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

USENIX OSDI23

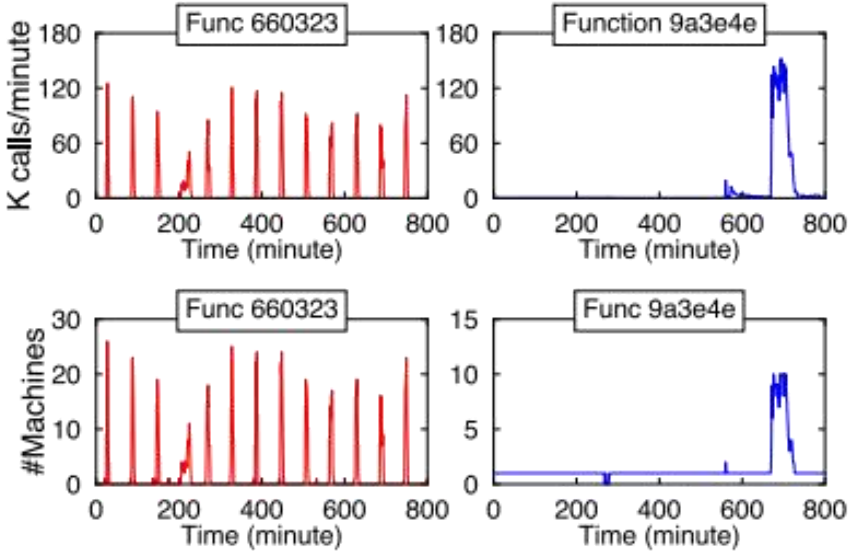
Xingda Wei

Institute of Parallel and Distributed Systems, SEIEE, Shanghai Jiao Tong University

2024.4.26

# 背景

- Serverless中，将功能缩放到多台机器是常见的，因为单台机器处理及时负载峰值的功能能力有限。
- 考虑从Azure函数的真实轨迹中采样的函数。函数9a3e4e的请求频率可以激增到每分钟150 K以上的调用，在一分钟内增加了33,000×。
- 为了避免阻断大量新到达的函数调用，平台应该立即在多台机器上启动足够的container。



# 问题：对于短暂的函数，container的启动速度很慢

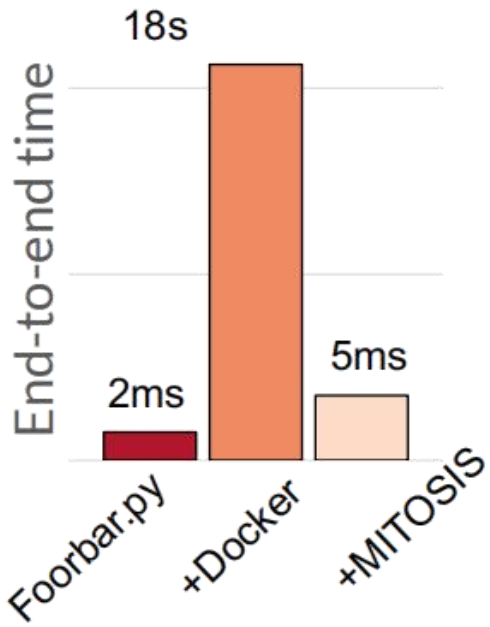
- E.g., docker run SOME_IMG python foobar.py
- foorbar.py执行了一个简单的程序
- 但是，container启动会使程序的执行速度慢9000倍（18秒）

foobar.py

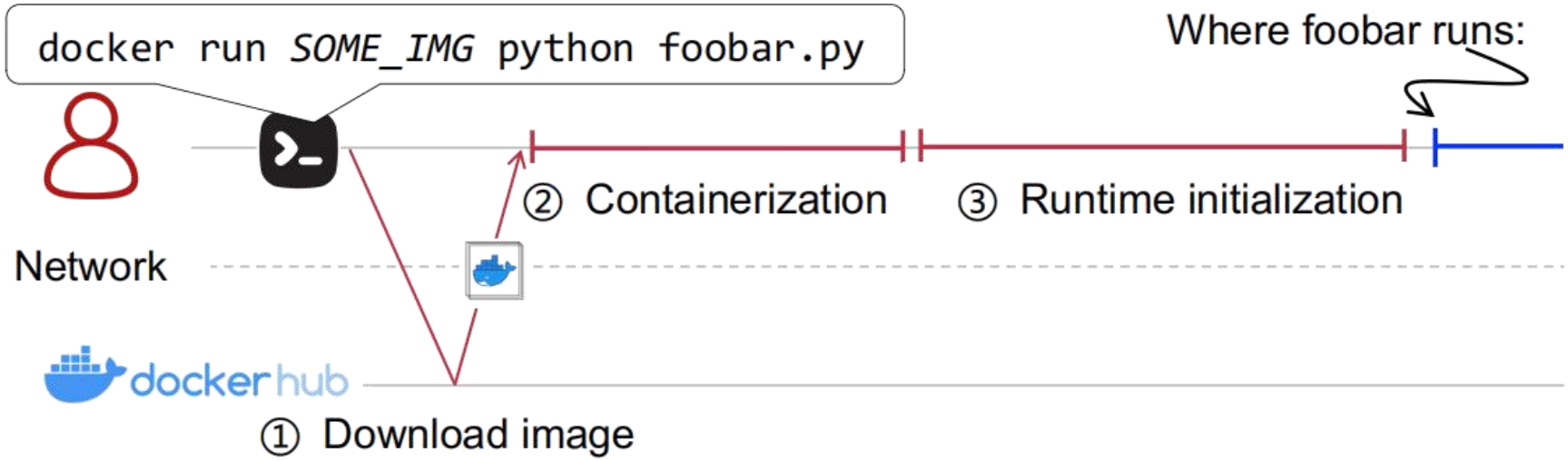
```
import time

print("hello world")
```

- MITOSIS：以最小的资源耗费，快速启动container
- 最快的方法下，container在空白机器上的starup<5ms
- 一秒钟内在5台机器上启动超过10万个container



# container冷启动为什么慢?

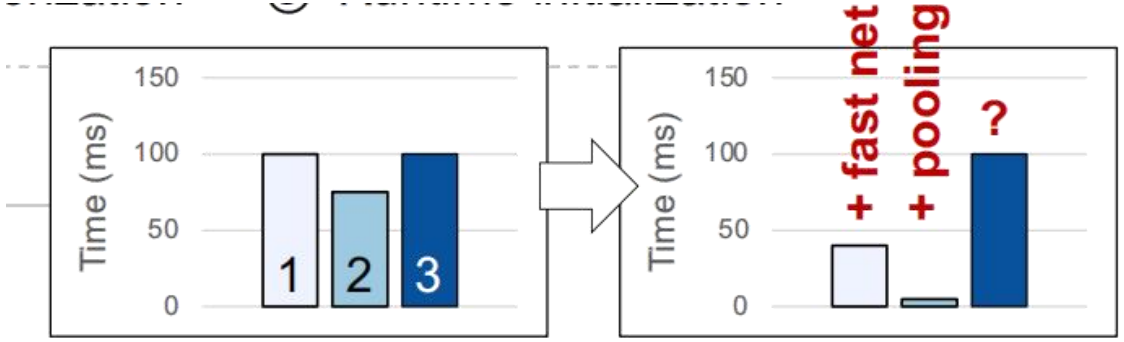



启动container来运行应用程序代码涉及多个步骤:

- 从registry(e.g., dockerhub, aliyun)中下载container image
- 容器化：设置cgroup和namespace
- 运行时初始化：初始化Python运行时，导入库（e.g., import pytorch）

# 如何为启动提速?

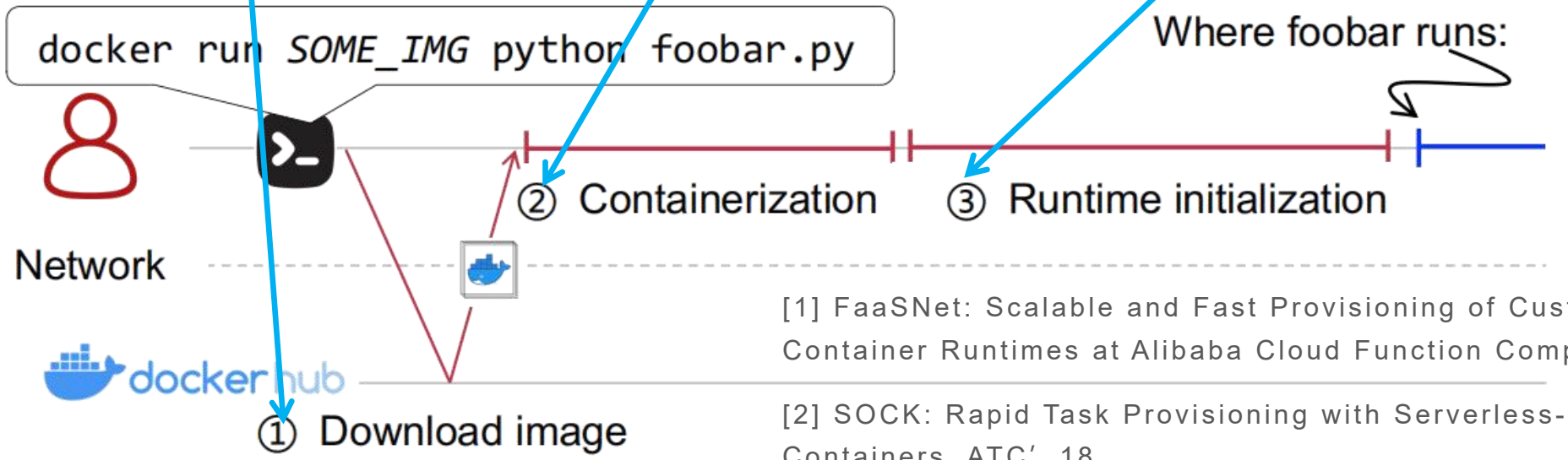
- 加速每一步的潜在解决方案:




Containerization：使用cgroup和namespace pool来隐藏其成本[2]

Download Image：优化pull操作，但仍有一定成本[1]

Runtime initialization：**可以怎么做?**

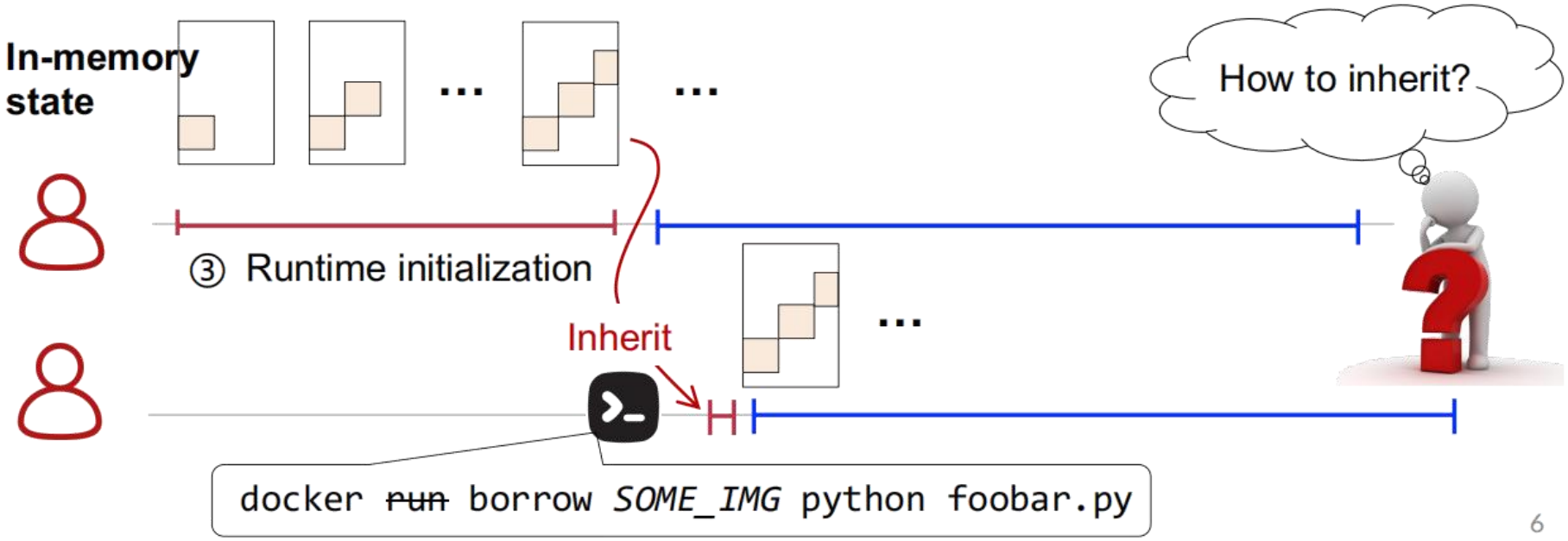



[1] FaaSNet: Scalable and Fast Provisioning of Custom Serverless Container Runtimes at Alibaba Cloud Function Compute, ATC' 21

[2] SOCK: Rapid Task Provisioning with Serverless-Optimized Containers, ATC' 18

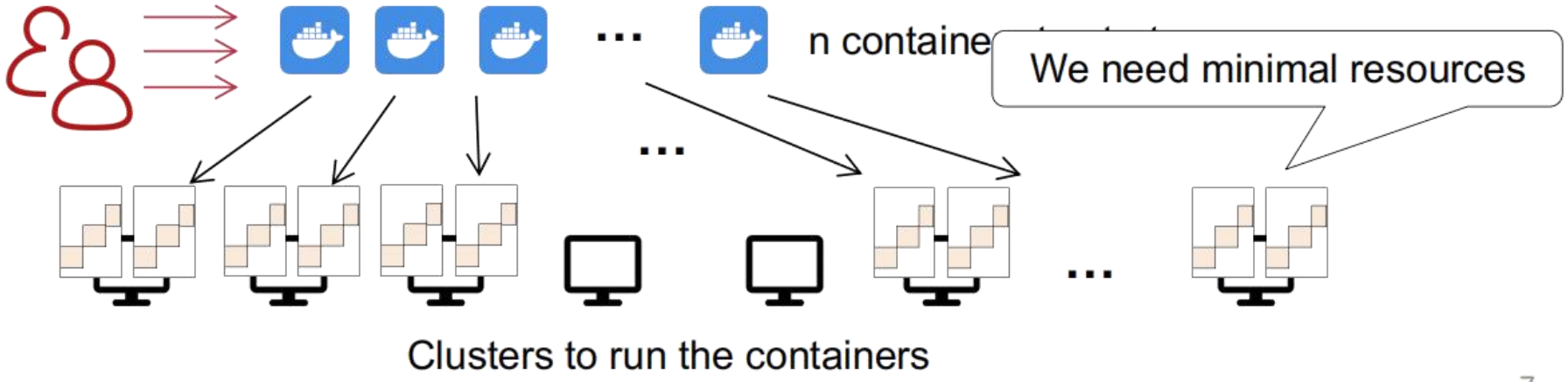
# idea:从其他container中reuse初始化状态

- 观察：运行时初始化+image==初始化container虚拟内存
  - 一个新的container可以从一个已初始化的container继承该状态
  - 不需要再下载image或初始化运行时



# 设计要求：没有已配置的并发

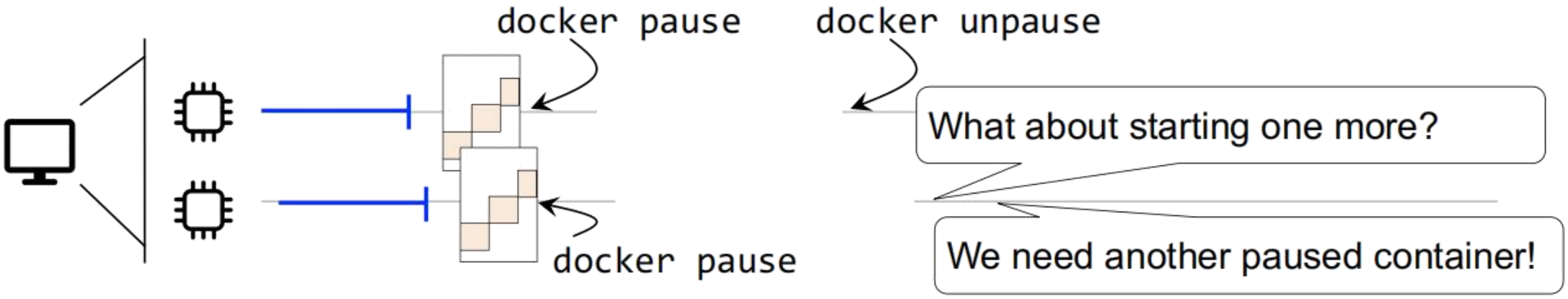
- 假设我们有n个container要启动，要存储多少个初始化的状态？
  - 所需的存储状态数通常被称为已配置的并发
  - 理想情况：没有已配置的并发，即已配置的实例与已启动的container无关
  - 为什么希望没有已配置的并发？
    - 已配置并发性通常意味着需要提前准备和维护一定数量的实例



# 现有的解决方案需要已配置的并发

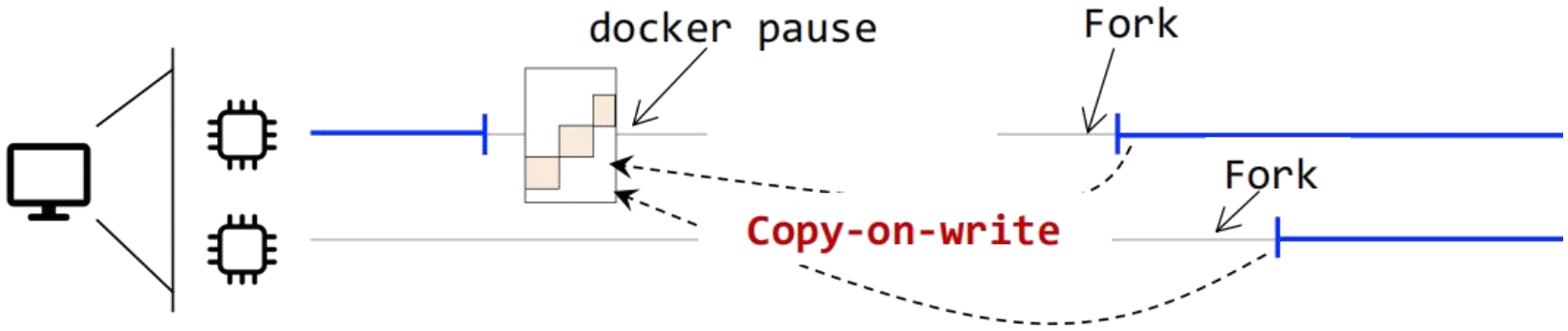
- 方案1：Caching,即暖启动
  - E.g., docker pause + docker unpause
  - docker pause：停止一个container，并在DRAM中存储其状态
  - docker unpause：恢复该container以供执行

**缺陷：需要O(n)数量的并发调用！**

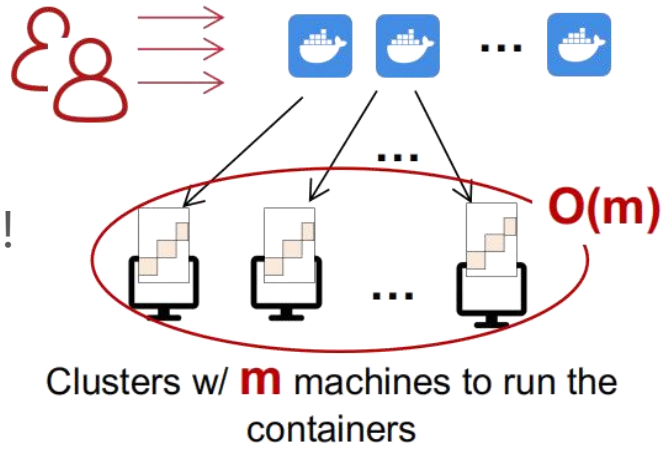


# 现有的解决方案需要已配置的并发

- 方案2：Fork,即以过程分叉的方式启动container
  - 每台机器只需要一个父节点，就可以同时启动多个container
  - 实现在单台机器上配置的O (1)个资源

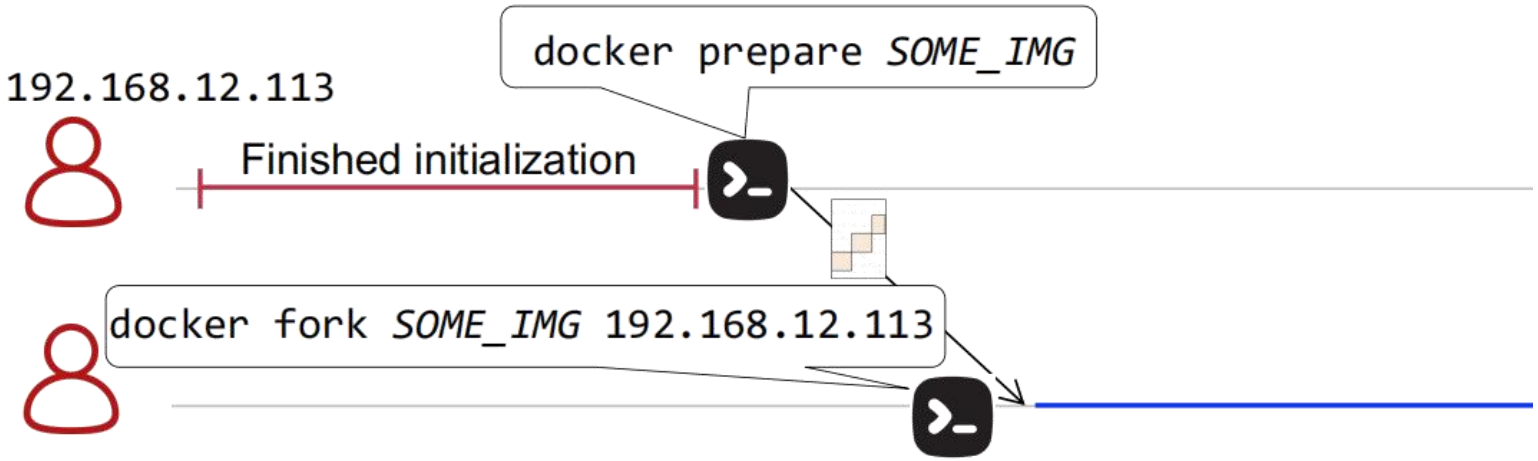



- 如果有一个应用程序需要启动许多container的负载峰值怎么办?
  - Fork仍然需要配置并发性（O (m)）：在每台机器上部署一个父节点!



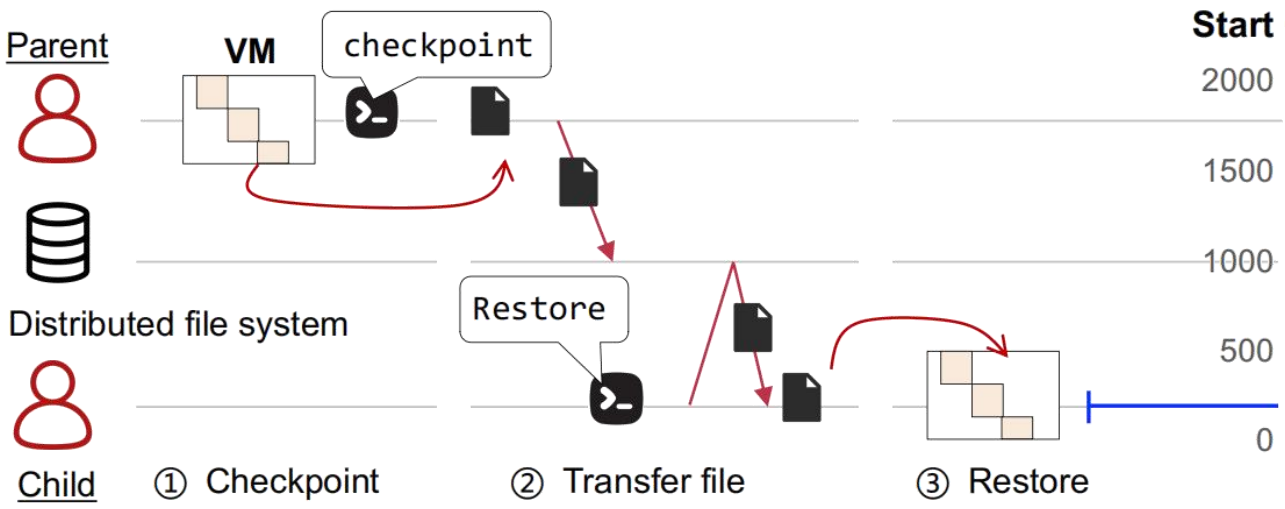
# MITOSIS：远程分叉→没有已配置的并发

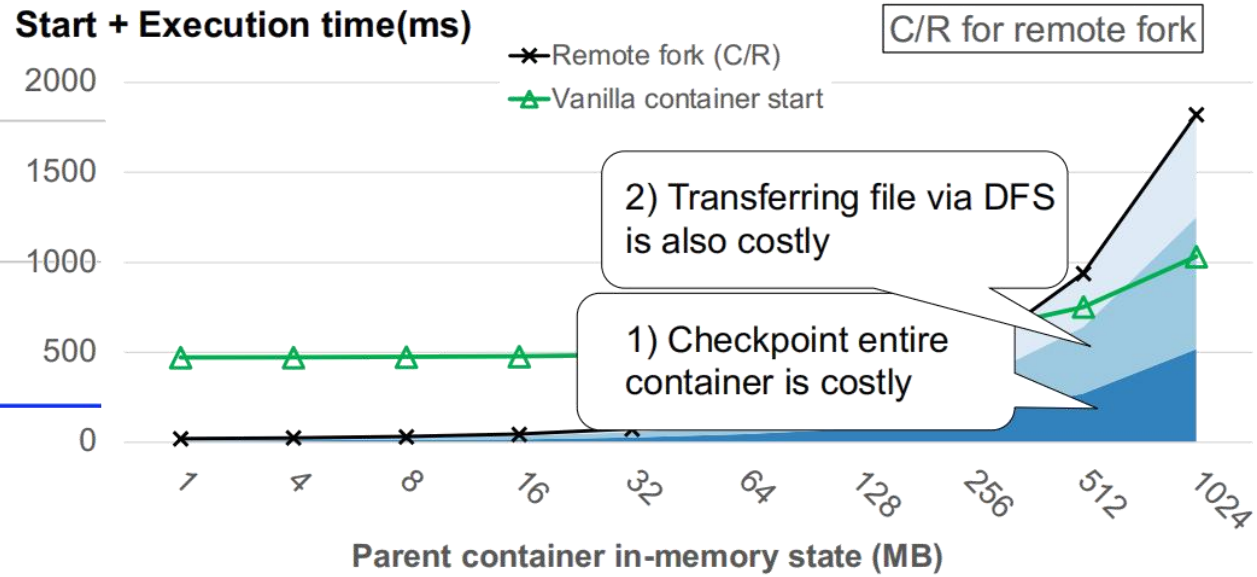
- 远程分叉（remote fork)是没有已配置的并发的原语
  - 观察：一个父节点就足以跨机器启动container
  - 将fork泛化到远程，从而在集群中不启用已配置的并发



# 如何有效地实现远程分叉?

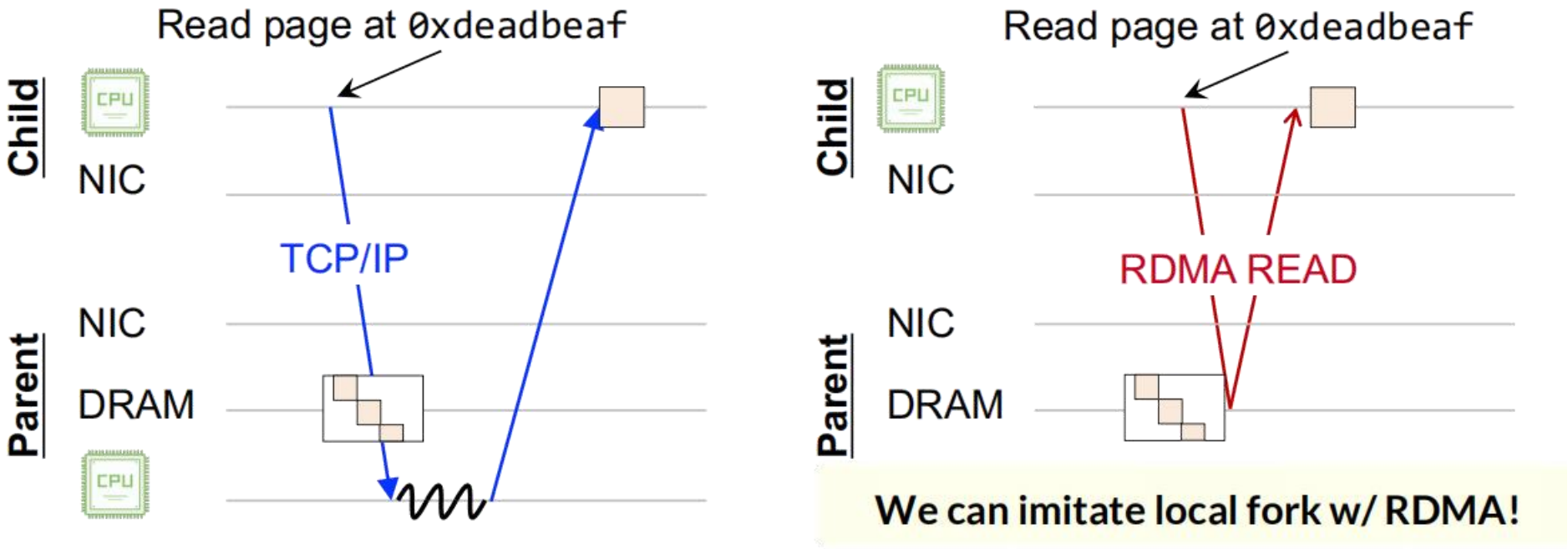
- 当前的解决方案——检查点和恢复（CRIU）的效率不够
  - checkpoint：停止并将内存转储到一个文件中
  - restore：根据文件重建VM，并恢复进程



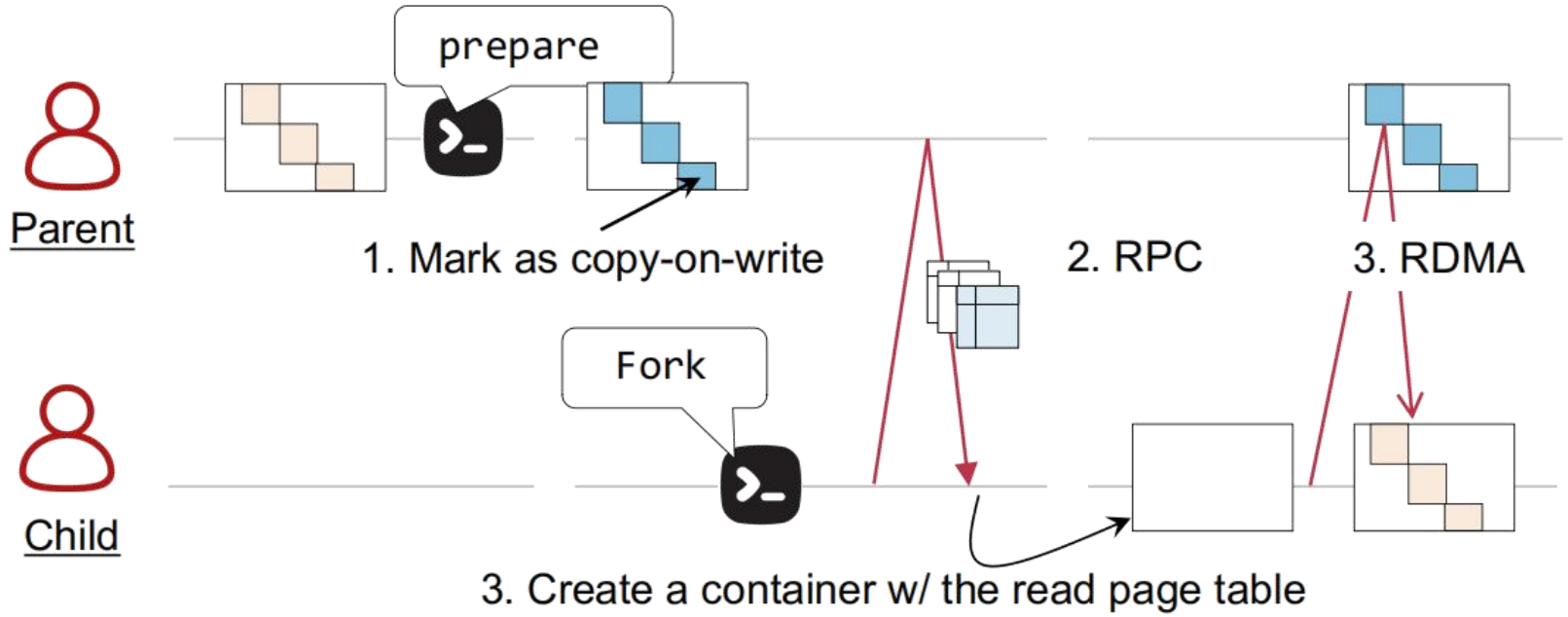
# 一个契机：远程直接内存存取（RDMA）

- 当前已有的远程分叉并不是为RDMA而设计的
- 一个快速的数据中心网络功能，允许直接的远程内存访问
  - 高带宽（400Gbps）和低延迟（600ns）
  - CPU旁路：内存读写被卸载到NIC硬件



# MITOSIS与RDMA共同设计远程分叉

- 在fork上，首先使用基于RDMA的RPC将页表读取给子节点
  - 由于网络放大，单侧RDMA在这一步中效率低下
- 之后，子节点以RDMA可访问的方式（按需）检索内存页面



# MITOSIS与RDMA共同设计远程分叉

- 比未与RDMA共同设计的基本C/R[1]快44-80%
  - C/R实现已经使用了基于RDMA的DFS来恢复状态

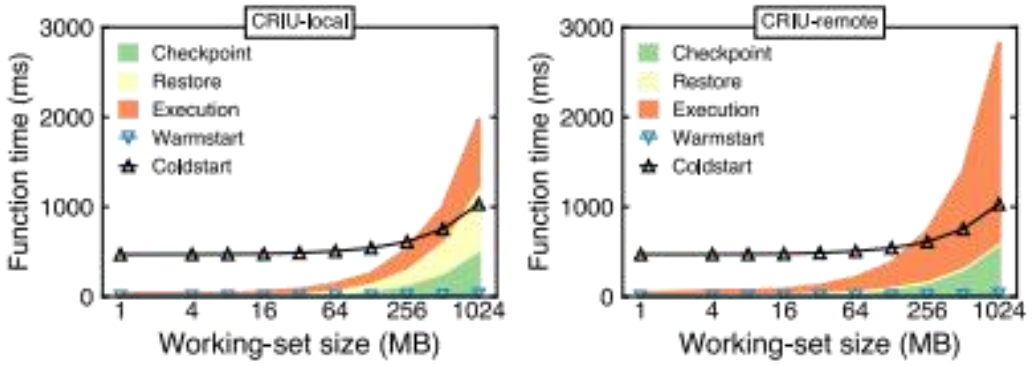
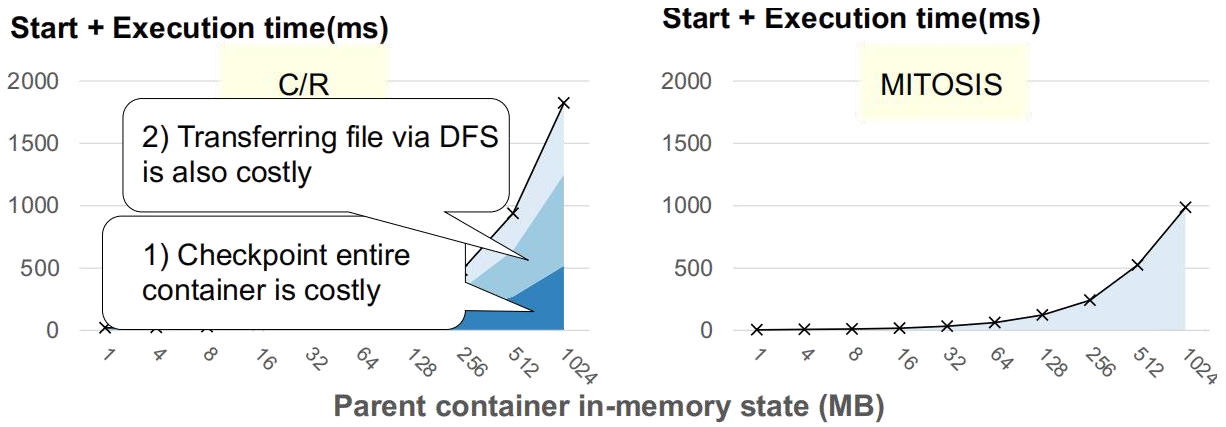



**Figure 4.** *Analysis of using C/R for remote fork.* **Setup**: *CRIU-local: CRIU with a local file system (e.g., tmpfs), which uses RDMA to transfer files between machines. CRIU-remote: CRIU with an RDMA-accelerated distributed file system (e.g., Ceph [89]).*




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

# MITOSIS与RDMA共同设计远程分叉

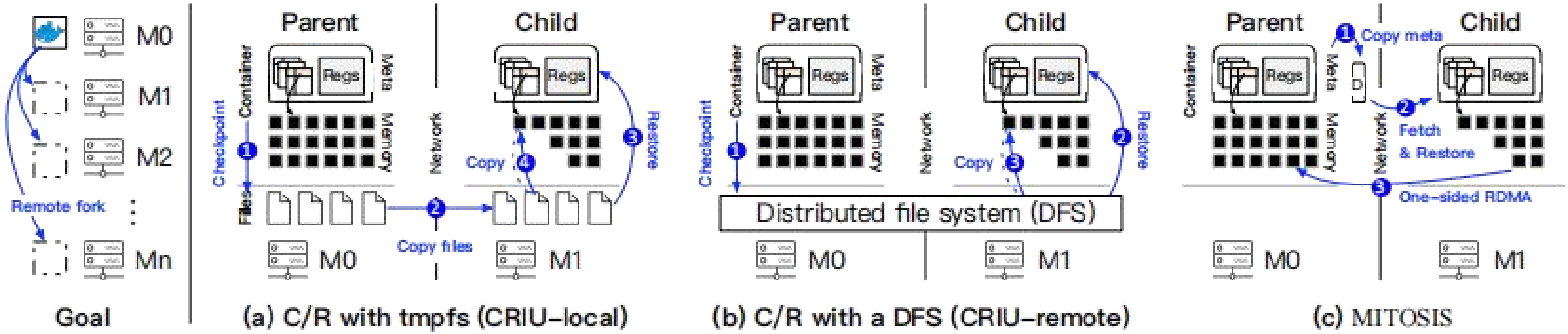



**Figure 5.** *An overview of different approaches to achieve ultra-fast remote fork, including (a) C/R with a local filesystem (e.g., tmpfs), (b) C/R with a fast distributed filesystem (e.g., Ceph [5]), and (c)* MITOSIS.

# MITOSIS与RDMA共同设计远程分叉

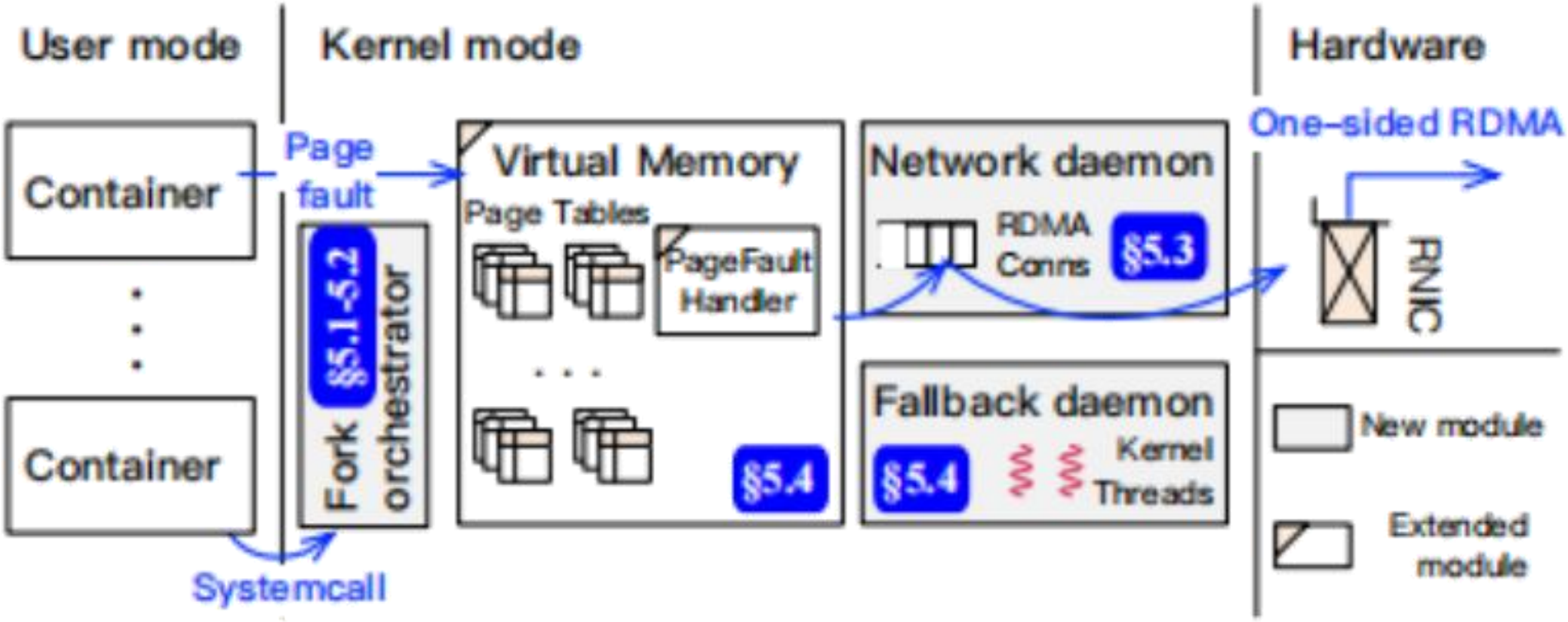
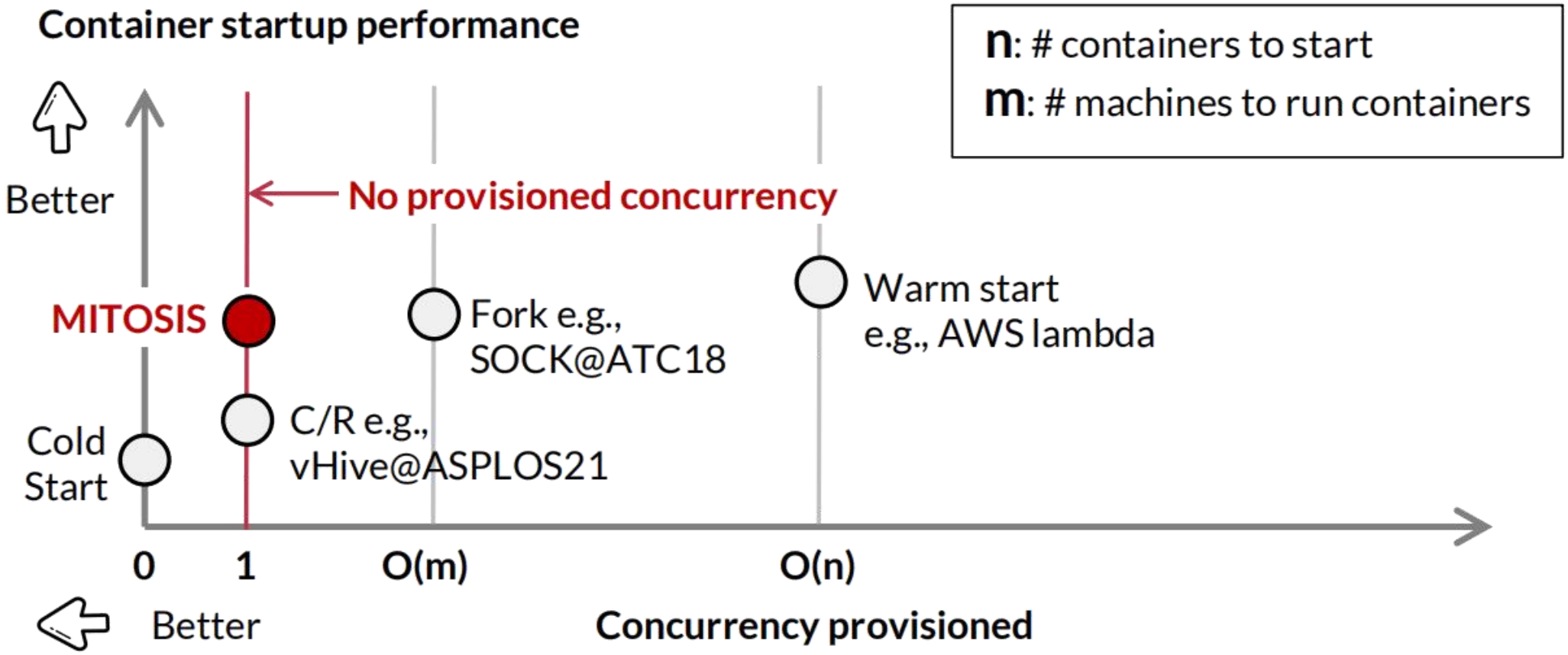



**Figure 6.** *The* MITOSIS *architecture.*

# MITOSIS对比前沿技术



# MITOSIS最佳应用：无服务计算

- 构建云应用程序的新范例
  - 用户将应用程序作为函数上传
  - 每个函数都在一个container中执行，便于部署

- MITOSIS对无服务器计算的两个关键属性助力：
  - container快速启动的资源高效的自动缩放
  - 在无服务器函数之间的快速状态传输——去序列化！


AWS Lambda


Microsoft Azure

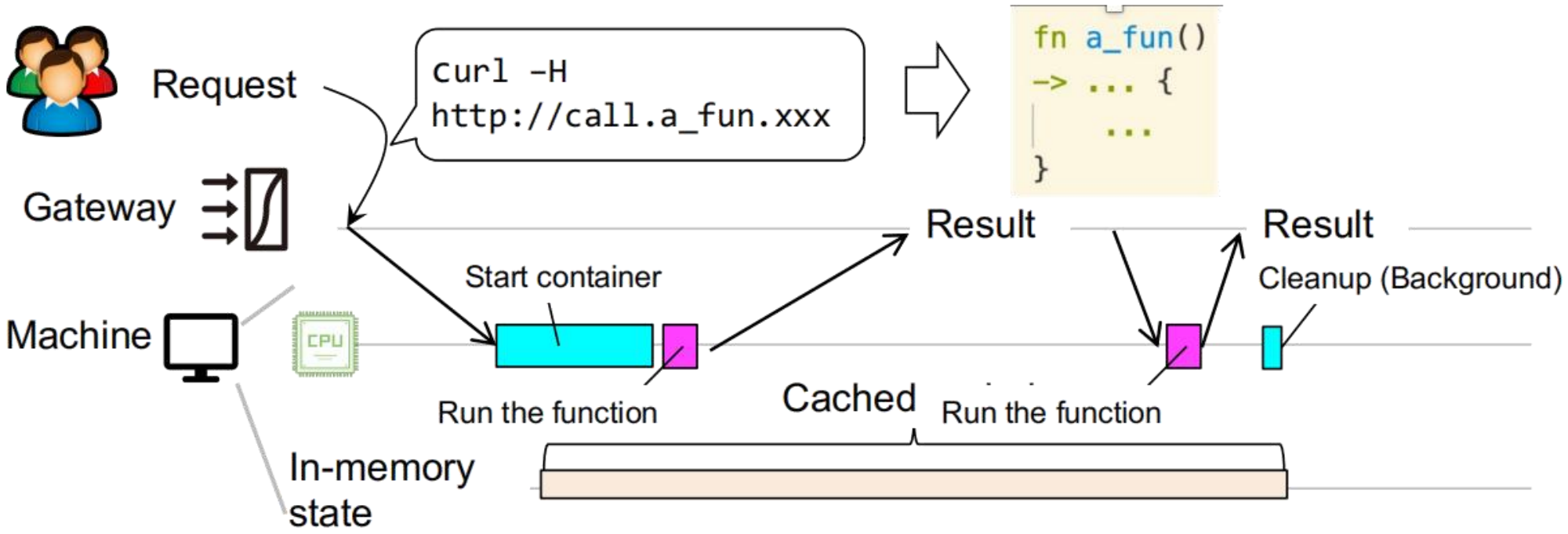

Google Functions


Huawei cloud functions

Opensoruce platforms

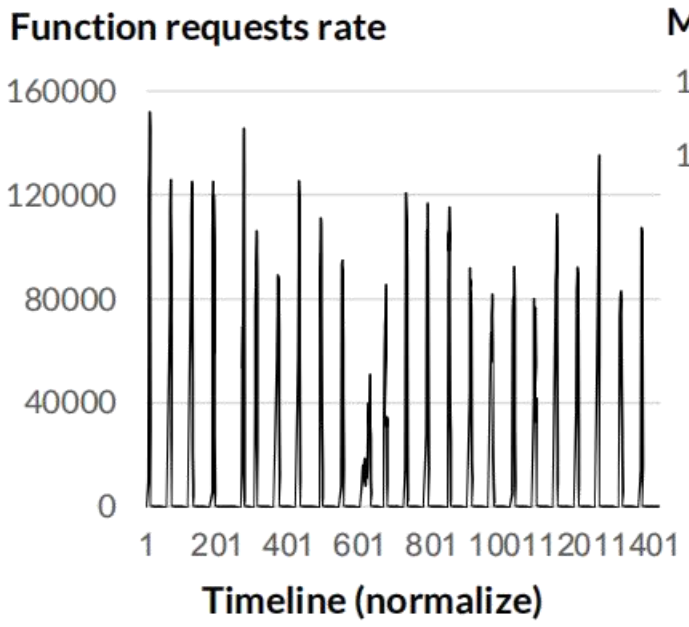
# 资源高效的自动缩放

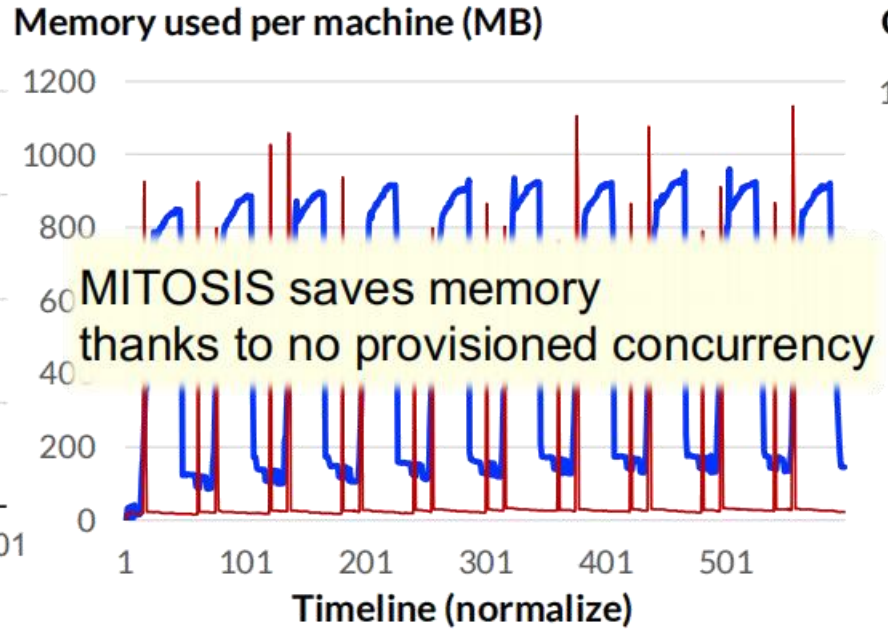
- 每个无服务器函数调用都将启动一个新的容器
  - 该container使用过后可以缓存较短的时间（例如，30秒），以防止冷启动

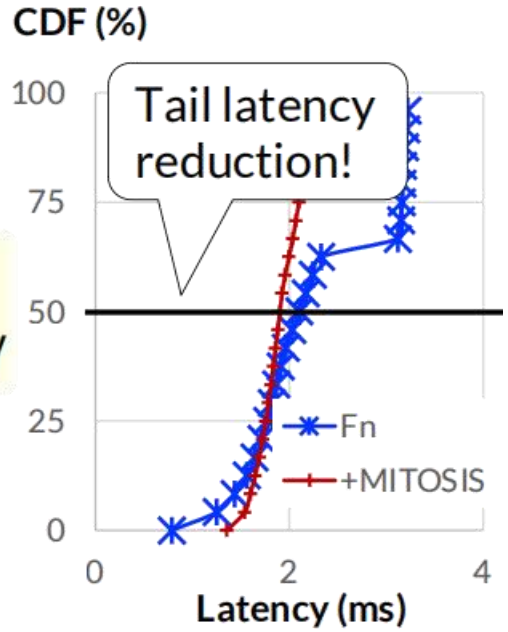


# MITOSIS以更高效的资源的方式处理负载峰值

- 工作负载：来自Azure函数[1]的跟踪（实例#660323）
  - 并发功能以负荷峰值方式调用
  - 设置：Fn，一个拥有24台机器的本地集群；功能：图像处理



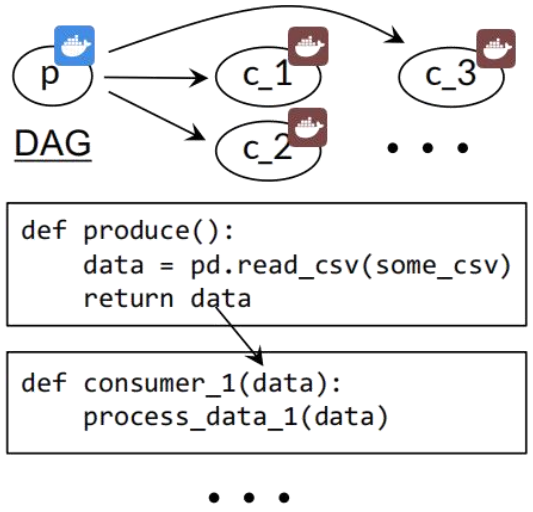



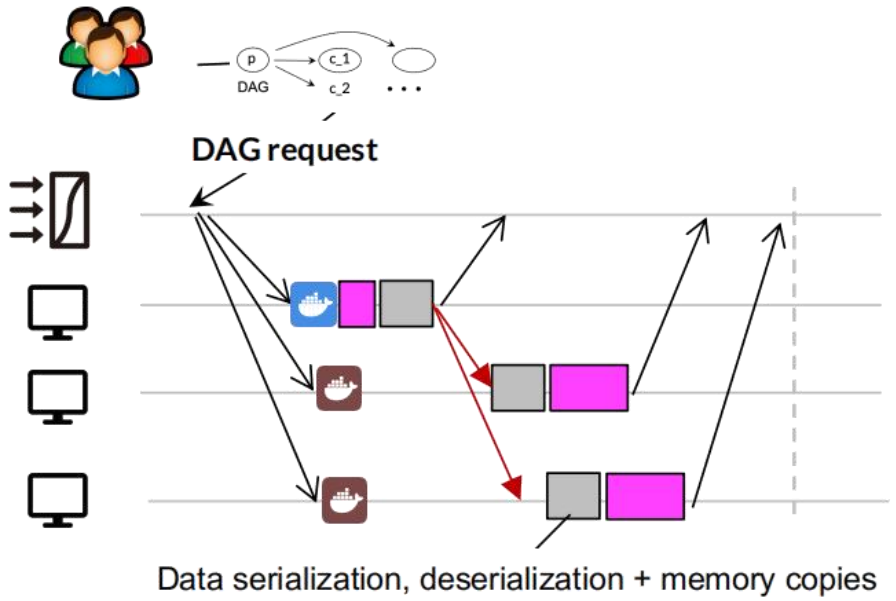



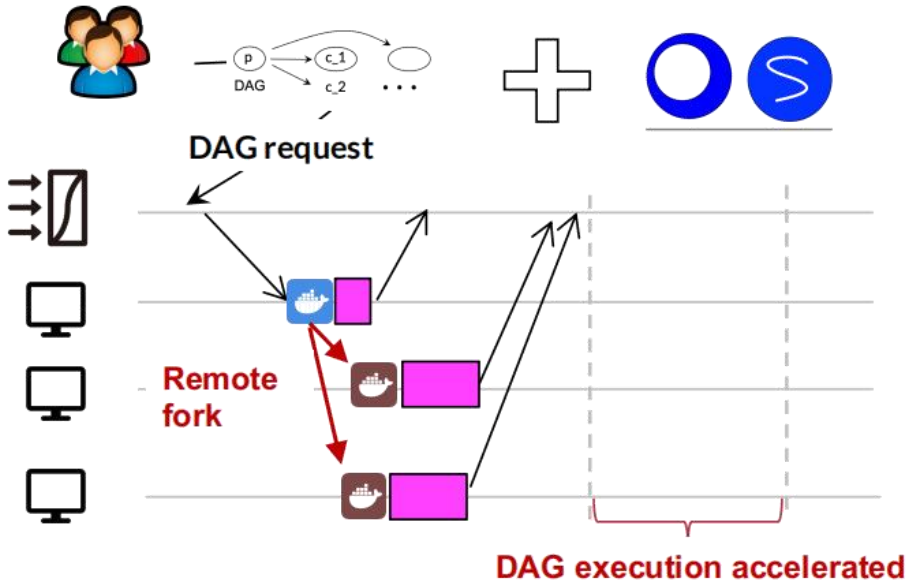

- [1] Serverless in the wild: Characterizing and optimizing the serverless workload at a large cloud provider. ATC’ 20

# 加速函数之间的状态传输

- 无服务器函数可以同时组成多个函数
  - 这些函数通常被组织成一个DAG（直接无环图）
  - 问题：由于（去）序列化+内存拷贝，传输状态代价高昂
- 远程分叉可以完全解决（去）序列化+内存拷贝的成本
  - 该数据已在父节点内存中预先实例化
  - 在远程分叉的帮助下直接继承

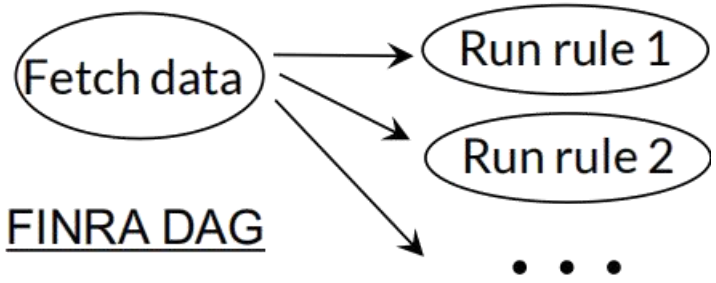


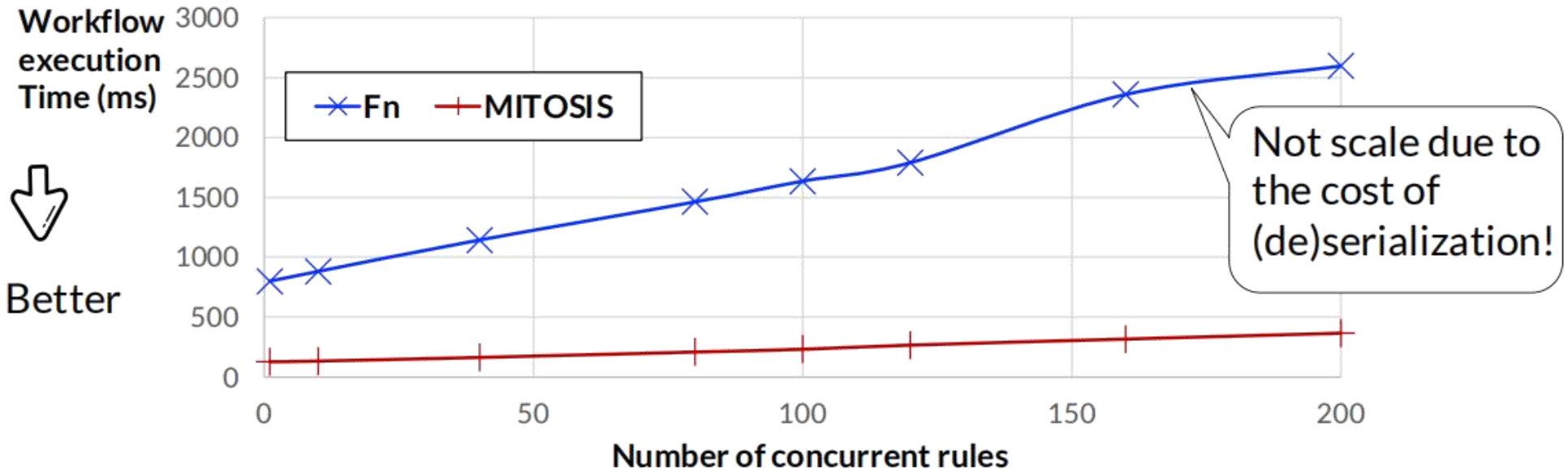



# MITOSIS加速以应对状态传输的高成本

- 工作负载：FINRA——一个真实世界的无服务器应用程序
  - 与无服务器函数并发地验证交易
  - 设置：Fn，baseline采用pickle以（去）序列化



# 性能总结

**Table 1:** *A comparison of startup techniques for autoscaling $n$ concurrent invocations of one function to $m$ machines.* Local *means the resources for the startup are provisioned on the function execution machine. The function is a simple python program that prints 'hello world'.*

| | Coldstart [9, 119] | Caching [63, 123, 94, 102, 122] | Fork [37, 17, 36] | Checkpoint/Restore [120, 37, 117, 20] | Remote fork MITOSIS |
|---|---|---|---|---|---|
| **Local startup performance** | Very slow ($100\,ms$) | Very fast ($< 1\,ms$) | Fast ($1\,ms$) | Medium ($5\,ms$) | Fast ($1\,ms$) |
| **Remote startup performance** | Very slow ($1,000\,ms$) | N/A | N/A | Slow ($24\,ms$) | Fast ($3\,ms$) |
| **Overall resource provisioning** | $O(1)$ | $O(n)$ | $O(m)$ | $O(1)$ | $O(1)$ |

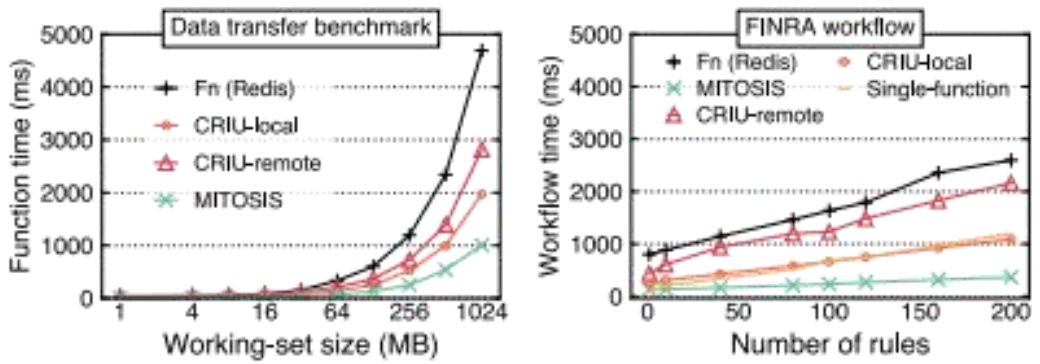



**Figure 20.** *(a) The state-transfer performance between two functions and (b) performance of FINRA.*

# 感谢倾听，敬请指正