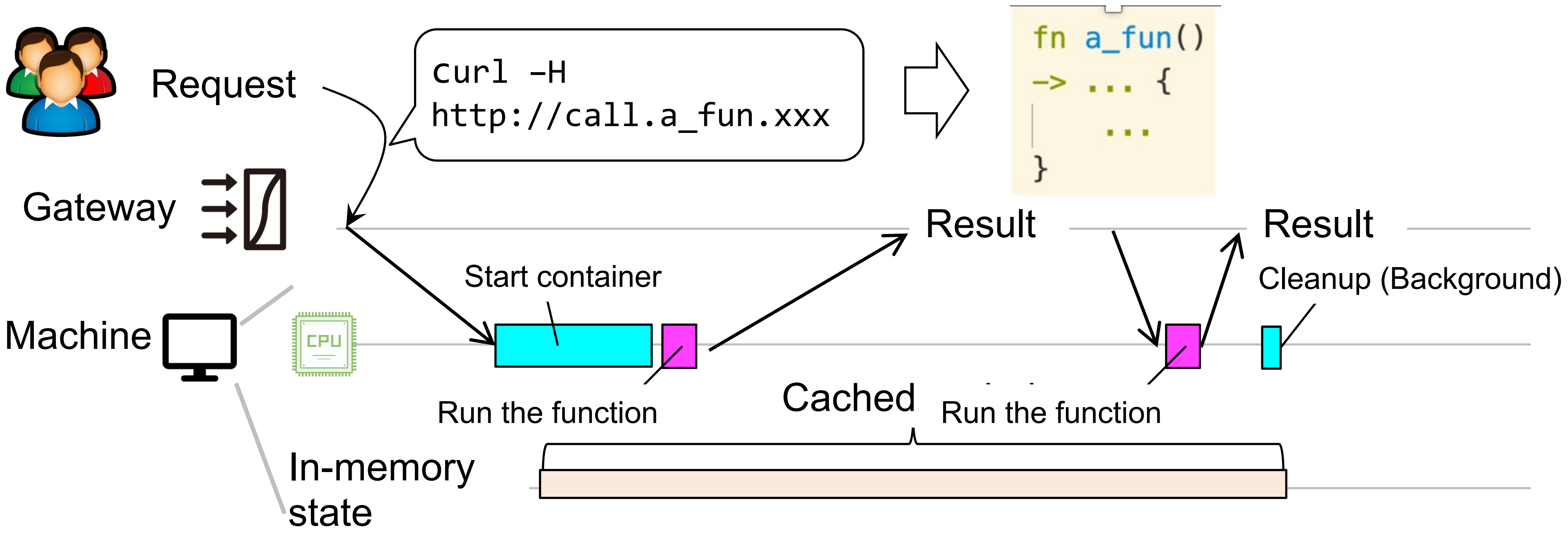
For elasticity, each serverless function invocation will start a new container



Case study #1. Resource-efficient auto-scaling

For elasticity, each serverless function invocation will start a new container

– The container can be **cached** for a short period (e.g., 30 secs) to prevent cold start

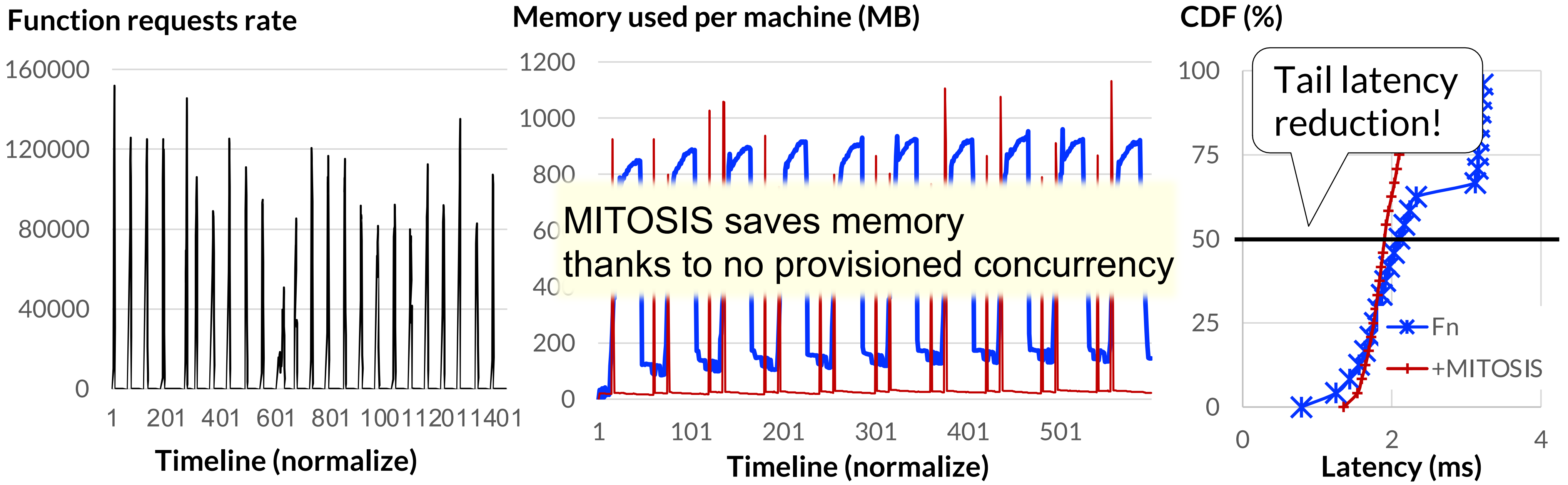


Results: handling load spikes in a more resource efficient way

Workloads: trace from the Azure function [1] (Instance #660323)

– Concurrent function invocations in a load spike manner

– Setup: Fn , a local cluster w/ 24 machines; function: image processing

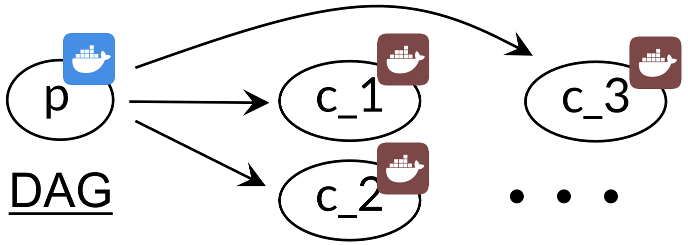


[1] Serverless in the wild: Characterizing and optimizing the serverless workload at a large cloud provider. ATC'20

Case study #2: accelerate state transfer between functions

Serverless function can compose multiple functions together

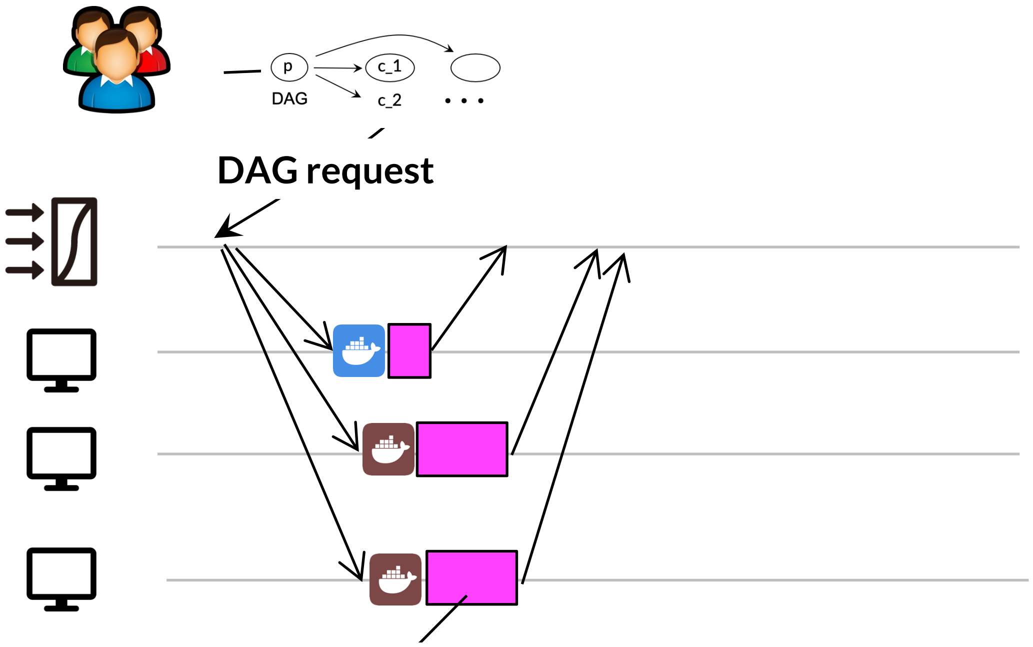
– The functions are typically organized into a DAG (Direct acyclic graph)




```
def produce():
    data = pd.read_csv(some_csv)
    return data
```

```
def consumer_1(data):
    process_data_1(data)
```

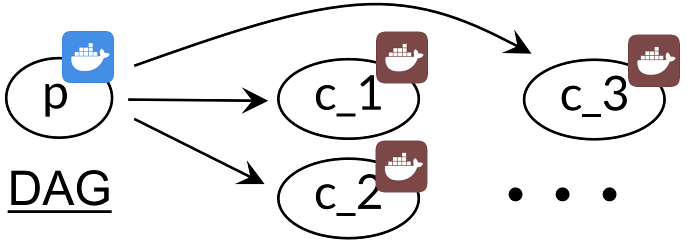
• • •



Case study #2: accelerate state transfer between functions

Serverless function can compose multiple functions together

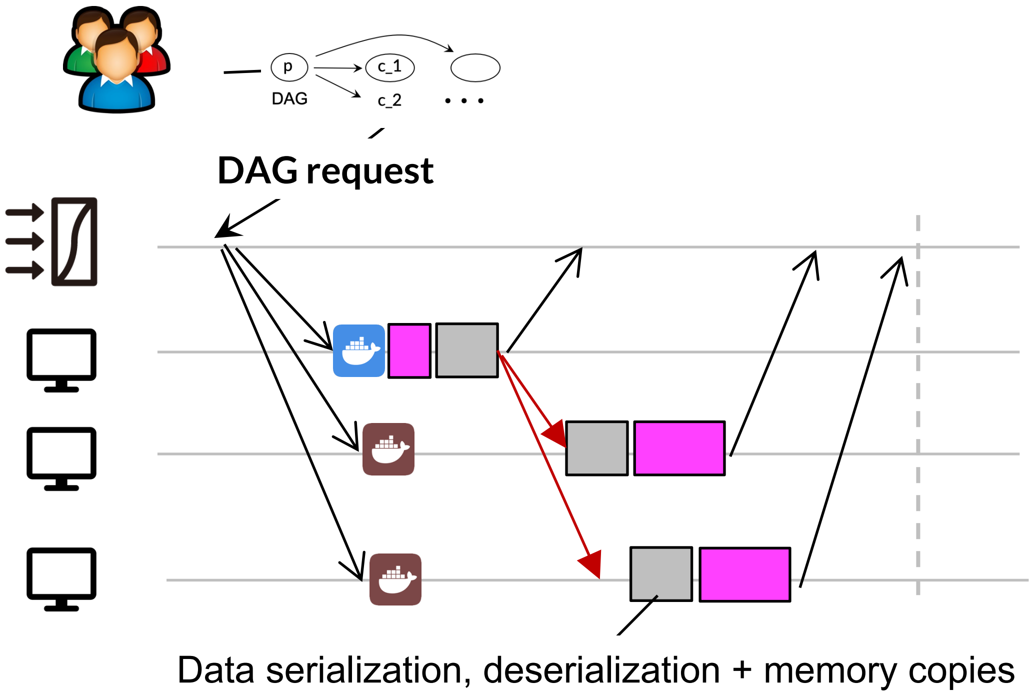
- The functions are typically organized into a DAG (Direct acyclic graph)
- **Problem**: Transferring states are costly due to (de)serialization + memory copies




```
def produce():
    data = pd.read_csv(some_csv)
    return data
```

```
def consumer_1(data):
    process_data_1(data)
```

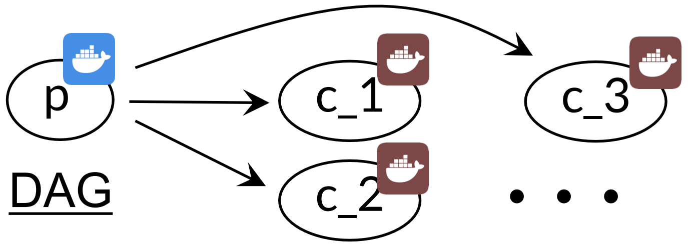
• • •



Case study #2: accelerate state transfer between functions

Remote fork can completely address the costs of (de)serialization + memory copies

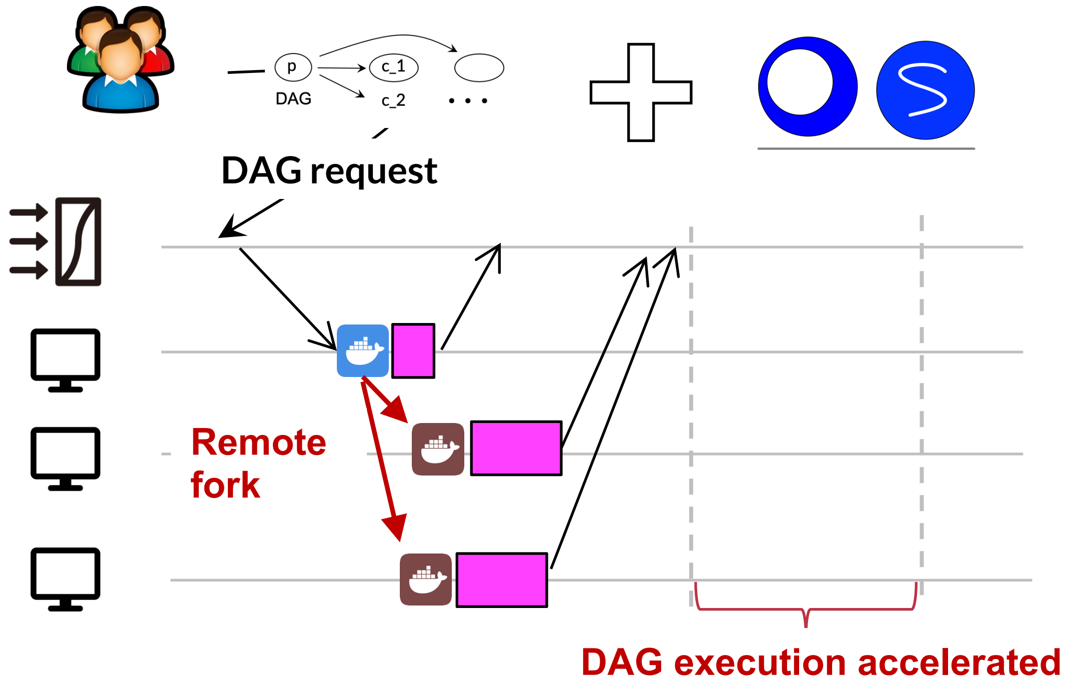
- The data has been **pre-materialized** in the parent memory
- Which is directly inherited by the child containers w/ the help of remote fork




```
def produce():
    data = pd.read_csv(some_csv)
    return data
```

```
def consumer_1(data):
    process_data_1(data)
```

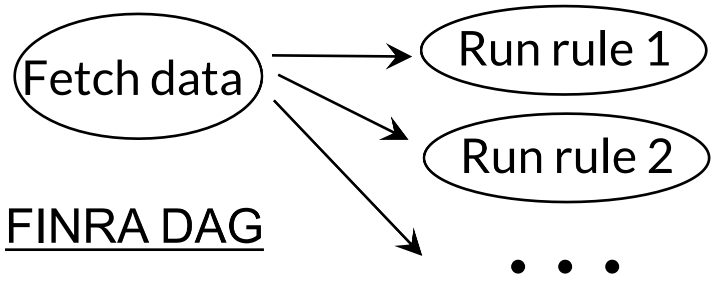
• • •

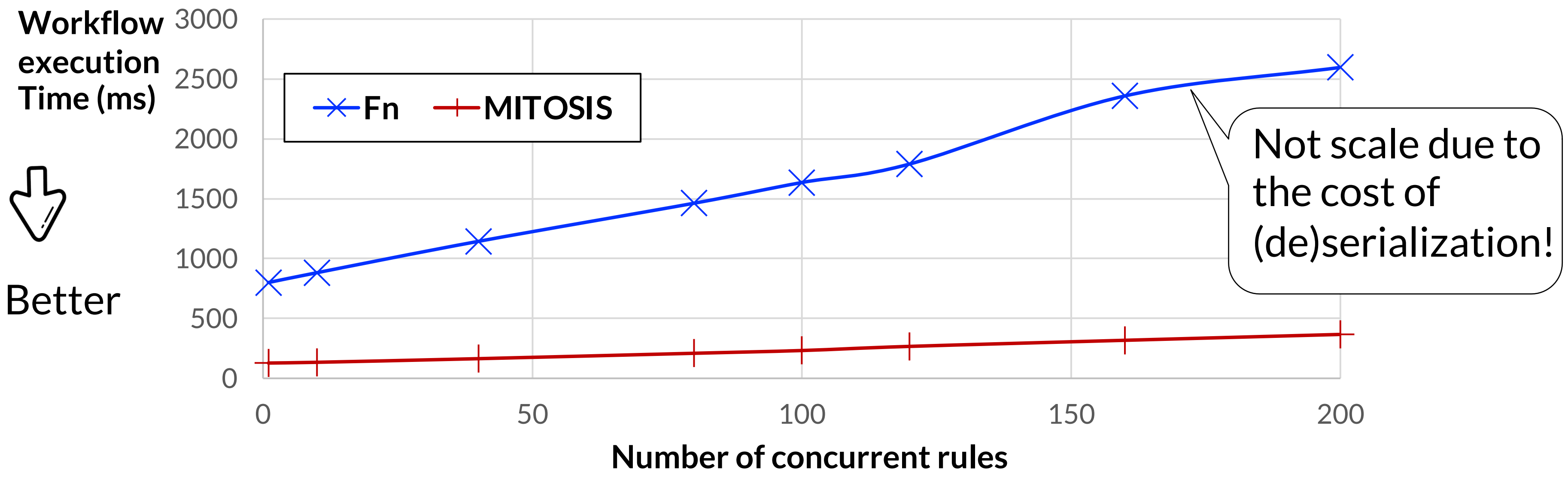


Transfer state has a high cost, MITOSIS can accelerate it!

Workloads: FINRA---a real-world serverless application

- Validate trades concurrently with serverless functions
- **Setup**: Fn, baseline adopts `pickle` for (de)serialization

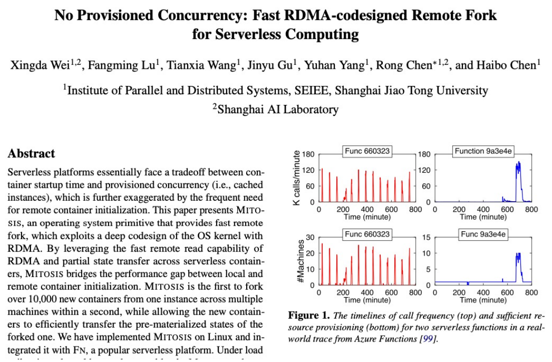




[1] https://aws.amazon.com/cn/solutions/case-studies/finra-data-validation/

Many technical challenges to bring RDMA to remote fork

1. **Detailed implementation w/ RDMA**
   - On-demand vs. eager state inherit
   - Performance optimizations, e.g., caching or prefetch
2. **Memory management w/ RDMA**
   - A co-design with advanced RDMA technologies
3. **Integration w/ serverless framework**
   - A strong cooperation is needed so as to fully utilize the power of MITOSIS
4. **More detailed evaluations**
   - Where the performance improvement comes, & the bottleneck of approach, etc.


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

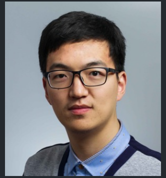
Serverless platforms essentially face a tradeoff between container startup time and provisioned concurrency (i.e., cached instances), which is further exaggerated by the frequent need for remote container initialization. This paper presents MITOSIS, an operating system primitive that provides fast remote 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 across serverless containers, MITOSIS bridges the performance gap between local and remote container initialization. MITOSIS is the first to fork over 10,000 new containers from one instance across multiple machines within a second, while allowing the new containers to efficiently transfer the pre-materialized states of the forked one. We have implemented MITOSIS on Linux and integrated it with FN, a popular serverless platform. Under load

Figure 1. *The timelines of call frequency (top) and sufficient resource provisioning (bottom) for two serverless functions in a real-world trace from Azure Functions [99].*

Please check our paper if you have interests!

Conclusion, Thanks & QA




MITOSIS: Fast remote fork design & implementation for starting containers

- With a codesign between OS and RDMA

Achieve no provisioned concurrency

- O(1) resource usage for starting serverless containers

Killer application: serverless computing

- Achieve resource—performance—efficient coldstart mitigation
- Achieve (de)serialization-free state transfer between serverless functions

Publicly available at:

https://github.com/ProjectMitosisOS/ProjectMitosisOS