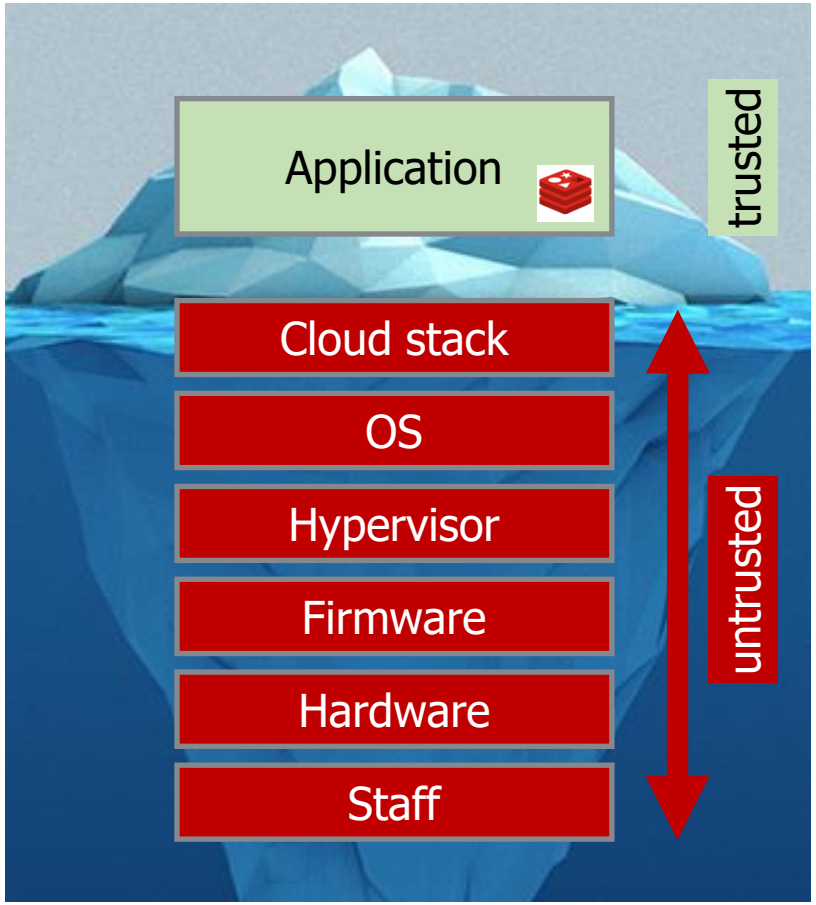
Imperial College London

Rethinking service isolation to reduce the cloud tax

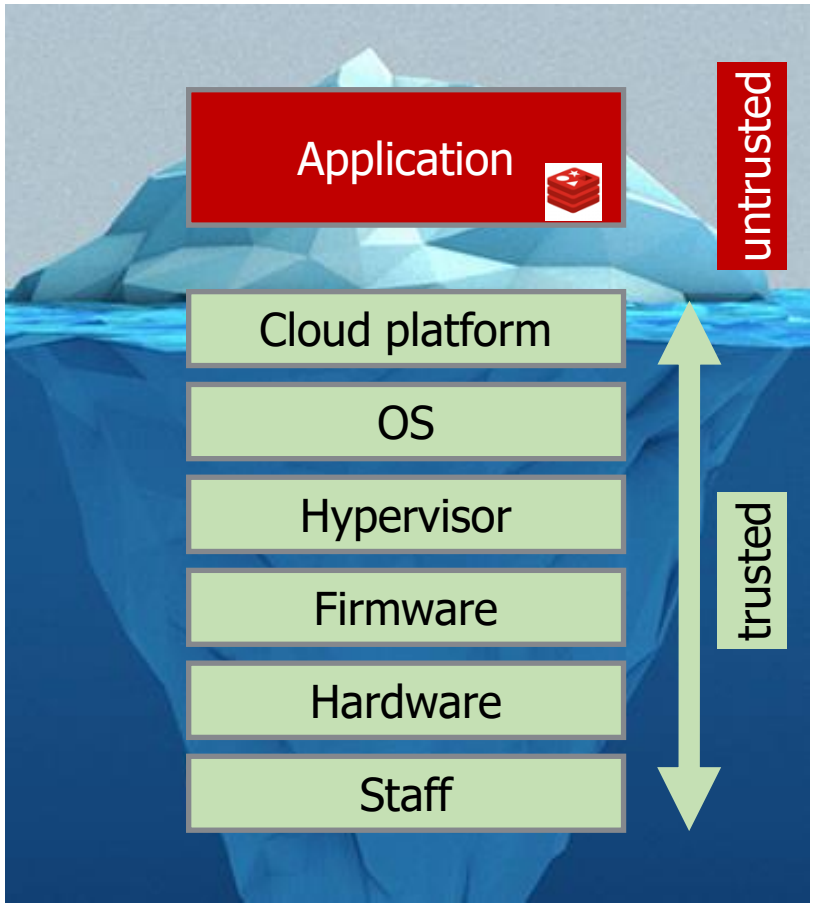
Lluís Vilanova
Imperial College London
<vilanova@imperial.ac.uk>

May 2023 - Huawei SSRC

Our Goal: Trustworthy and Efficient Clouds



Cloud user perspective



Cloud provider perspective

Security Challenges in Cloud Environments

Cloud environments have a large **trusted computing base (TCB)**

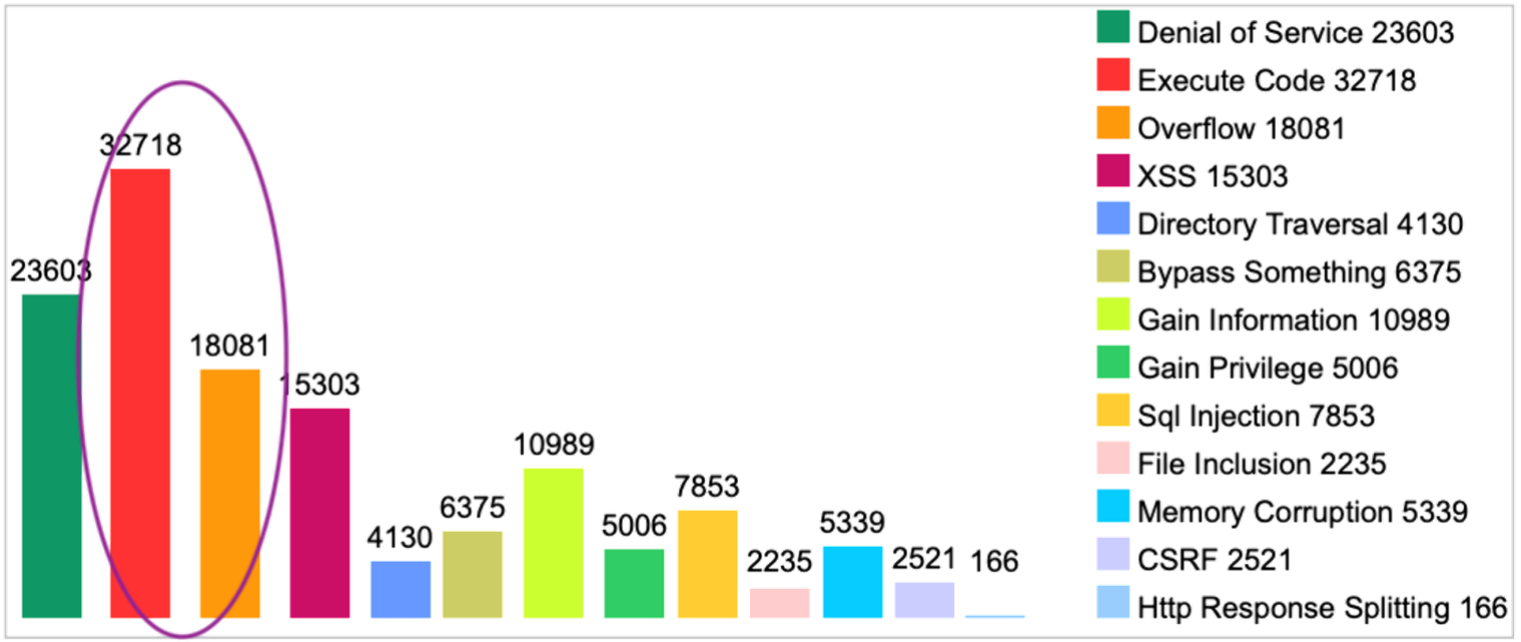
- Linux: 20 million
- KVM: 13 million
- OpenStack: 2 million

Large TCB = many **security vulnerabilities**

- Xen: 184 (2012-16)
- Linux: 721 (2012-16)

Many **attack vectors**

- Control-flow hijacking
- Code injection
- Return-oriented programming



☛ Need a fundamentally different approach for engineering complex software stacks

The Cloud Tax: A Systemic Inefficiency Problem

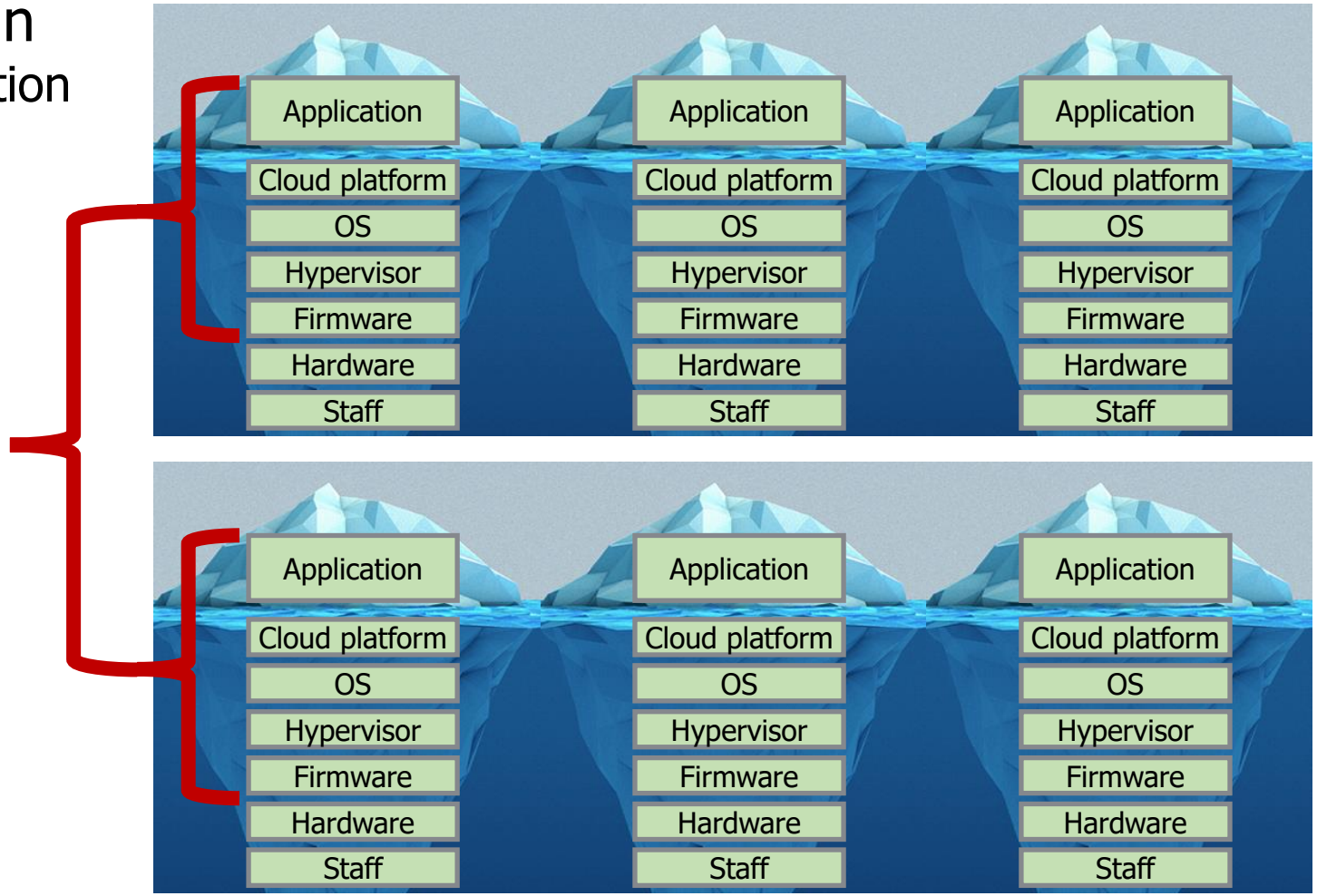
Cloud stack inefficient by design
- Poor CPU & memory consolidation
- Increased TCO

Overheads across resources:

Memory bandwidth
- Runtime data copies

CPU cycles
- Runtime data copies
- Crossing multiple layers

Memory utilization
- Components are replicated
  - Same apps, libraries, OS...

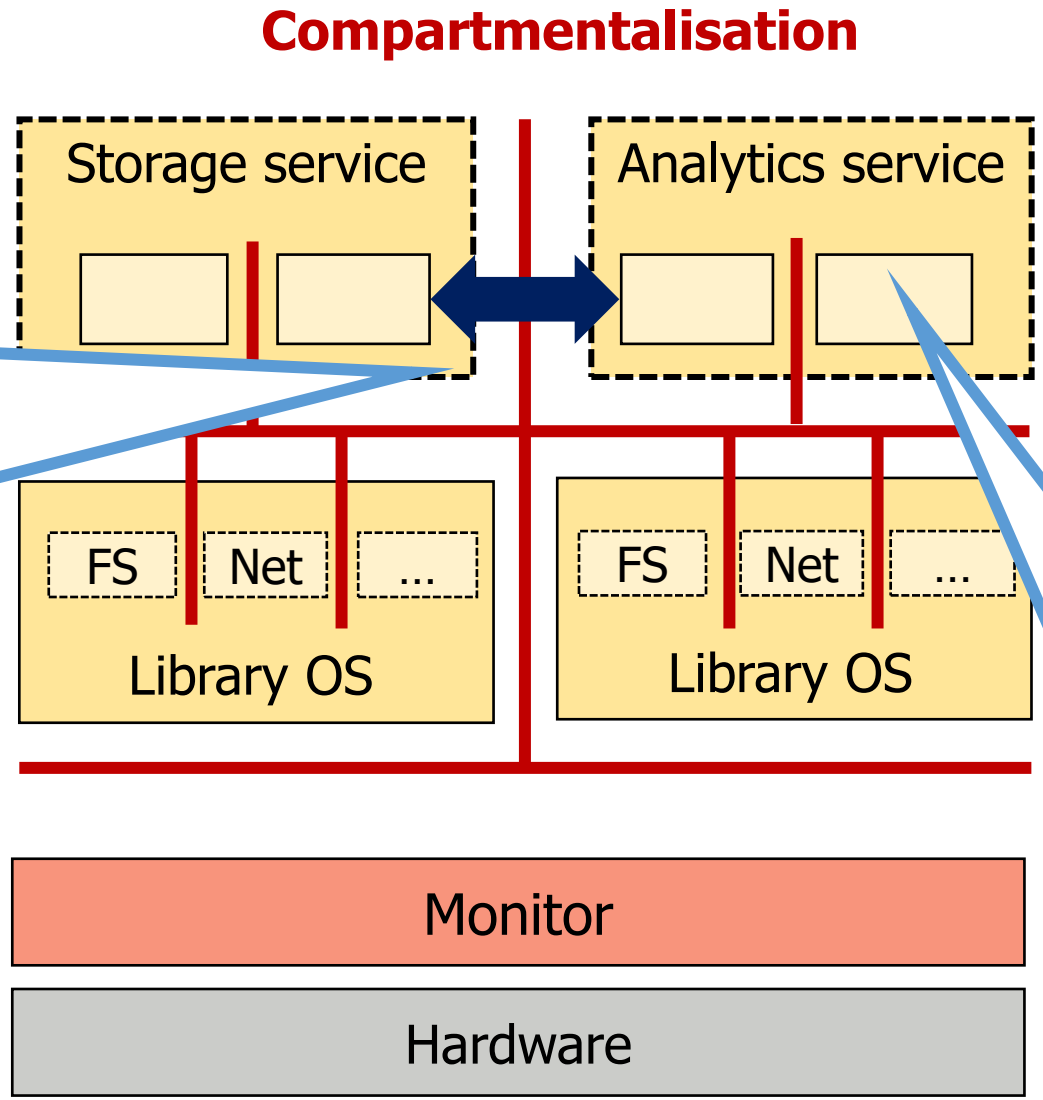


Roadmap: Slashing the Cloud Tax

1. CAP-VMs:
Capability-Based Isolation and Sharing in the Cloud

[USENIX OSDI 2022]

Improve **data sharing** without weakening security



Share code by design without weakening security

2. Single-Copy Objects (SCO):
Reducing the Memory Footprint in the Cloud

[USENIX OSDI 2023]

Memory Capabilities: A Powerful Primitive for Security and Efficiency

Isolation puts privileged memory management unit (MMU) at the center

MMU always involved in **sharing**, e.g. inter-process communication (IPC)

- Set up shared buffers
- Execute syscalls/hypercalls

MMU shares data at **page granularity**

Incorrect sharing may expose extra data

Insufficient sharing wastes compute and bandwidth (memory copies)

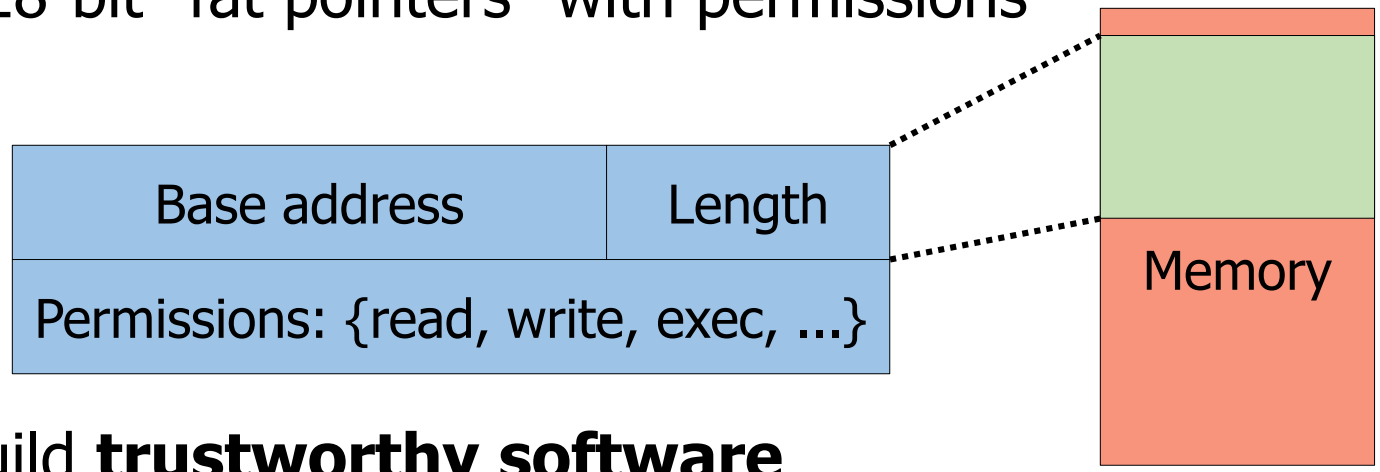
What if the hardware provided an efficient mechanisms to share arbitrary-sized memory regions between otherwise isolated entities?

CHERI: Capability Hardware Enhanced RISC Instructions

Memory Capabilities with CHERI

Capabilities: Unforgeable 128-bit "fat pointers" with permissions

- Identifies memory region (integer, array, …)
- Isolation at byte-granularity
- Protected by the hardware



Capabilities can be used to build **trustworthy software**

- Can be passed via registers and memory
- New capabilities can only be derived from existing capabilities
- Limited dependency on privileged layers (OS kernel, hypervisor)

Mix of existing and **cap-aware instructions**

- Existing code uses implicit capabilities -> **hybrid code**
- New code uses explicit capabilities -> **pure-cap code**
  - Typically means changing memory allocators, no pointer arithmetic etc.

Real-World CHERI Hardware Available

Morello: Armv8-A architecture with CHERI support

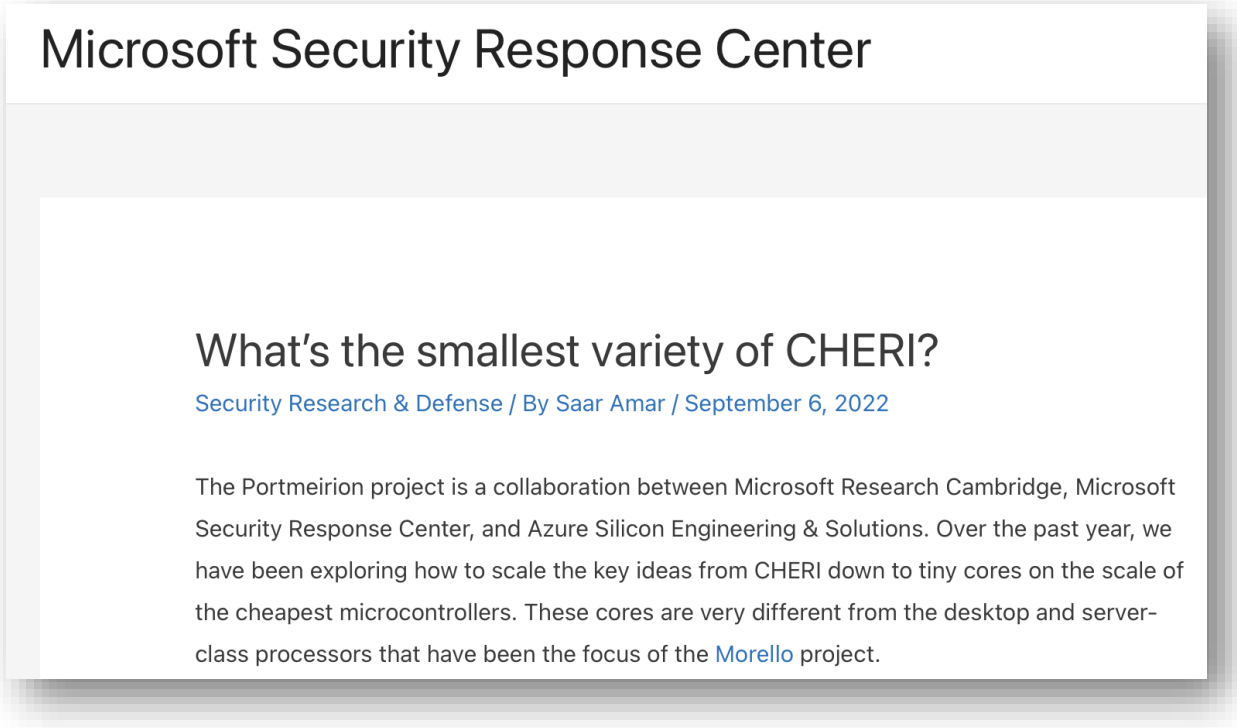
- Available since Spring 2022

Supports existing software stacks

- FreeBSD, Linux port (ongoing), …

Part of UK's cybersecurity and semiconductor strategy

Memory Capabilities and RISC-V



source: https://msrc-blog.microsoft.com/2022/09/06/whats-the-smallest-variety-of-cheri/

Microsoft CHERI implementation for embedded RISC-V

- 32-bit RISC-V32E micro-controller
- Based on lowRISC Ibex core
- 64-bit capabilities in all registers

New RISC-V CHERI working group

- Led by Google
- Focus on applications and embedded

☛ Waiting for CHERI RISC-V hardware

(1) CAP-VMs: Capability-Based Isolation and Sharing in the Cloud

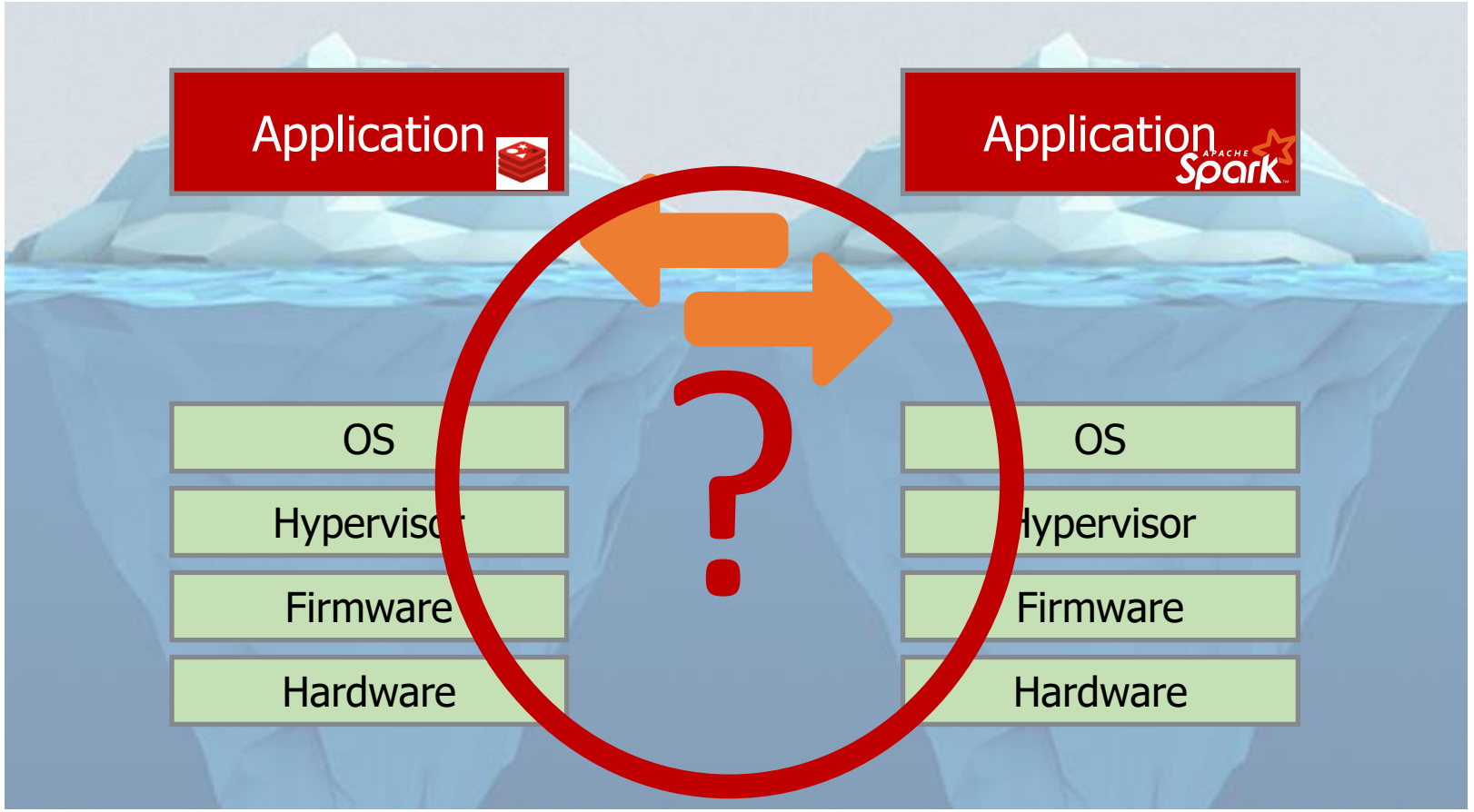
[USENIX OSDI 2022]

Isolation and Communication in the Cloud

Hyperscaler's perspective:
- Reduce cost through consolidation

Tenant's perspective:
- Data sharing across apps/services

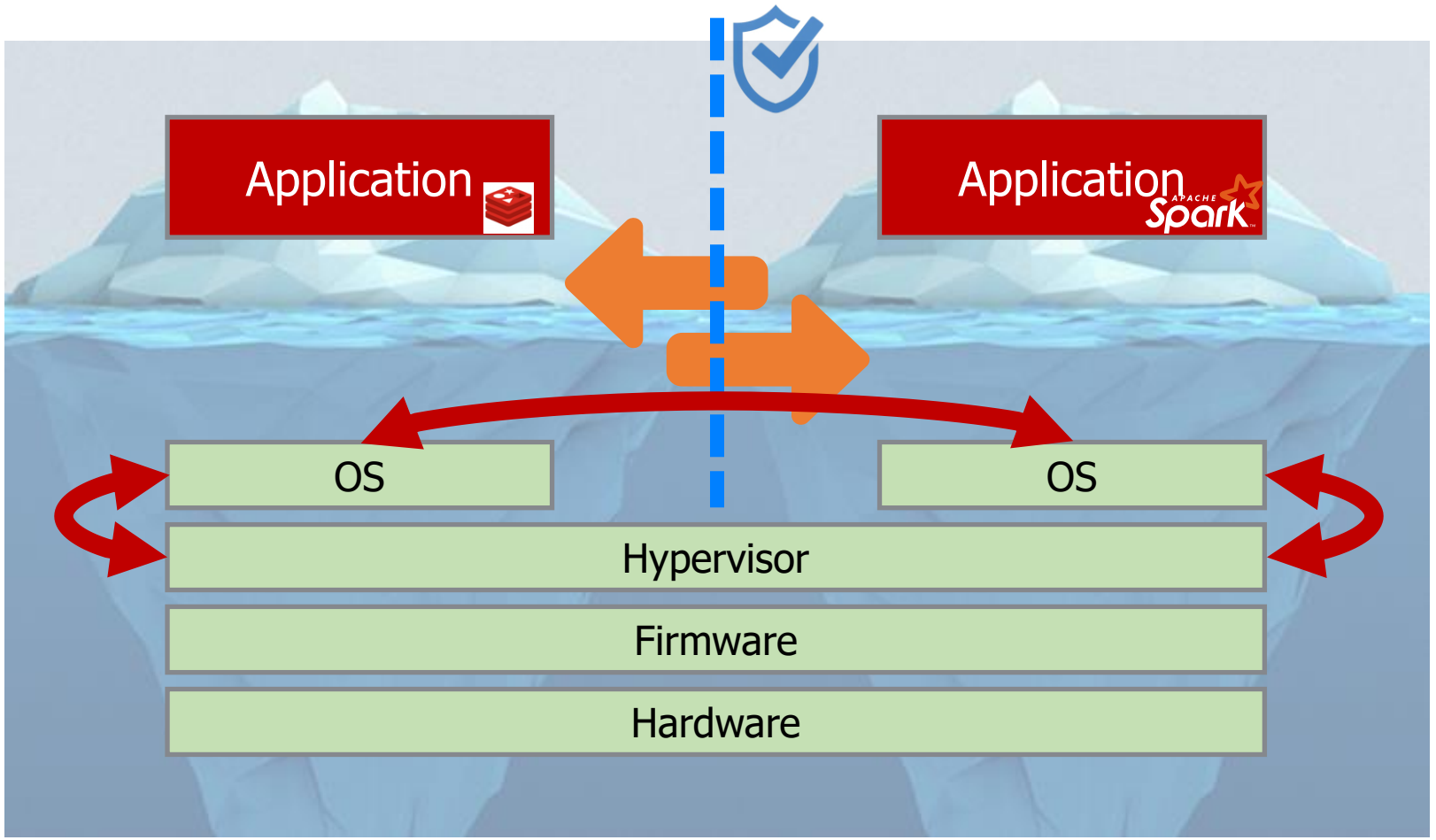
Fundamental tension to handle isolation and communication



Isolation and Communication: VMs

+ Strong isolation
+ Small TCB (hypervisor)

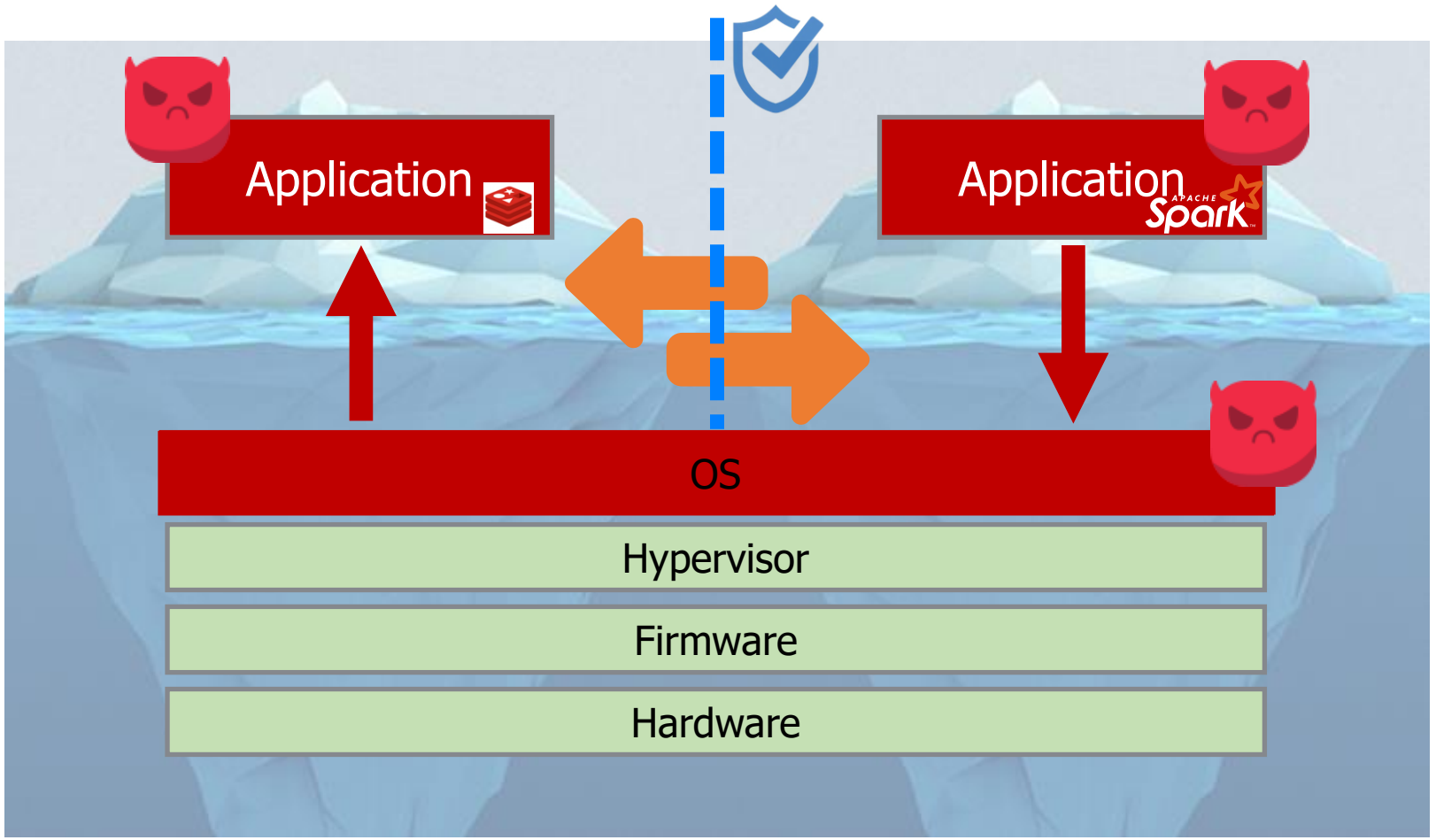
- Expensive communication (TCP/IP)
- Expensive transitions (hypercalls)



Isolation and Communication: Containers

+ Lightweight isolation
+ Efficient communication (IPC)

- Large TCB (host OS)



VMs & Containers: Isolation ←→ Communication

☛ **Challenge: Efficient communication,**
Strong isolation with small TCB

Requirements at odds with current designs (containers vs. VMs)

Tension between compatibility and whole-system redesign

☛ **Goal: Efficiency of container communications,**
Strong isolation and small TCB of VMs,
Compatible with existing codebases

Use CHERI to isolate, while removing dependency on host OS

A Cloud Stack with Hardware Capabilities

How to design a new **software stack** for cloud environments that uses **hardware memory capabilities**?

Challenges of a capability-based cloud stack:

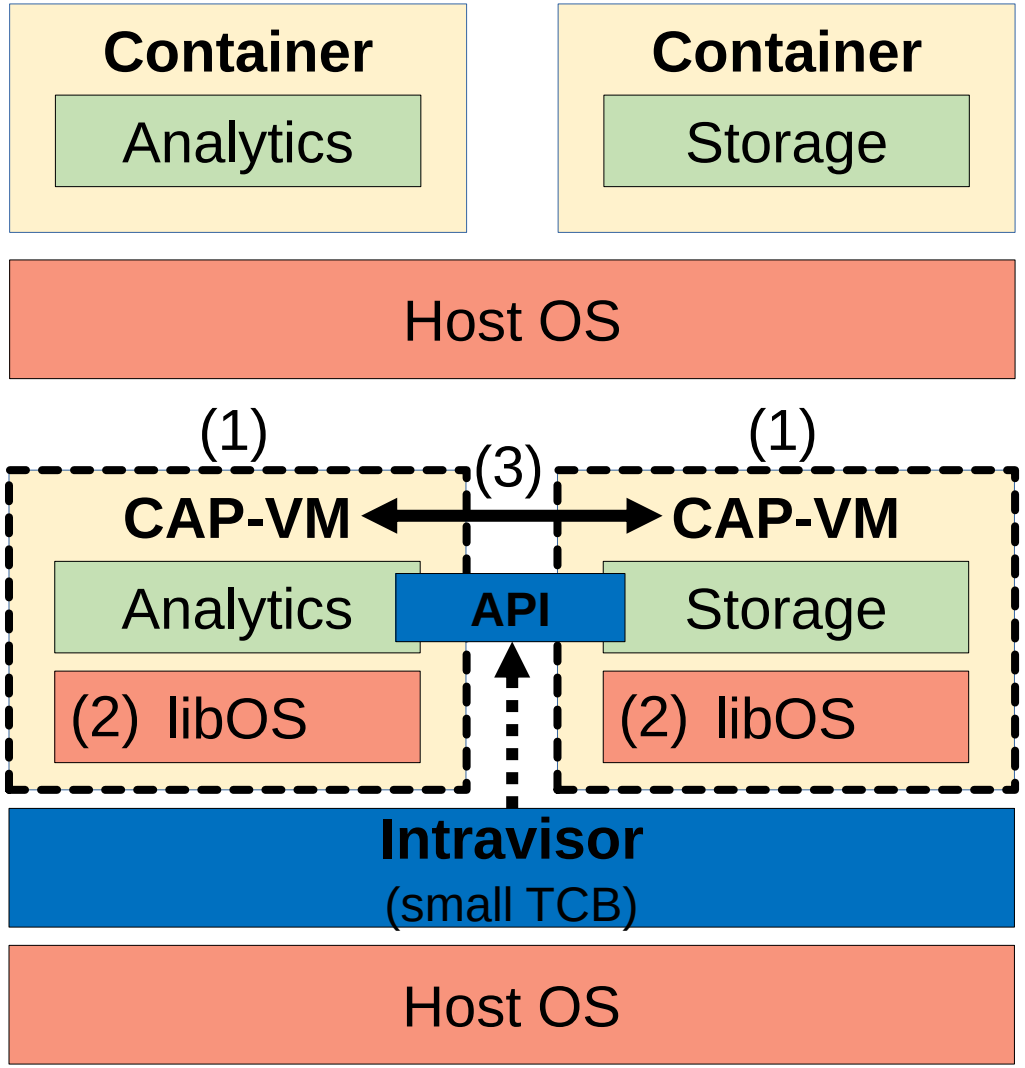
1. **Support capability-unaware software**
2. **Compatible with existing OS APIs**
3. **Provide small-TCB shared stack in the host**
4. **Enable efficient capability-based IPC interfaces**

CAP-VMs: Fusing VMs and Containers

1. **Per-container strong isolation**
   - Shared page table + CHERI RISC-V default capabilities (similar to i386 default segments)
   - Transparent to applications

2. **Per-container library OS**
   - LKL (vanilla Linux as a library)
   - Per-container OS, trivially isolated

3. **Communication API**
   - Asynchronous buffer, file, and call APIs (capability-unaware API, capability-accelerated)
   - Controlled by small TCB (intravisor)



Experimental Evaluation: CAP-VMs

Experiments on two platforms:

CHERI RISC-V 64-bit

- Cycle-accurate single-core simulator
- FPGA simulator running on AWS-F1 instances

SiFive HiFive RISC-V Unmatched

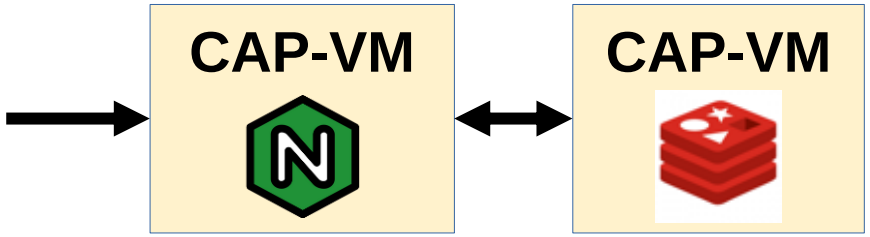
- Quad-core ASIC
- No CHERI support, but same software stack

Source code: https://github.com/lsds/intravisor

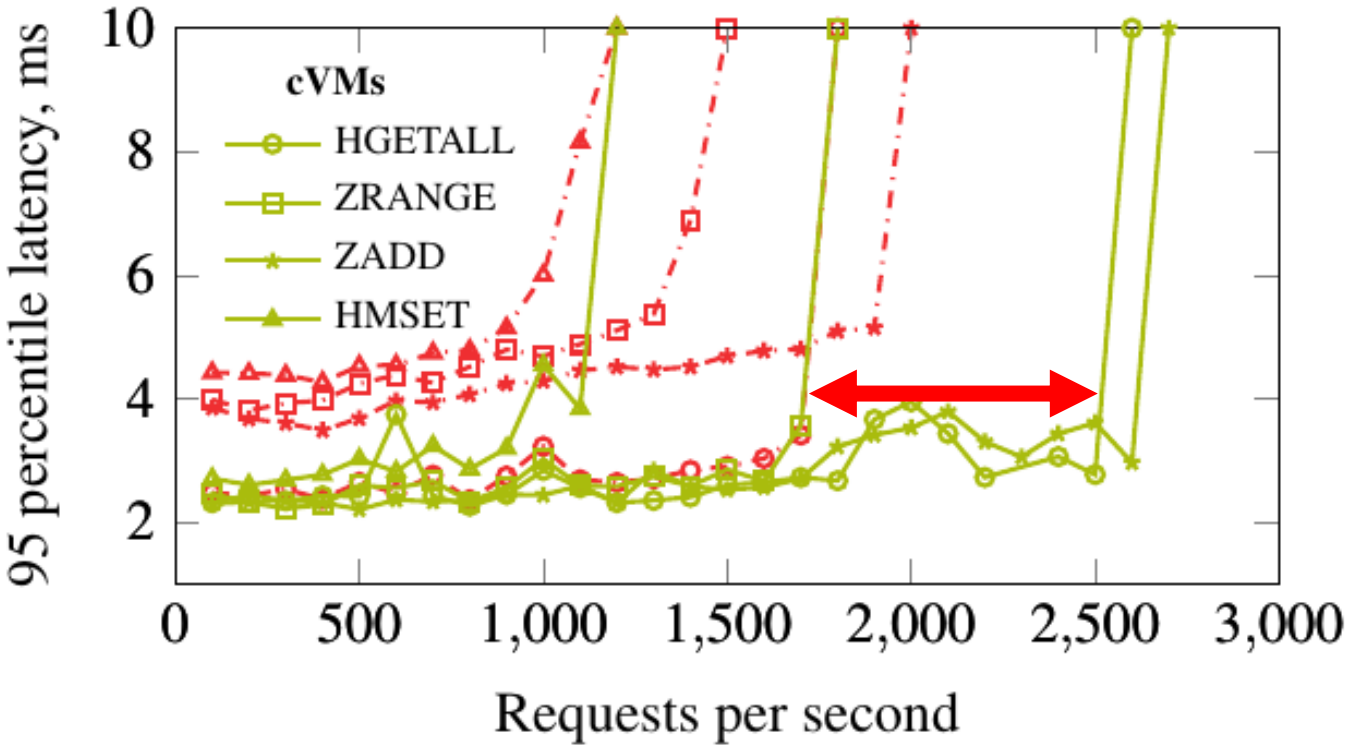
Results: Multi-tier Cloud Service

Is communication efficient?

- YCSB queries on NGINX + Redis



- Docker+TCP/IP vs. CAP-VMs



☛ 1.5x throughput
at 95th percentile latency
+
stronger isolation

(2) Single-Copy Objects (SCOs): Reducing the Cloud's Memory Footprint using Capabilities

[USENIX OSDI 2023]

Duplicate Software Components in the Cloud

☛ Many VMs/containers have **similar memory content**

Virtual machines (VMs) have their own **guest OS kernel**

- Usually the same across VMs

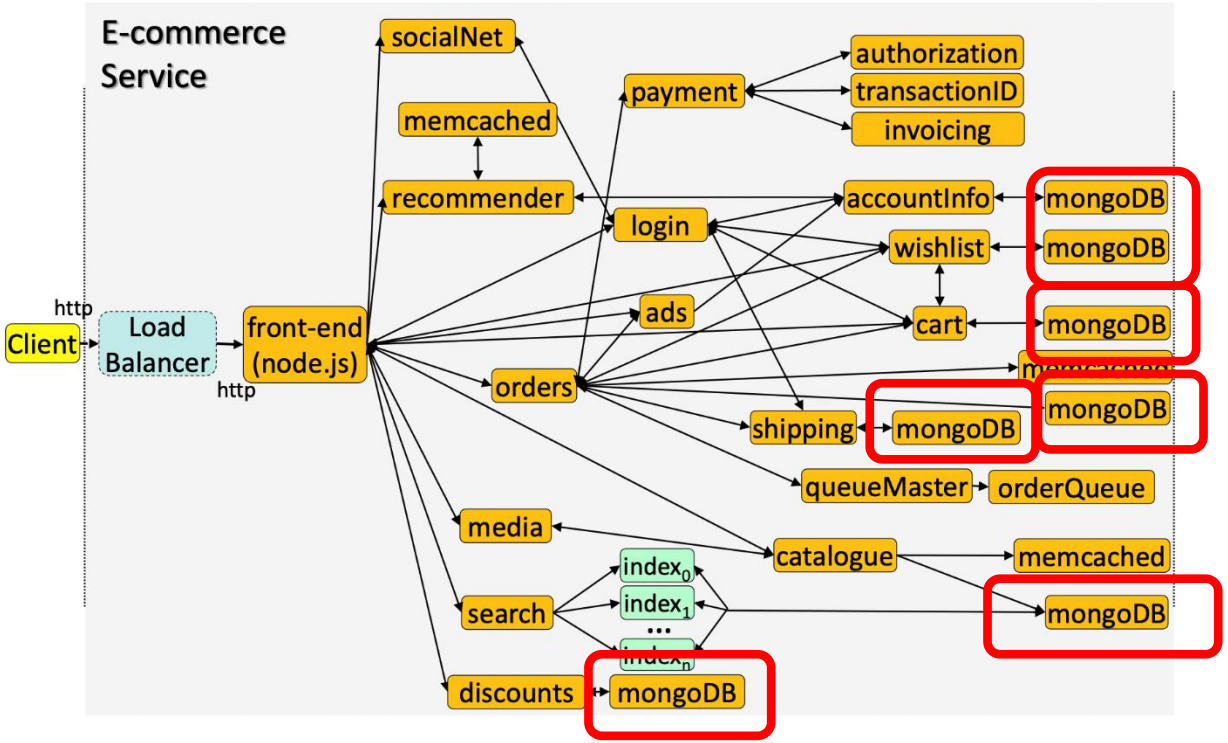
VMs and containers run **similar applications**

- E.g. many users deploy the same NGINX web server

Different applications built on top of **typical frameworks/dependencies**

- E.g. use the same Python runtime
- E.g. many dynamic libraries will be the same (*lib*.so*)

Example: Microservice Architecture



Microservice architecture of a typical e-commerce cloud application (DeathStarBench)

Source: Y. Gan et al., ASPLOS 2019

Each mongoDB instance is a container:

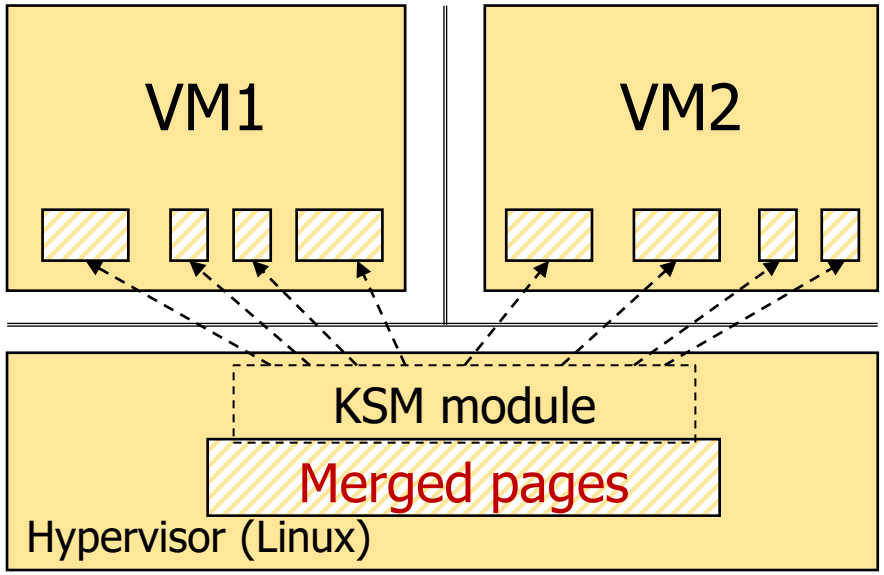
```
148M Nov  5  2021  mongosh
104M Dec 19  2013  mongod
72M Dec 19  2013  mongos
56M Dec 19  2013  mongo
17M Oct 12  2021  mongofiles
17M Oct 12  2021  mongorestore
16M Oct 12  2021  mongodump
16M Oct 12  2021  mongoimport
16M Oct 12  2021  mongoexport
16M Oct 12  2021  mongostat
16M Oct 12  2021  mongotop
14M Oct 12  2021  bsondump
3.4M Oct 19  2020  perl
3.4M Oct 19  2020  perl5.30.0
1.2M Jun 18  2020  bash
1.1M Jan  6  2021  gpg
875K Jan  6  2021  gpgcompose
736K Aug 23  2021  openssl
```

Substantially duplicates **memory usage**: 518 MB out of 640 MB ➔ ~80%

State-of-the-Art: Hypervisor-led Memory Deduplication

Kernel Same-Page Merging (KSM) done by Linux KVM hypervisor

Periodically compares memory pages and removes duplicate pages



KSM Works with VM-level Isolation, But:

Only hosting provider benefits from KSM

- Deduplication decisions and overheads are outside of user control

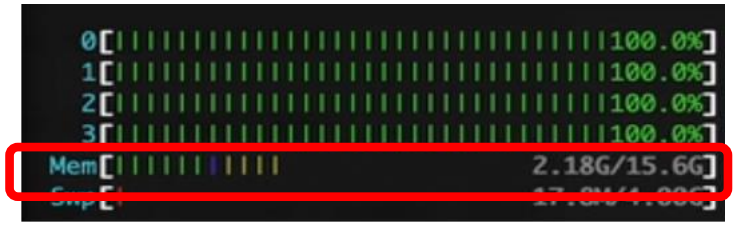
KSM is **probabilistic in nature**

- Deduplication should be predictable and guaranteed

KSM **consumes CPU cycles**

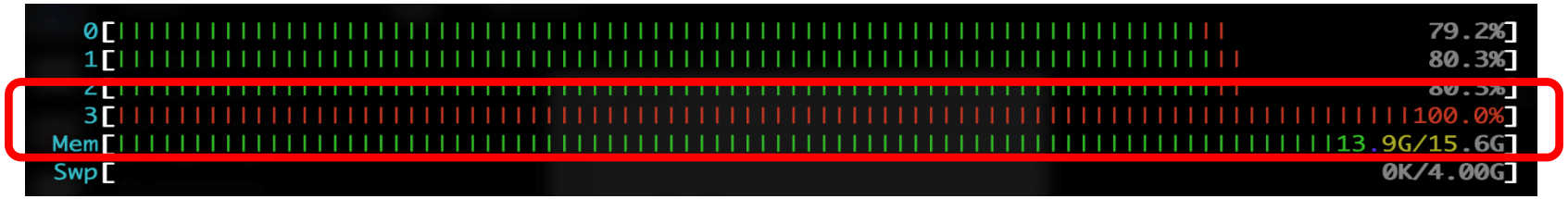
- Deduplication should have minimal overheads

First KSM run:
memory reduces
13 GB ➔ 2.2 GB



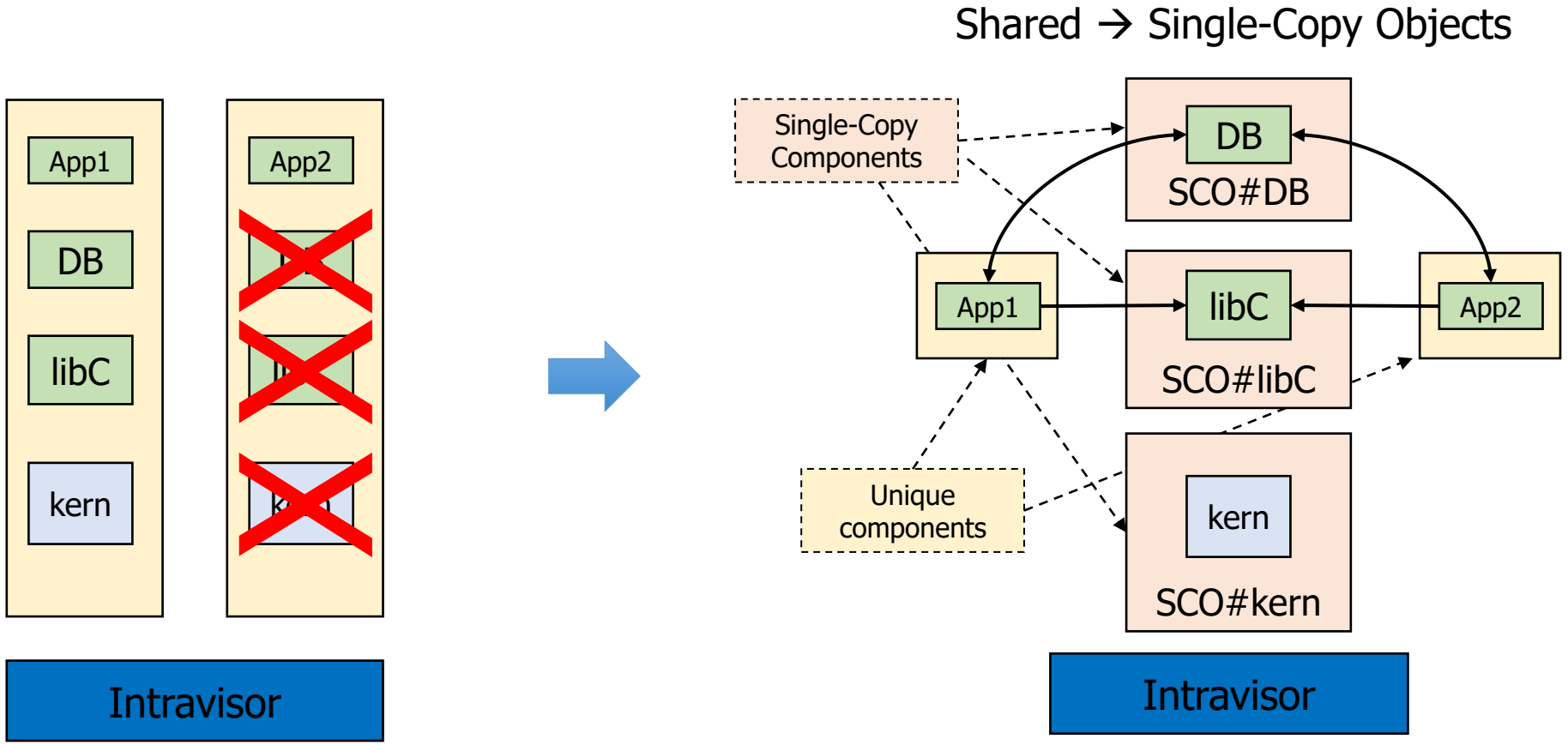
Next KSM run:
memory reduced
13 GB ➔ 11 GB

Active KSM:
100% CPU



Our Approach: Single-Copy Objects (SCO)

Sharing by design -> break application down into reusable components



SCO: Challenges

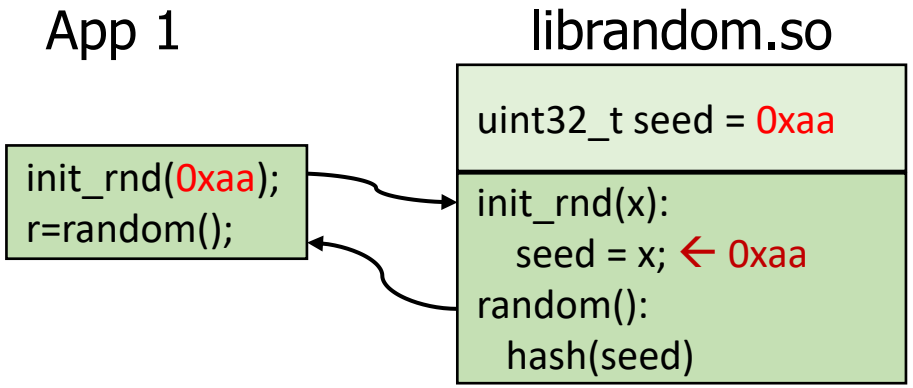
Challenge 1: Code decomposition
- Must support existing code
- Requires lightweight isolation between components
- Needs fast switches between isolated components

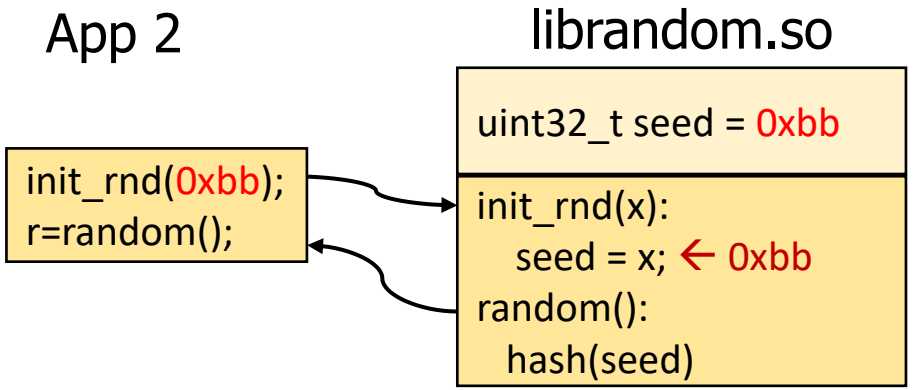
Solved by CAP-VMs & CHERI memory capabilities

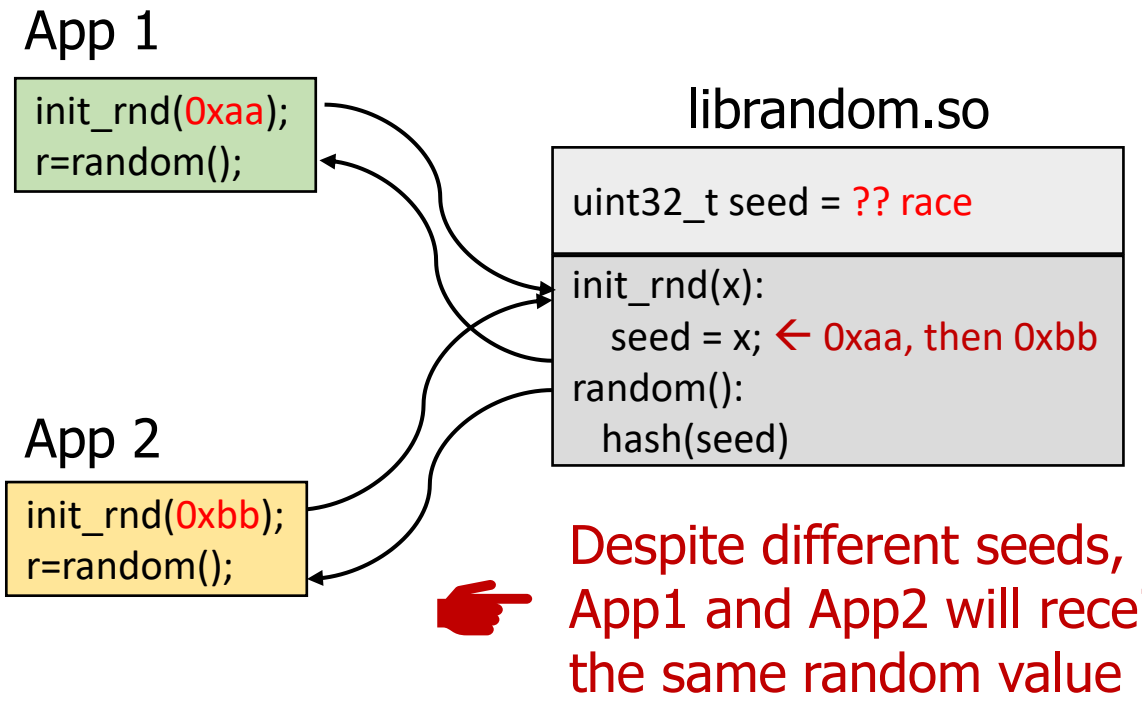
Challenge 2: Appropriate memory sharing
- Not everything is shared within an SCO
- SCOs must access different state according to calling CAP-VM

Challenge 3: Trusted sharing of untrusted objects (see paper)
- Must allow untrusted programs to load and create objects
- Do not rely on trusted code (smaller TCB size)

Challenge 2: Appropriate Memory Sharing







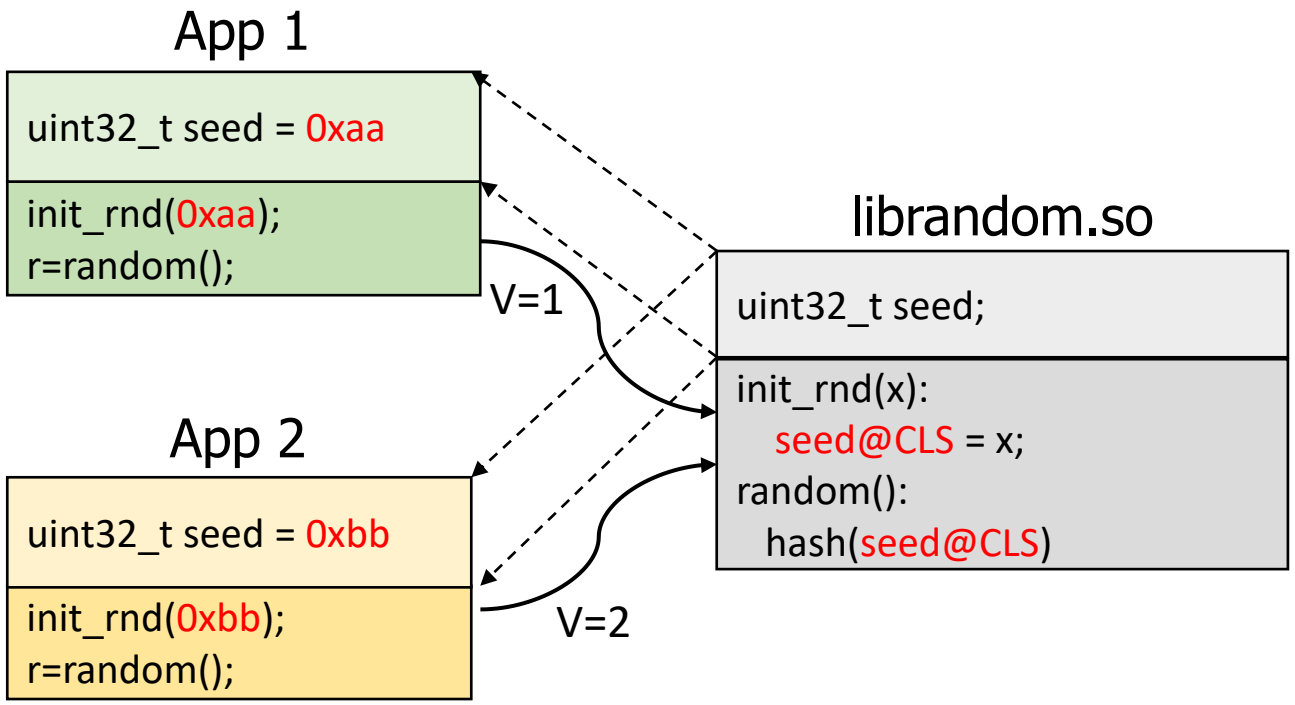
Despite different seeds, App1 and App2 will receive the same random value

Components have state (e.g. variables)

State must not be shared

Naïve SCO deduplication → state corruption

Compartment-Local Storage



SCOs do not store state

Introduce **compartment-local storage (CLS)** to access state within SCO

- Replicated on each CAP-VM (App 1/2)

New linker relocation, using memory capabilities

- Similar to thread-local storage (TLS)
- Implemented as LLVM pass

Experimental Evaluation: SCOs

Question: How efficient are SCOs compared to KSM?

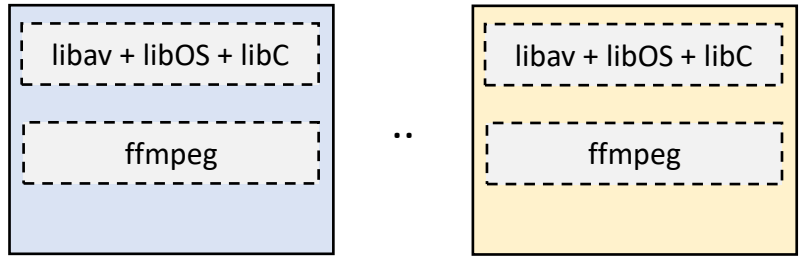
Scenario:

- On-demand **realtime video transcoding micro-service** (lots of sharing potential)
- Limited by either **memory** or **CPU time**
  - Needs low launch and deduplicate latency for new processes
  - Must have low CPU overhead

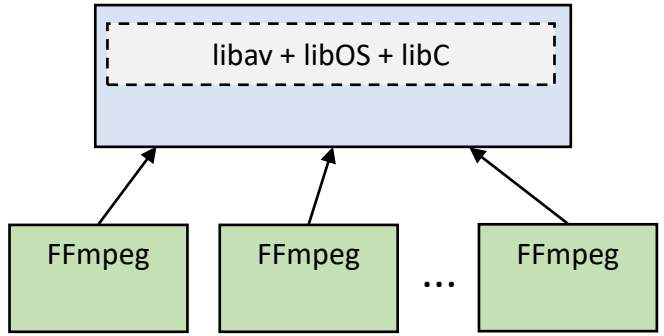
Environment:

- ffmpeg video transcoder + libraries + library OS **(110 MB, 10% SCO-shareable + heap)**
- 16 GB memory, 4 CPU cores

With KSM



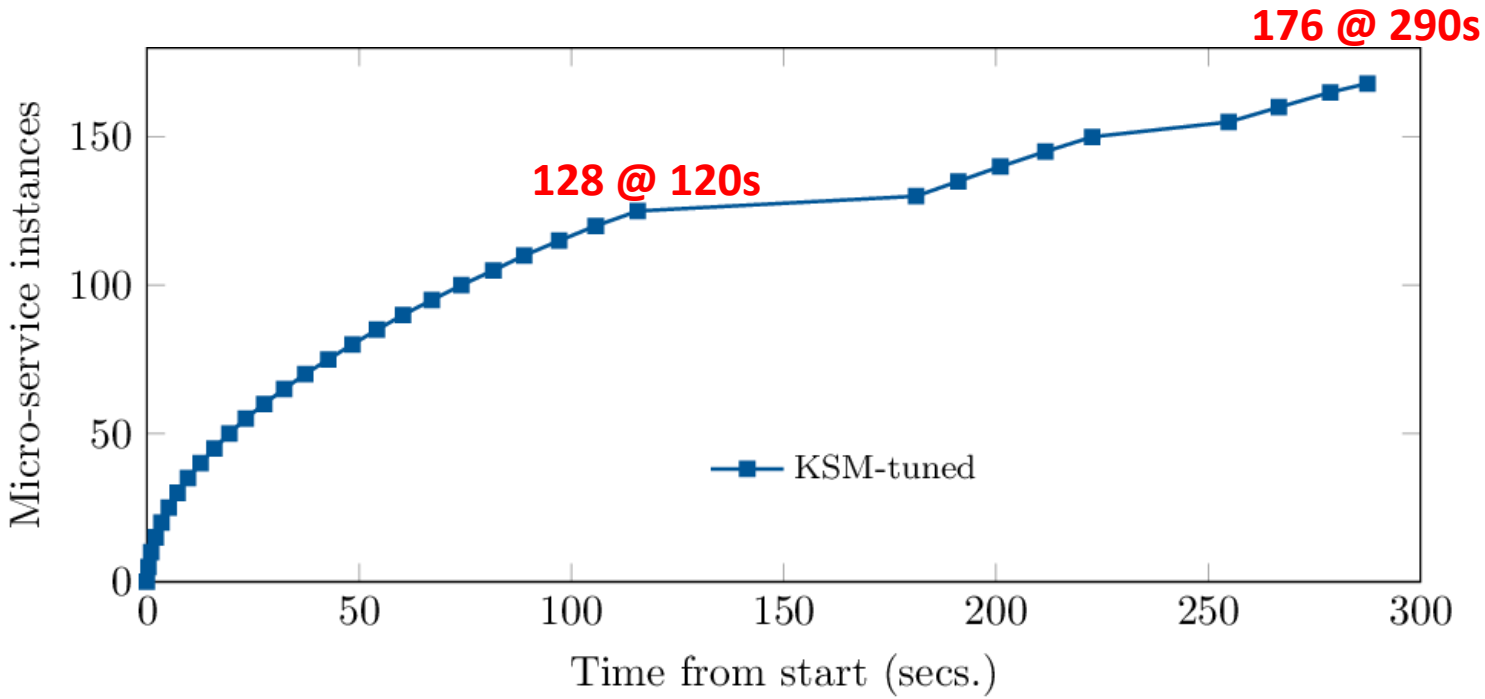
With SCOs



Benchmarking SCO and KSM on Arm Morello

Theoretical: ~180 instances

Auto-tuned KSM (KSM-tuned):
128–176 workers, 120–290 secs

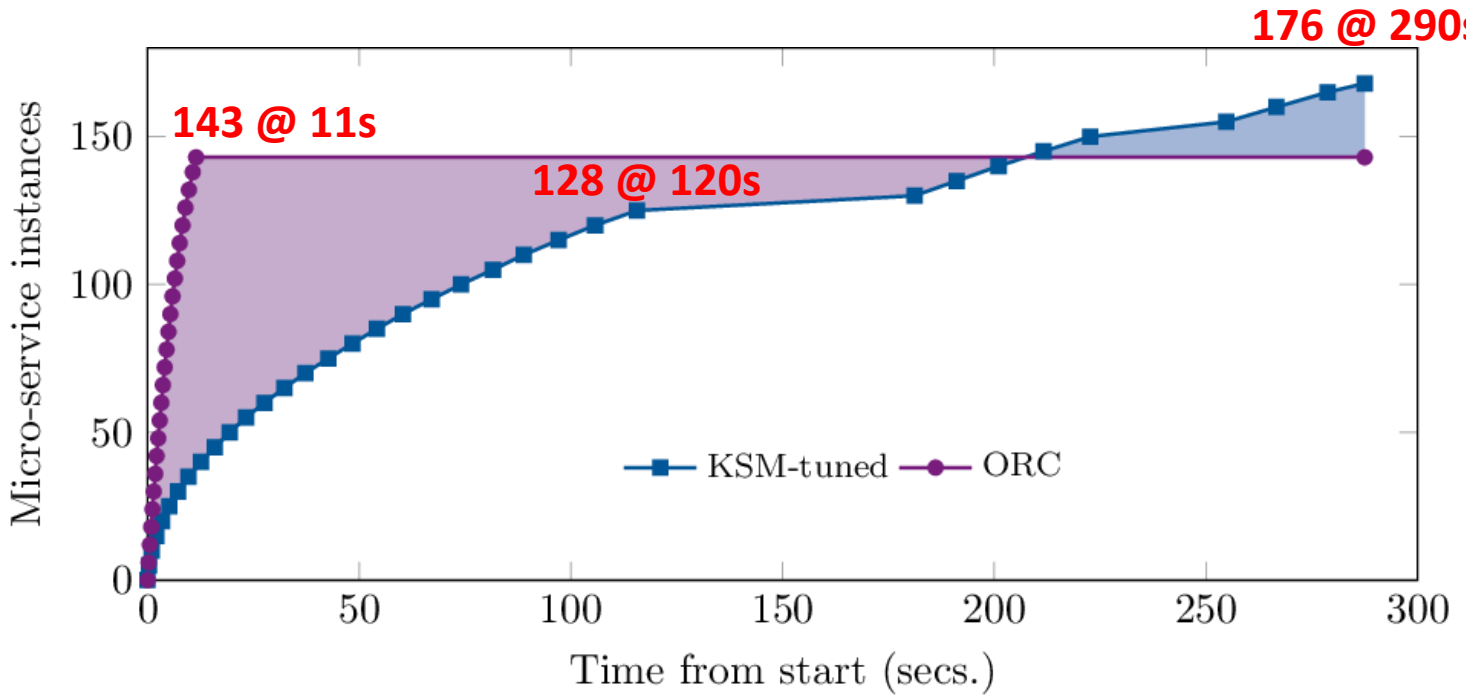


Benchmarking SCO and KSM on Arm Morello

Theoretical: ~180 workers

Auto-tuned KSM (KSM-tuned):
128–176 workers, 120–290 secs

SCO: 143 workers, 11 secs



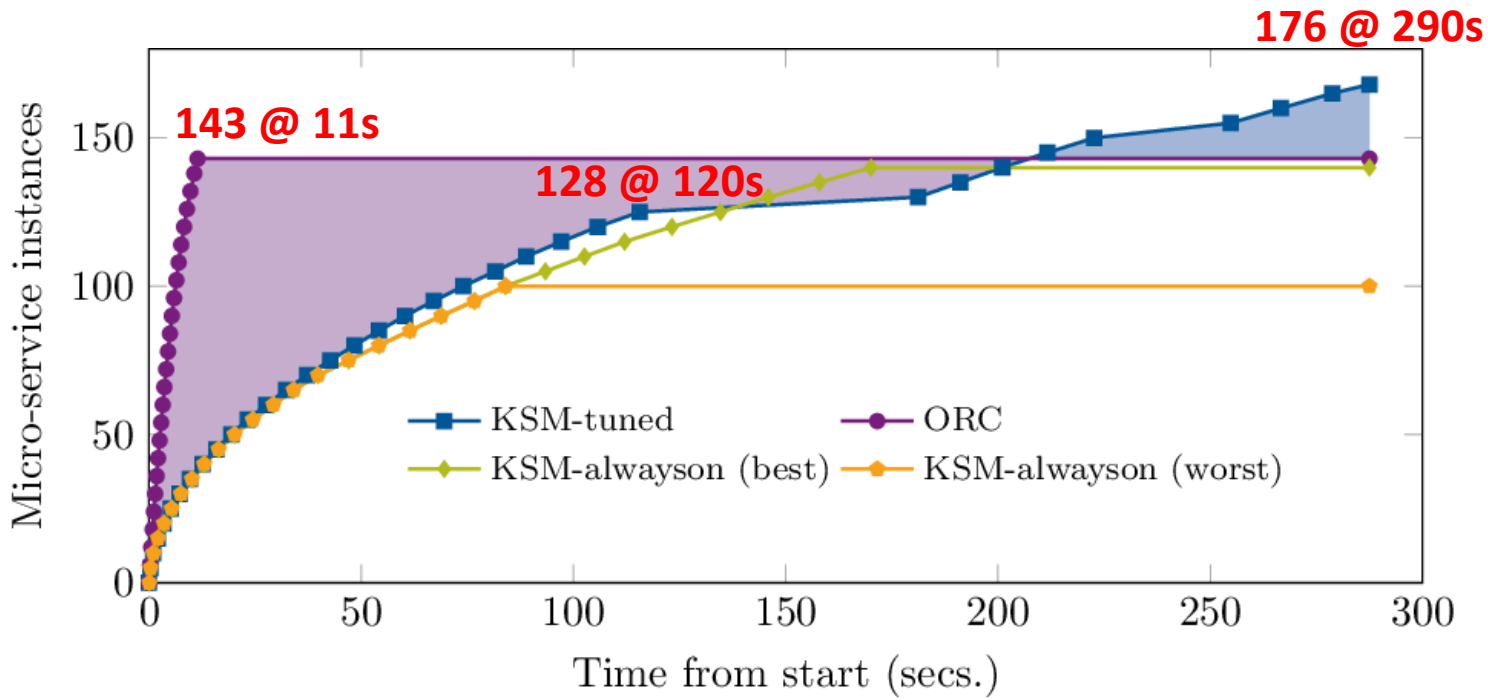
Benchmarking SCO and KSM on Arm Morello

Theoretical: ~180 workers

Auto-tuned KSM (KSM-tuned):
128–176 workers, 120–290 secs

SCO: 143 workers, 11 secs

Always-on KSM:
100–140 workers, 80–170 secs



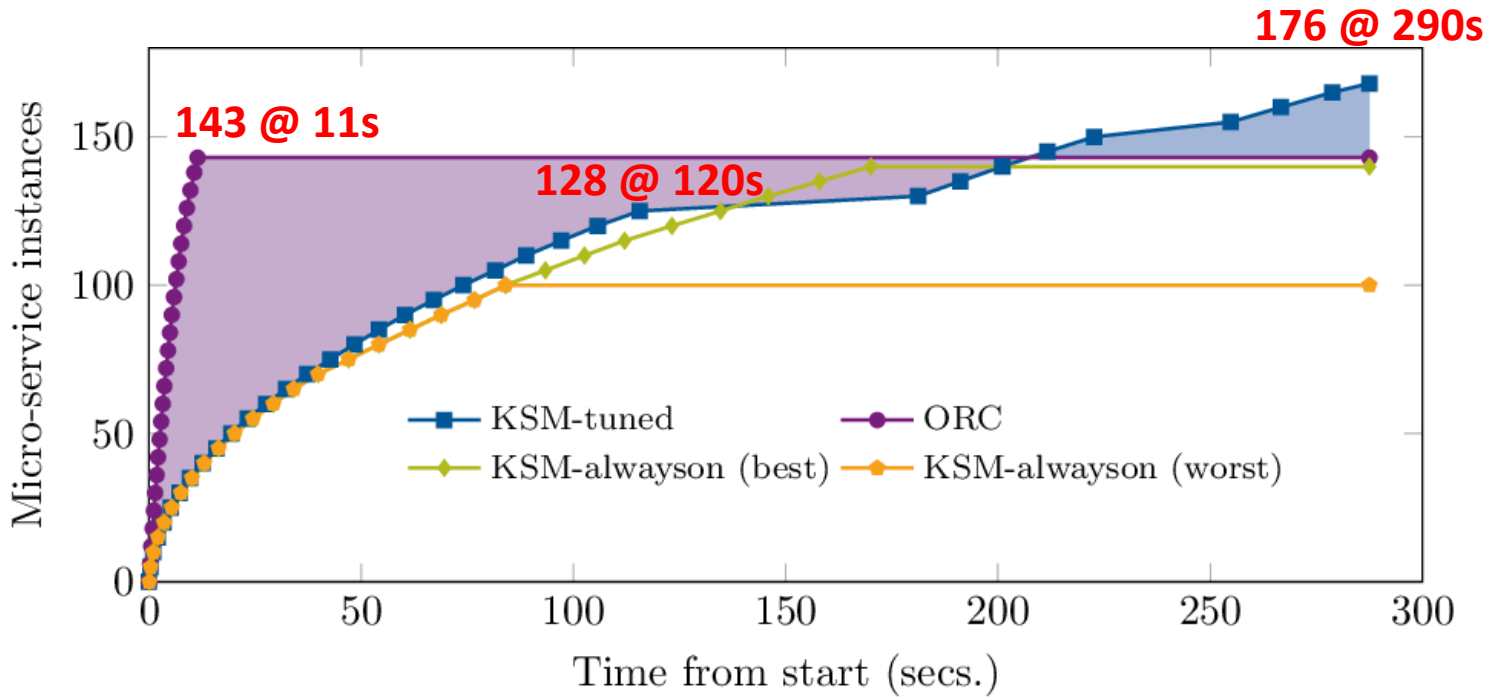
Benchmarking SCO and KSM on Arm Morello

Theoretical: ~180 workers

Auto-tuned KSM (KSM-tuned):
128–176 workers, 120–290 secs

SCO: 143 workers, 11 secs

Always-on KSM:
100–140 workers, 80–170 secs



☛ Always-on KSM performs worse

☛ KSM deduplicates more (data pages), if run for a lot longer

☛ SCO always deduplicates immediately, and has much faster spawning

Conclusions: Slashing the Cloud Tax

Traditional **cloud environments** have **efficiency/isolation trade-offs**

- Require strong isolation between tenants
- Efficient data communication between isolation domains
- Redundant memory contents at all levels (application, libraries, OS, …)

Let's rethink the cloud stack

- Use new **hardware memory capabilities** to avoid traditional OS & MMU overheads
- Design to minimize disruption to existing applications
- Cloud affords us changes across hardware and software stack

CAP-VMs: Strong isolation while enabling fine-grained data sharing

Single-Copy Objects: Consolidate memory through sharing-by-design

Lluís Vilanova
<vilanova@imperial.ac.uk>