Enabling Serverless Research with the vHive Ecosystem

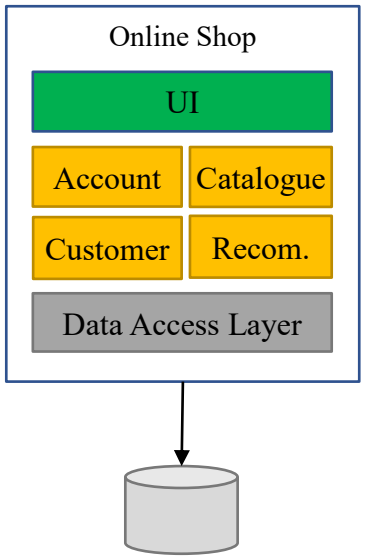
Boris Grot

Edinburgh Architecture & Systems Lab (EASE)

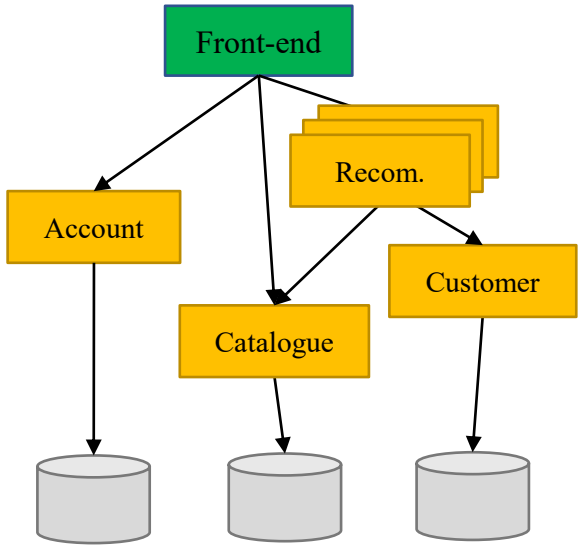
University of Edinburgh

Cloud Applications: from Monoliths to Serverless

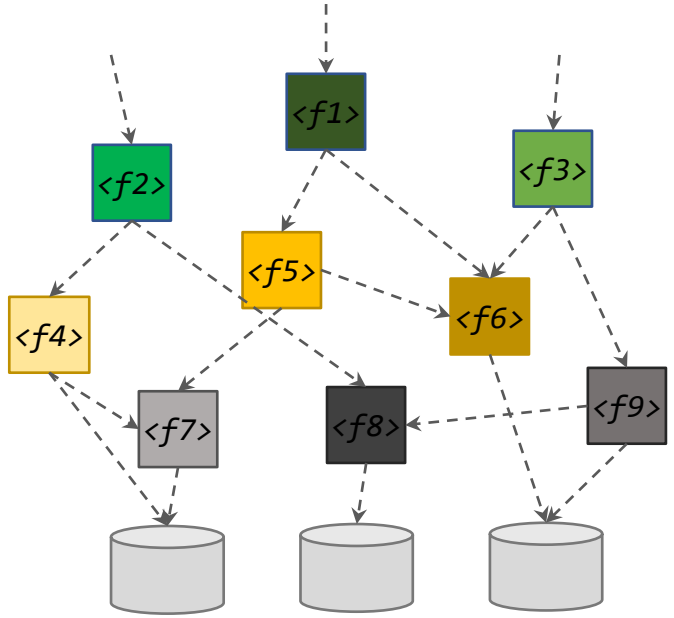
Monolithic app



Microservices



Serverless

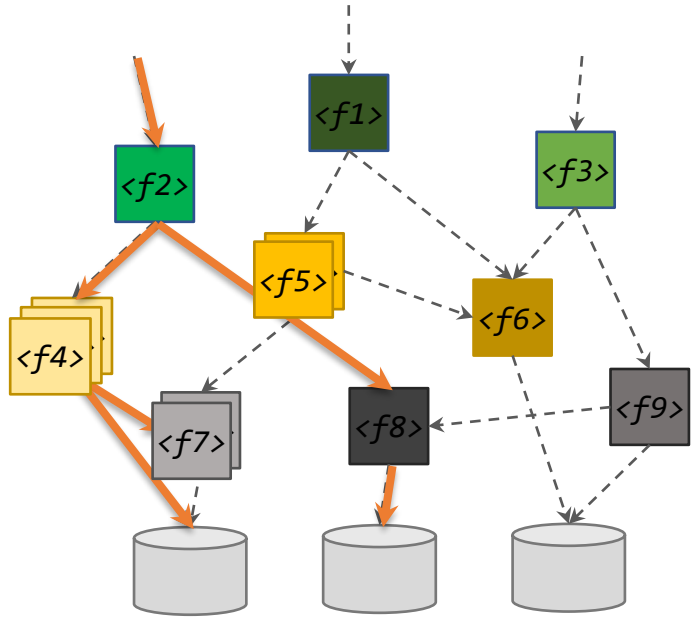


Trend toward **greater modularity** and **disaggregation** in cloud applications

Serverless 101

Datacenter application organized as a collection of stateless functions

- Functions invoked on-demand
  - via triggers (e.g., user click) or by another function
- Functions are stateless: facilitates on-demand scale-in/scale-out
- Developers: pay only per invocation (CPU+memory), not idle time
  - Key difference from monoliths & microservices!
  - Financial incentive to reduce function footprint
- Cloud providers: high density and utilization at the server level



What are the implications of the serverless model?

State of Serverless Clouds Today

Good: Programming & deployment simplicity; pay-per-use cost model

Bad: Poor performance & low efficiency

- Frequent scaling due to traffic changes → cold start delays, overprovisioning
- Functions are stateless → communication bottlenecks inherent
- Massive degree of function interleaving on a server → poor uarch efficiency
- …

Ugly: Proprietary serverless stacks across cloud providers

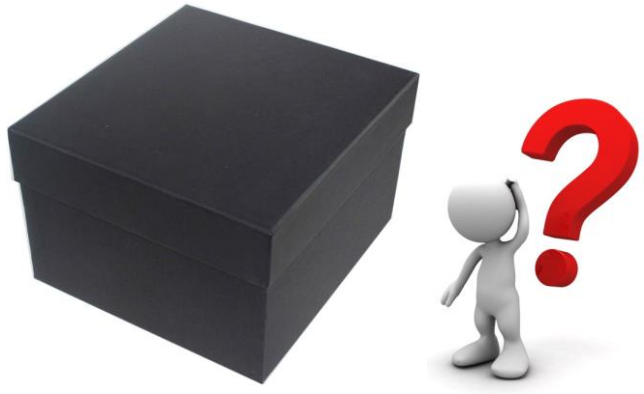
How to study and innovate?

Big challenges are big opportunities for research!

State-of-the-Art in Serverless Experimentation

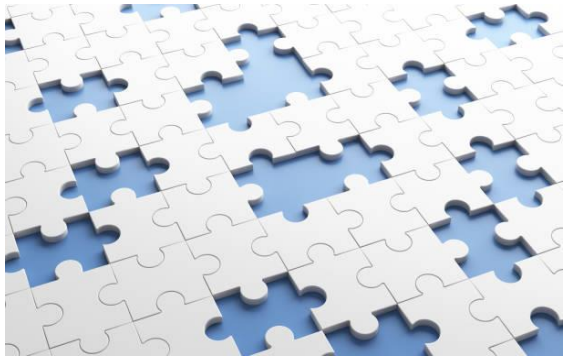
Industry

Bleeding-edge but **proprietary** serverless stacks



Research/academia

Incomplete or **non-representative**



Need for a full-stack open-source framework for serverless research

Idea: Integrate Open-Source Components from across the Industry

Cluster scheduler & Function-as-a-Service API (Google, Cloud Native Computing Foundation)

Kubernetes + Knative

MicroVM (Amazon, Google)

Firecracker gVisor

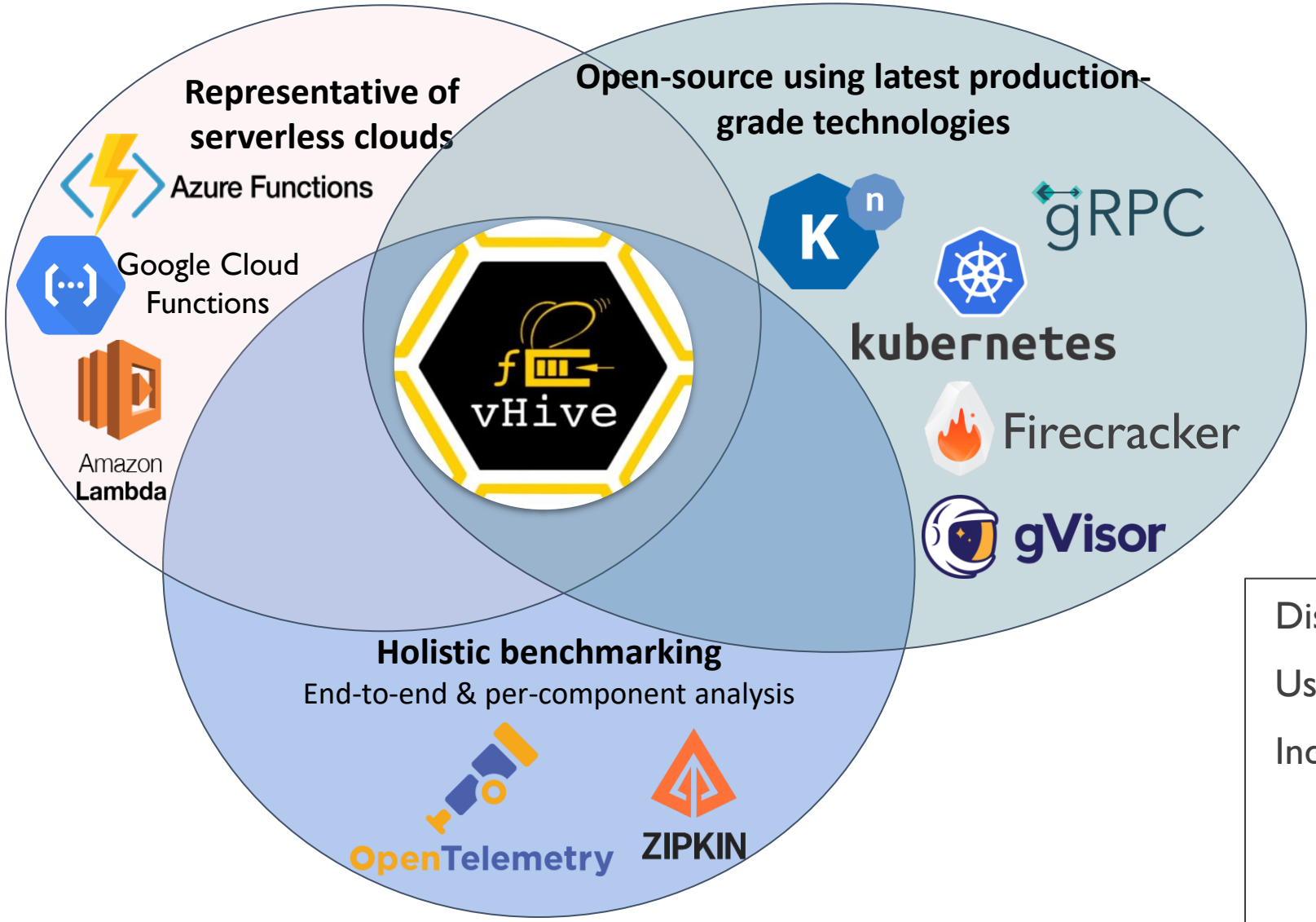
Host management, container runtime (Cloud Native Computing Foundation)

containerd

Communication (Google)

gRPC

The vHive Ecosystem



vHive: an open-source serverless stack
https://github.com/vhive-serverless

Representative of today's clouds

- Knative FaaS API, Firecracker & gVisor MicroVMs, Kubernetes
- First to support Firecracker snapshots

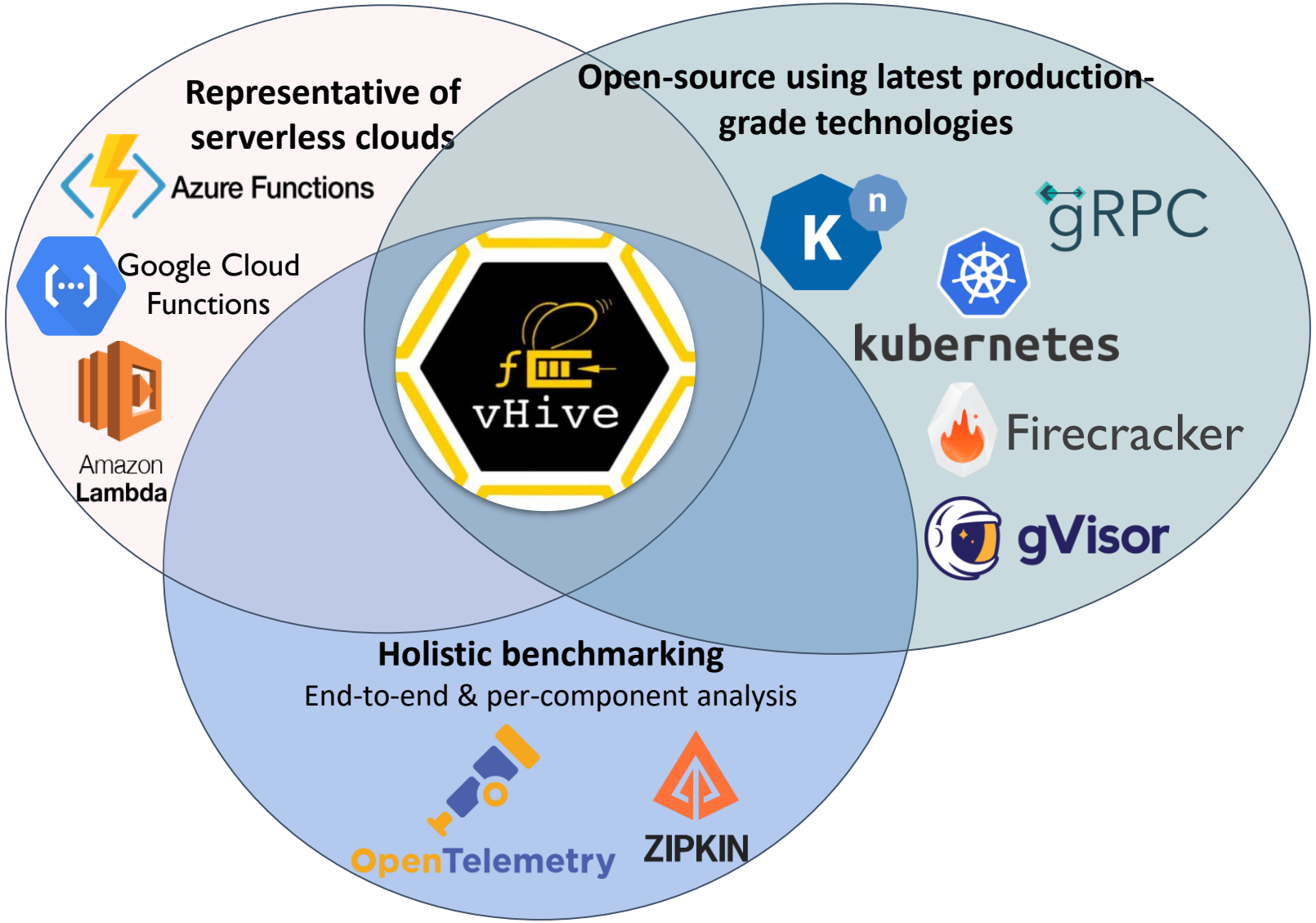
Robust methodology & performance analysis tools

Distinguished Artifact Award @ ASPLOS

Used by 30+ universities for research & teaching

Industry support, collaboration and adoption

The vHive Ecosystem



vHive: an open-source serverless stack
https://github.com/vhive-serverless

Representative of today's clouds

- Knative FaaS API, Firecracker & gVisor MicroVMs, Kubernetes
- First to support Firecracker snapshots

Robust methodology & performance analysis tools

vSwarm: a serverless benchmark suite
https://github.com/vhive-serverless/vswarm

Comprehensive real-world benchmarks

- ML training & inference, video analytics & encoding, MapReduce, distributed compilation
- Varied runtimes & function composition patterns
- Data transfers via different mediums (inline, S3)

Gem5-runnable container images

- Enables full-system microarchitectural simulation

What kind of research can vHive enable?

vHive in action:

Understanding & Accelerating Function Cold Starts

The "Cold Start" Problem

Cloud services exhibit bursty traffic

- Traffic spikes & dips are frequent
- Infrastructure needs to react quickly to traffic changes
- Fast spawning of new function *instances* is crucial

Problem: starting a new function instance (**cold start**) is slow

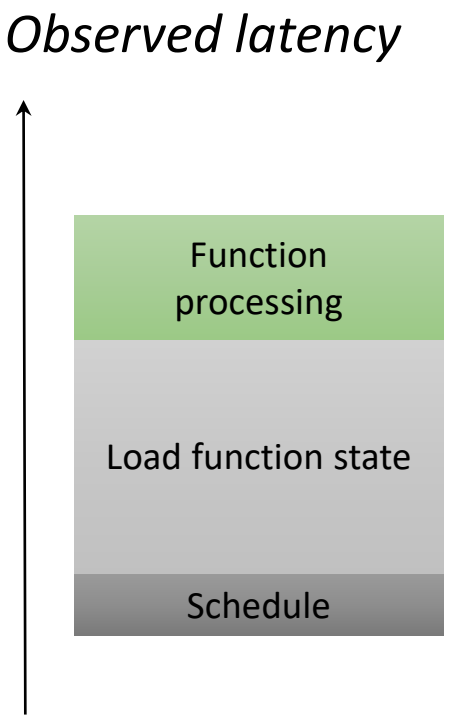
- Potentially multiple orders-of-magnitude slower than executing a warm instance
- High latency of booting the microVM and initializing the runtime

State-of-the-art for faster cold starts:

- Checkpoint a fully-initialized microVM and language runtime into a **snapshot**
- Load a new function instance from the stored snapshot

Do snapshots solve the cold start problem?

Understanding Cold Starts from a Snapshot



Scheduler: predictably low latency [AWS'20]

- Irrespective of warm or cold invocation

Load function state

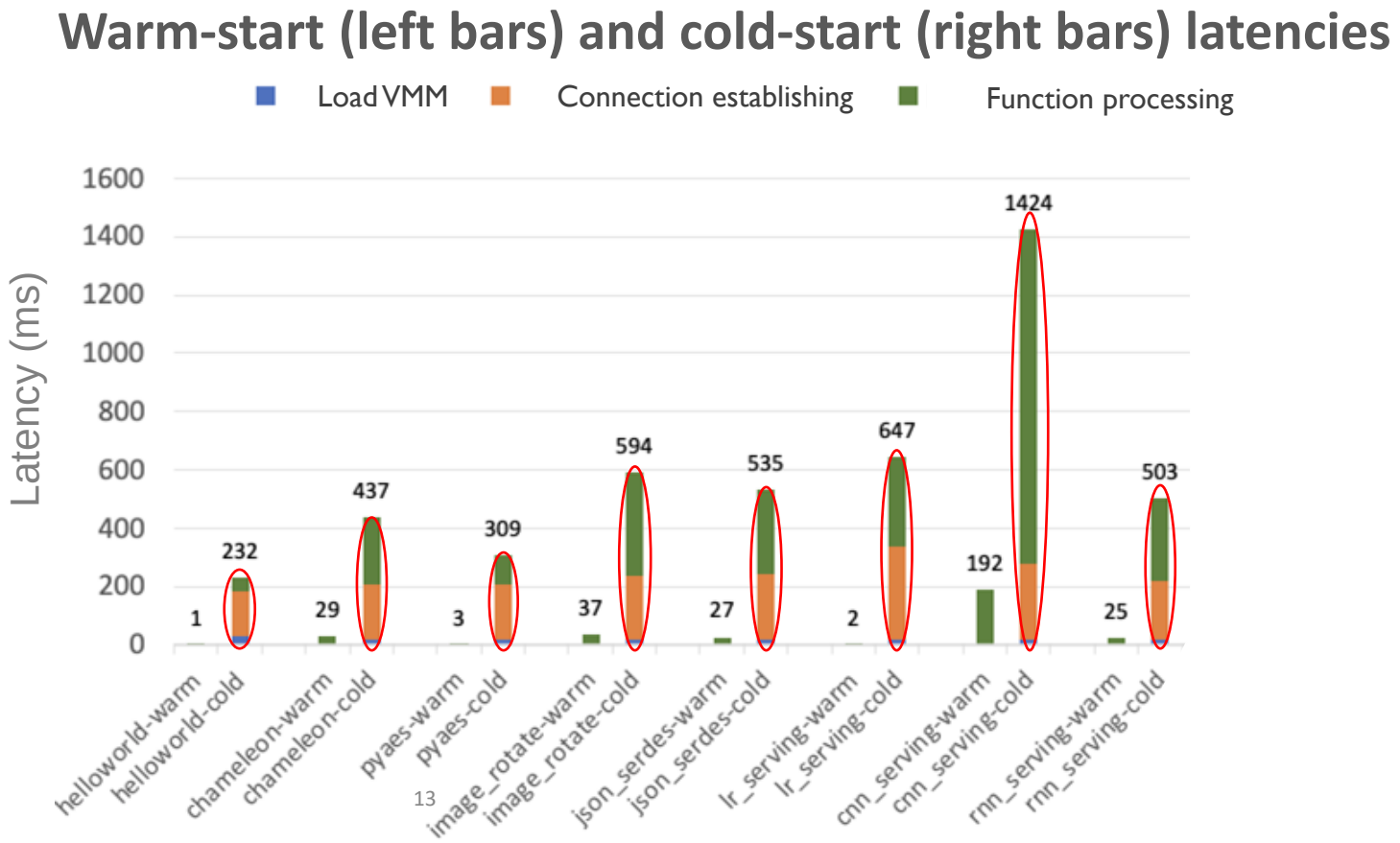
- Setup: vHive with Firecracker microVM + snapshot support

AWS Lambda

1. Restore virtual machine monitor (VMM) state
   a. Restore the architectural state (fast)
   b. Map the guest memory file *without* populating its contents
   c. Resume the VM from the point at which the snapshot was created
2. Restore the connection between the function instance and the Scheduler

Process the function invocation

Performance of Baseline Snapshots



Take-away: cold starts from a snapshot are ~20x slower vs. warm

Cold start delays dominated by **connection restoration** & actual **function processing**

What slows down function processing?

Function Memory Usage in Focus

Functions, even simple ones, have a sizable memory footprint

- Many libraries and modules in user code
- Large guest kernel footprint

Snapshots rely on **lazy paging**

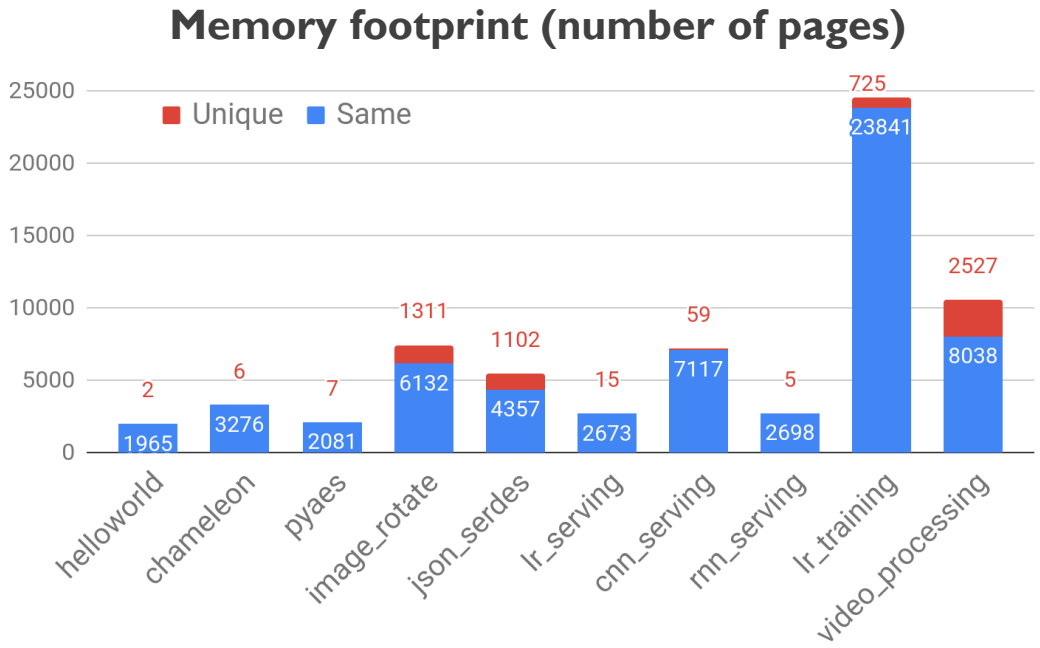
- Guest memory (file) is mapped but not populated with contents
  - Page faults are served **one by one**, on demand
  - Accessed memory pages are scattered on disk → poor spatial locality in disk I/O
- Lazy paging is the underlying cause of the slowdown in functions started from a snapshot

Take-away: serial and sparse disk accesses slowdown function execution

Function Memory Usage in Focus (con'd)

Study: Trace page faults with `userfaultfd` (stock Linux user-level page fault handling mechanism)

- Record memory usage across invocations of the same function



Take-aways:

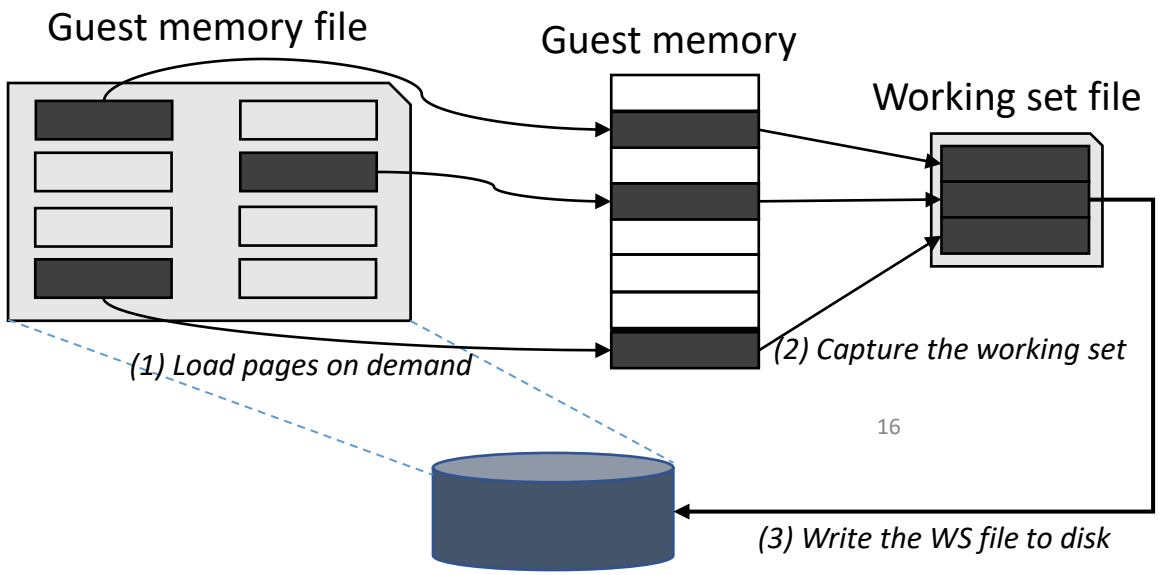
- Actual memory footprint is non-trivial
  - up to 100MB per invocation
- Memory working sets are **stable** across invocations
  - Same language runtime, libraries, guest networking stack, …
  - **76-99% of pages are the same**, even with different inputs!

Idea: Record & prefetch the pages comprising a function's working set

REcord-And-Prefetch (REAP) Snapshots

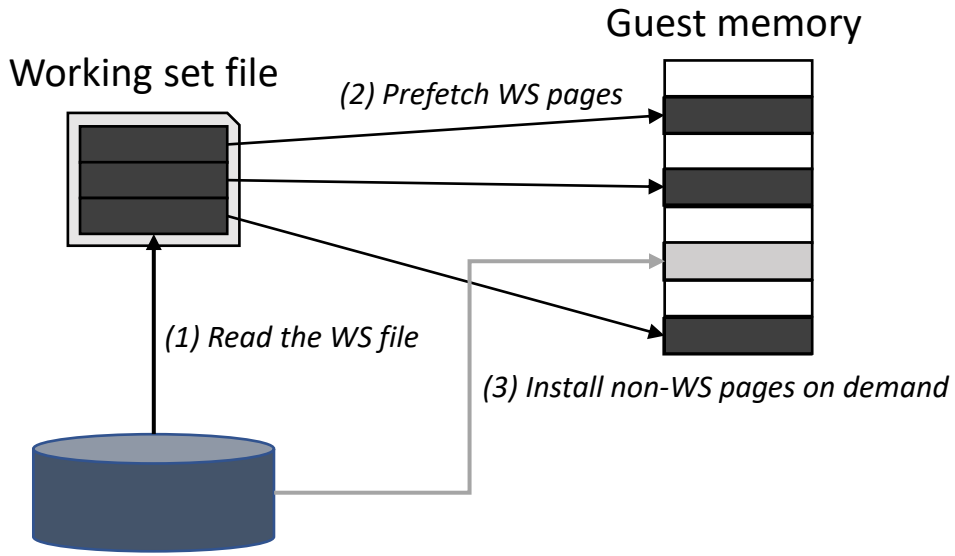
Record phase (1st invocation)

1. Intercept page faults with Linux `userfaultfd`
2. Capture working set (WS) pages in a compact file
3. Write the WS file to disk (SSD, HDD, AWS S3, …)



Prefetch phase (2nd and future invocations)

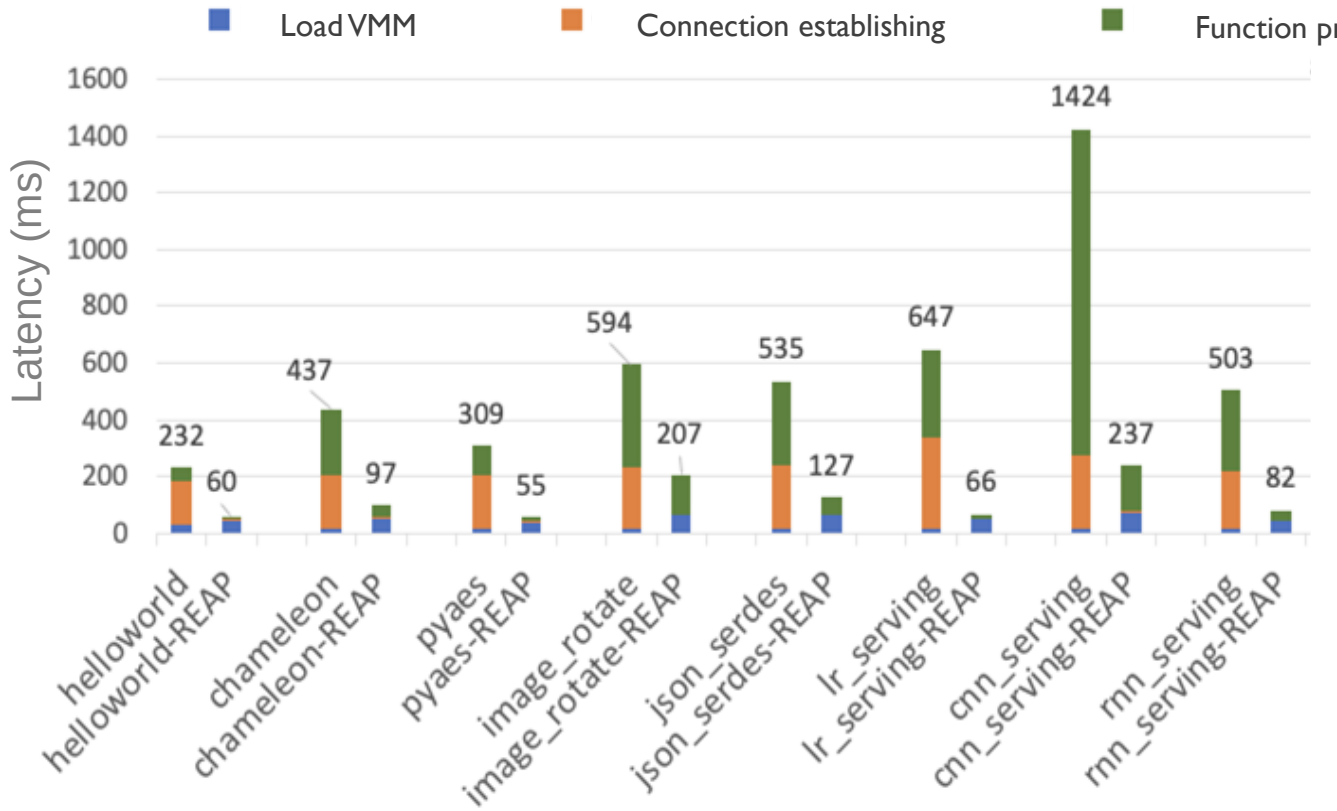
1. Read the WS file from the disk
2. Prefetch **all** WS pages into the guest memory
3. Install **missing,** non-WS, pages **on demand**



REAP trades off a little extra storage for faster cold starts

Evaluation: Baseline snapshots vs REAP

Execution latency from a cold start (left bars: Firecracker snapshots, right bars: REAP)



REAP reduces function processing time by 4.5x on avg

REAP eliminates 97% of all page faults on avg

3.7x faster cold function invocations vs baseline snapshots

REAP Snapshots Take-aways

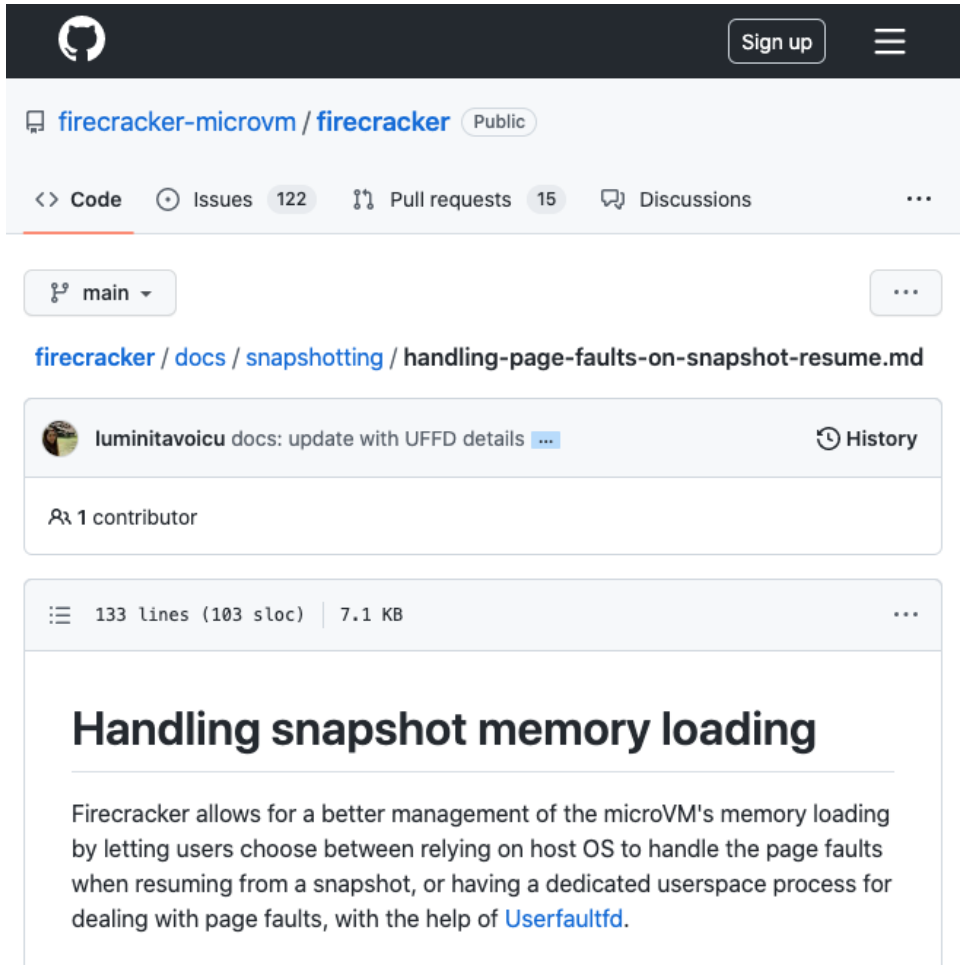
Baseline snapshots suffer from high page fault overhead of **lazy paging**

There exists high commonality in the memory working set across invocations

REAP records the working set of one invocation and prefetches it for subsequent invocations

- Acceleratates snapshot-based cold starts by 3.7x

Amazon has integrated REAP into Firecracker



Enabled by vHive

vHive in action:

Understanding & Accelerating Lukewarm Invocations

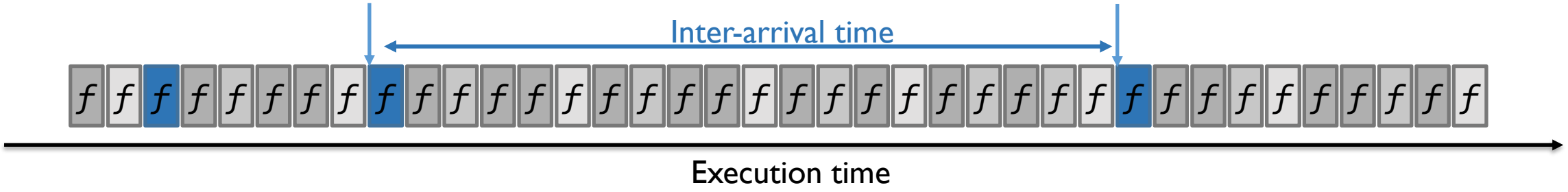
Serverless on a Server

Unique characteristics:

- Short function execution times: a few ms or less is common
  - Contrast: Linux scheduling quantum: 10-20ms
- Small memory footprint: as low as 128MB per instance
- Infrequent invocations (seconds or minutes) [Microsoft Azure @ATC20]

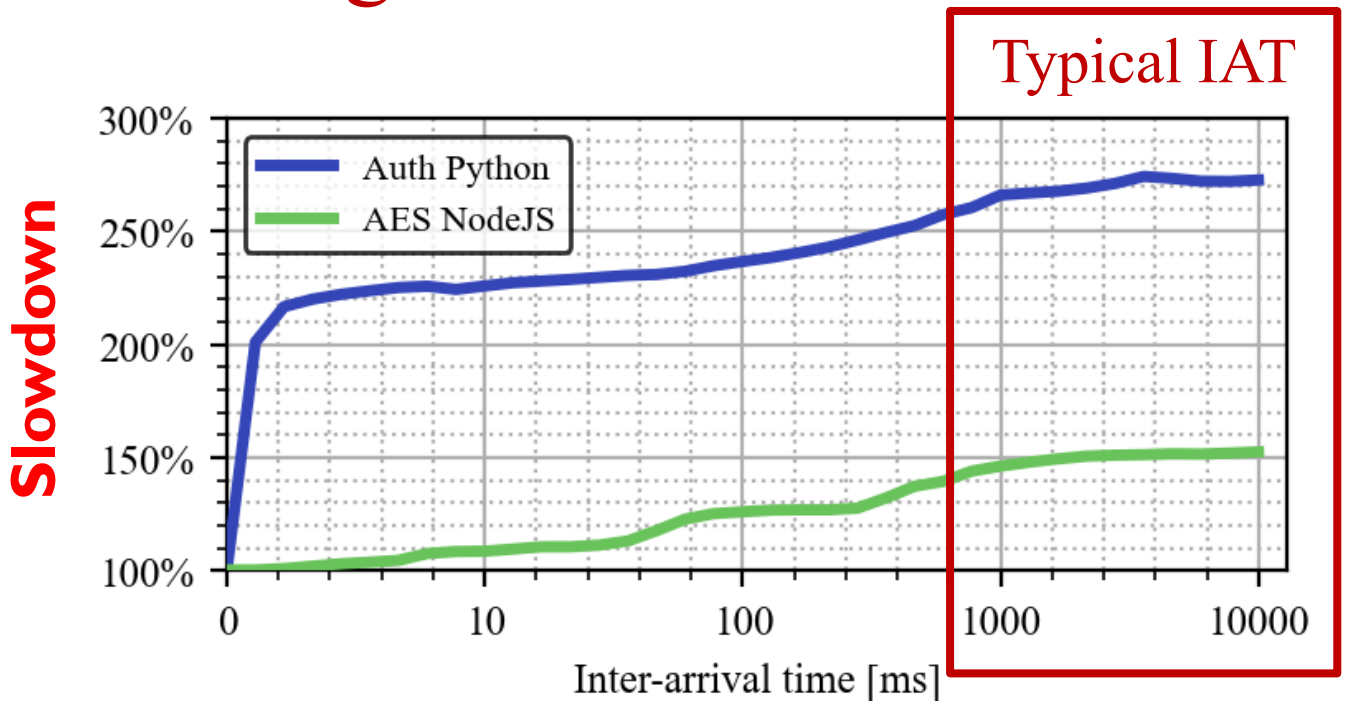
Implications:

- Thousands of functions resident on a server
- **Huge degree of interleaving** between two invocations of the same function



What are the implications for microarchitecture?

Effect of Interleaving



Longer inter-arrival times → Higher degree of interleaving → Higher slowdown

Drastic performance degradation for typical inter-arrival times (IATs)

- Up to 2.7x slowdown for IAT > 1s

What causes the performance degradation?

Characterization Methodology

Compare back-to-back to interleaved executions of a function

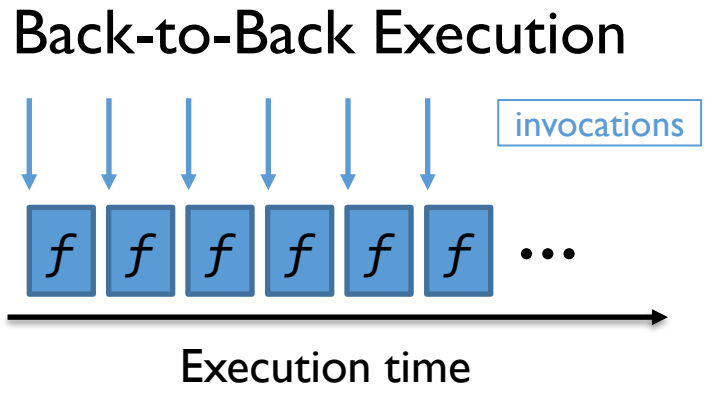
- Function-under-test runs isolated
- Interleaving modelled by a stressor

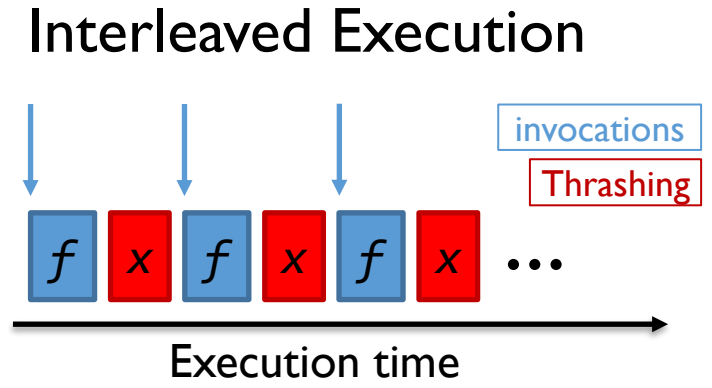
Use **Top-Down Methodology** for analysis

- Machine: Intel Broadwell CPU
  (10 cores, SMT disabled, 32KB L1-I/D, 256KB L2/core, 25MB LLC)
- Collect CPU performance counters

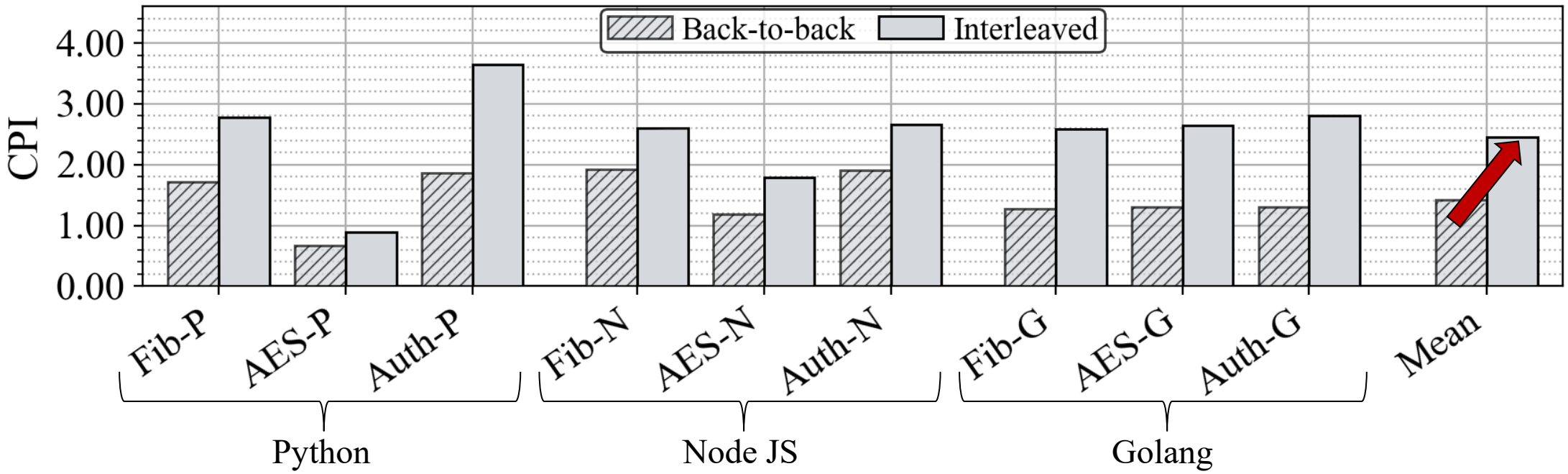
Serverless workloads: 20 functions

- Large variety in functionality and runtimes
- Compiled, JIT-ed and interpreted languages
- Publicly available https://github.com/vhive-serverless/vswarm



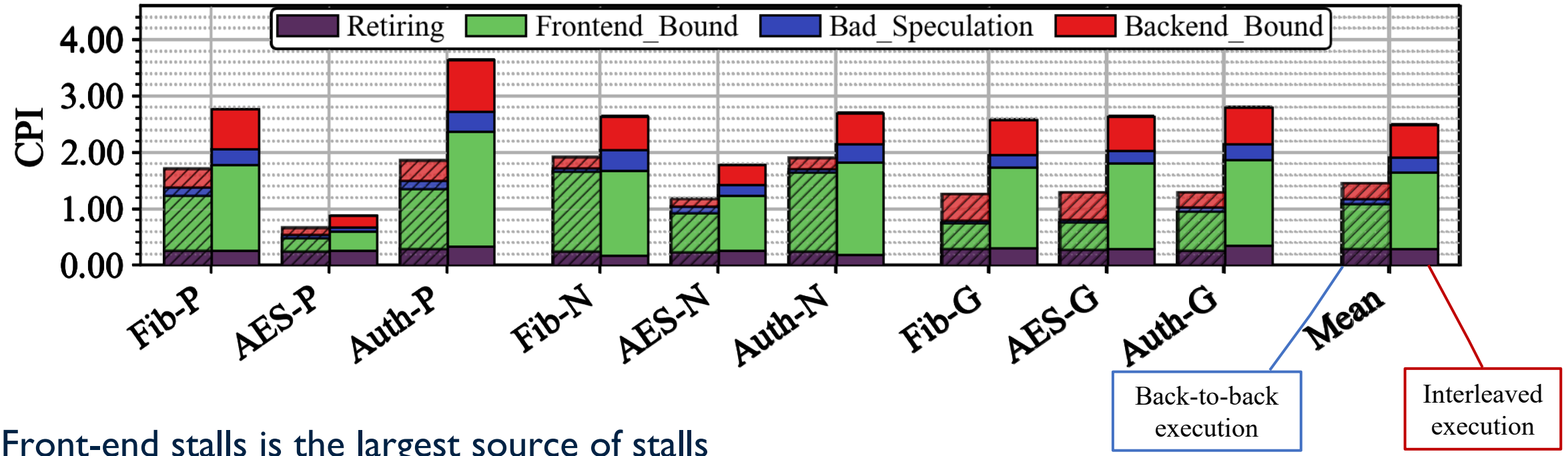


Understanding the Impact of Interleaving



- Interleaving increases the mean CPI by 70%
- Reason: Lukewarm execution
  - Function in memory, but no μ-arch state on-chip

Top-Down CPI Analysis



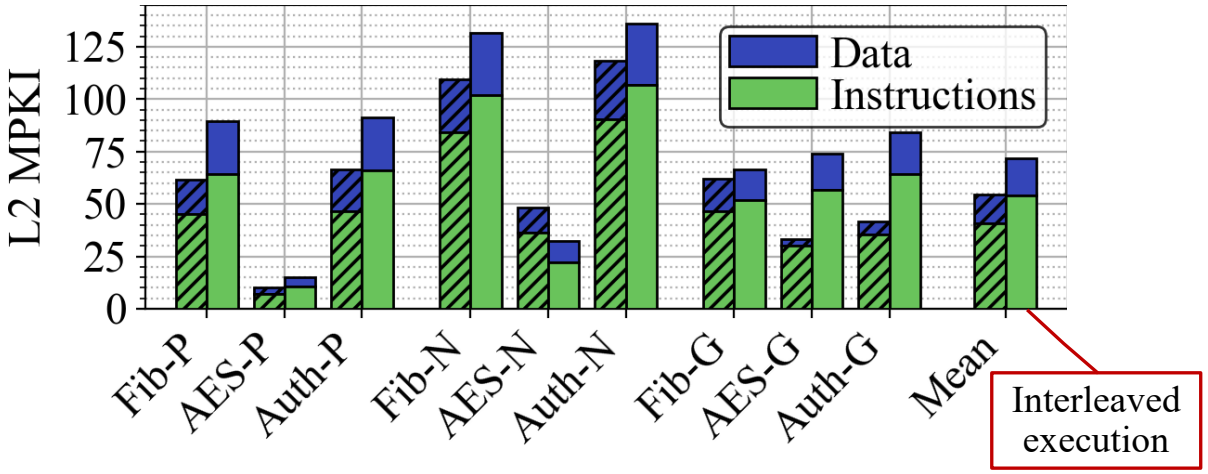
- Front-end stalls is the largest source of stalls
- 56% of additional stall cycles in interleaved execution come from **fetch latency**

Identical trends on Intel IceLake

Instruction delivery a critical performance bottleneck for warm invocations

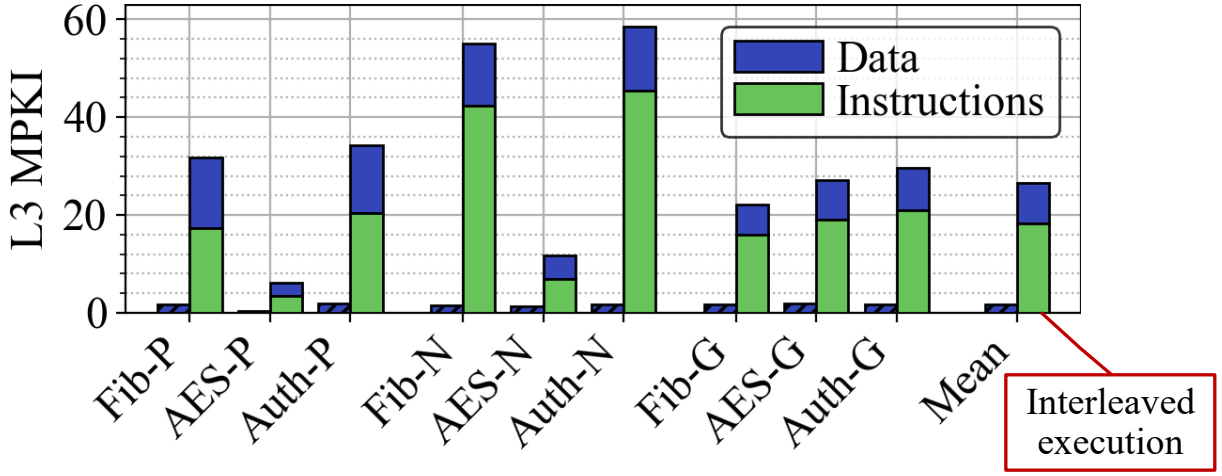
Instruction Fetch Pain Points

L2 Cache *(256KB/core)*



- Serverless workloads frequently miss in L2 cache
  - (50+ MPKI, on average)
- Dominated by instruction misses
- Similar for both back-to-back and interleaved

L3 Cache *(25MB)*



- Almost no L3 instruction misses for back-to-back execution
- Frequent L3 misses for instructions under interleaving (18 MPKI)
  - Instructions fetched from main memory → high stall cycles

L3 instruction misses hurt performance under interleaving

Understanding Instruction Misses

Studied instruction traces from 25 consecutive invocations of each function.

Compared **instruction footprint** & **commonality** at cache-block granularity across invocations

Two key insights:

1. **High commonality** across invocations
   - > 85% of cache blocks are the same in all invocations
2. **Large instruction footprint**: 300KB-800KB
   - Deep software stacks result in large amount of code

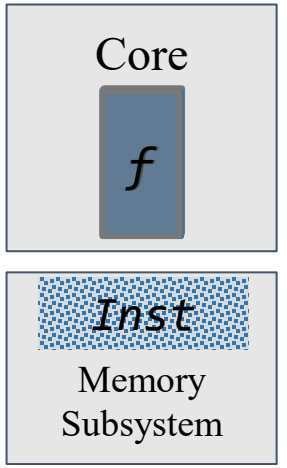
Identified a common problem for serverless functions:

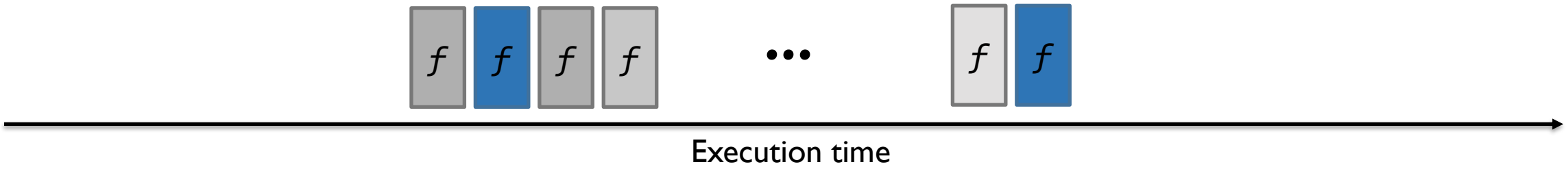
→ **Large instruction** footprints cannot be maintained on-chip under heavy **interleaving**

Addressing Cold On-chip Instruction State

Basic Idea:

- **Exploit high commonality** of function invocations
  - Prefetch common instruction state
- **Record instruction** working set of one invocation
- **Restore** the instruction working with the next invocation

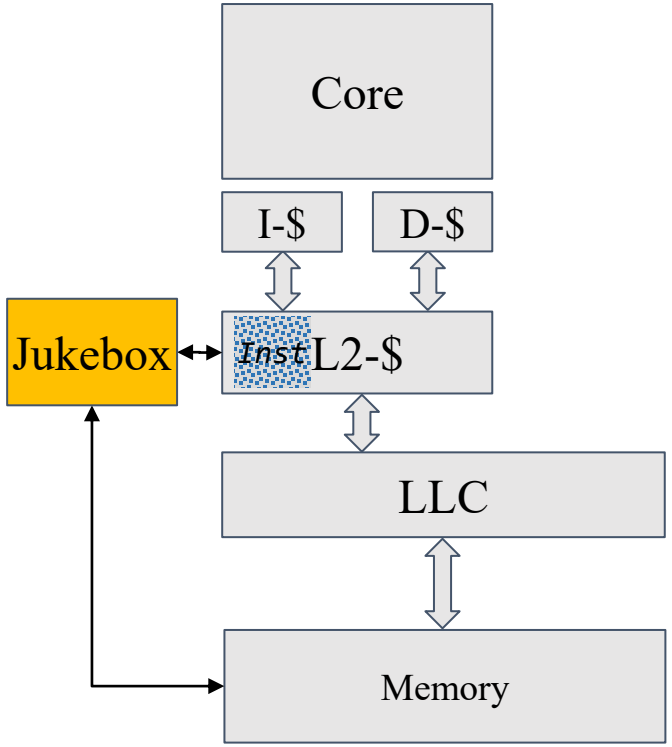




Jukebox: Instruction Prefetcher for Serverless

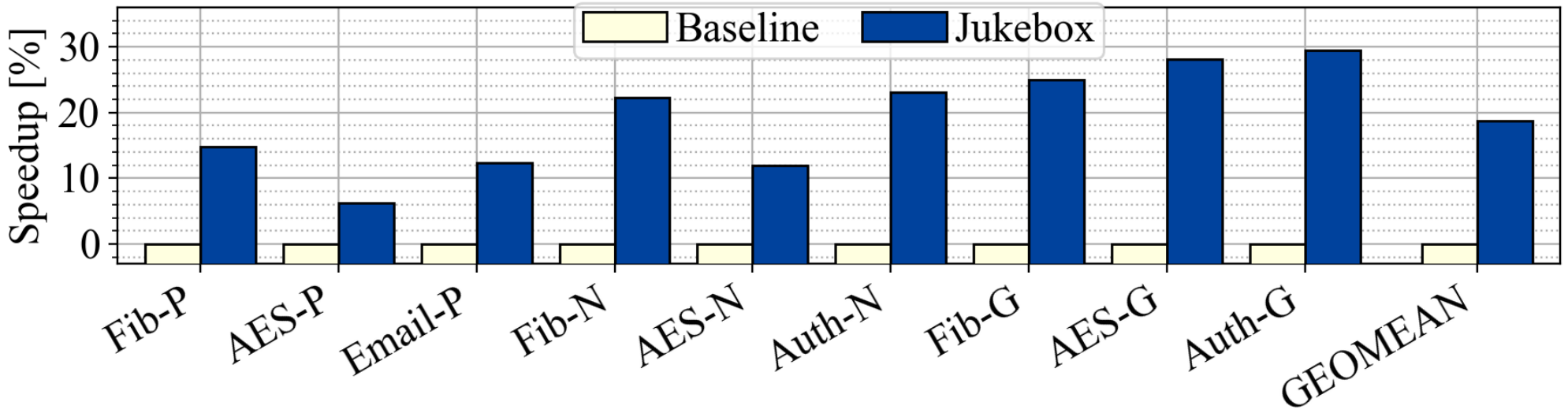
Jukebox: record-and-replay instruction prefetcher for lukewarm serverless function invocations

- **Record: L2 misses** using a spatio-temporal encoding
  - Stores records in main memory
- **Replay**: prefetch the recorded addresses **into the L2**
- Fully decoupled from the core
  - Triggered by function invocation
- Operates on virtual addresses
  - Not affected by page reallocation
  - Prefetching prepopulates TLB



Jukebox records and replays L2 instruction working sets

Jukebox: Performance Improvements



Jukebox's recording and replaying of instruction working sets:

- Improves performance by 18%, on average
  - Consistent improvement across benchmarks
- Covers >85% of off-chip instruction misses
- Requires only 32KB of metadata per function instance

Jukebox is simple & effective

Jukebox: Prefetch Effectiveness

Efficacy of recording and replaying instruction working sets

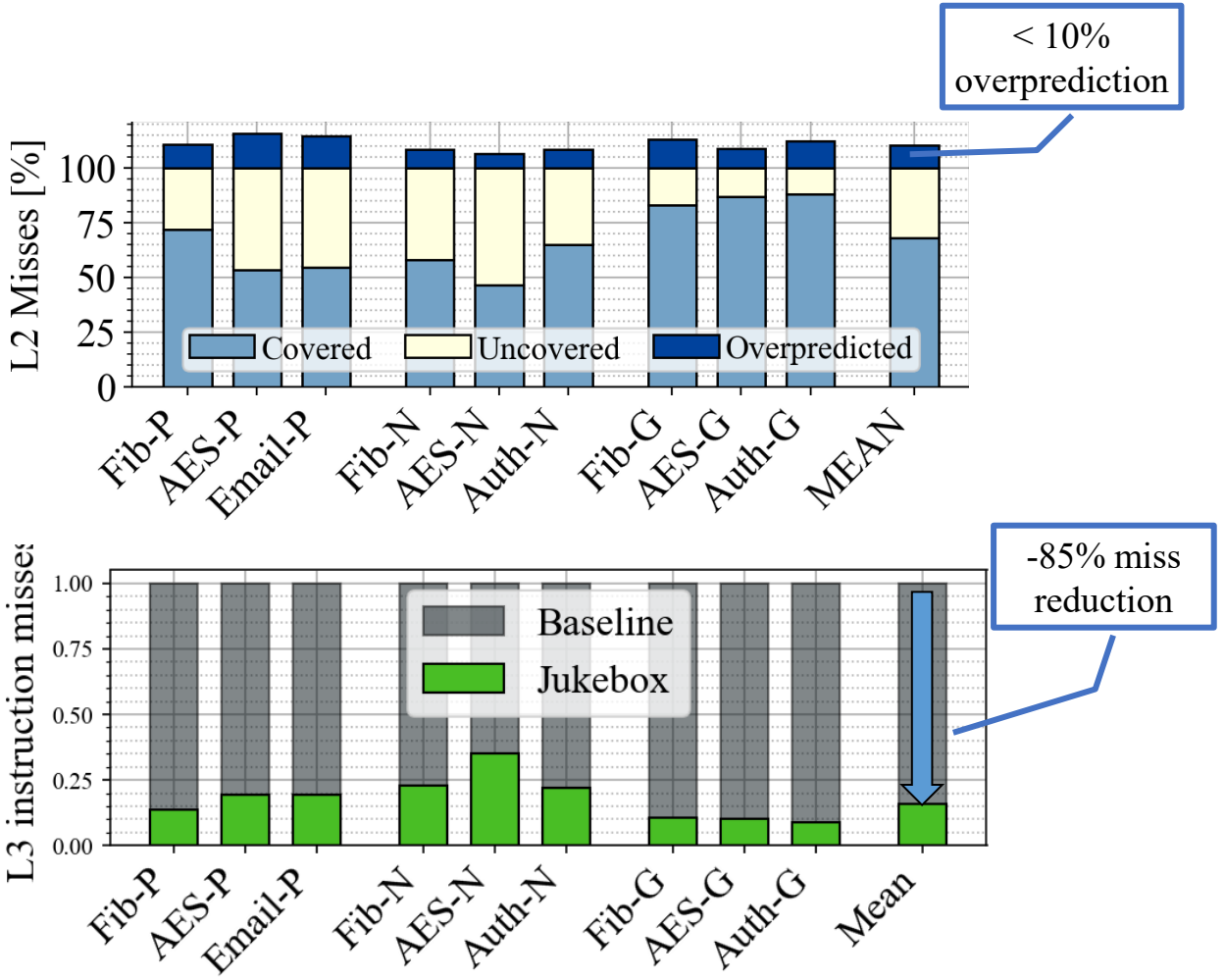
Few overpredictions (<10%)

- High commonality → accurate prefetching

Good coverage:

>85% reduction in LLC misses for instructions

Low metadata cost: 16KB per instance



High efficacy validates Jukebox's design

Jukebox Take-aways

Serverless functions present new challenges for modern CPUs

→ Need a **representative infrastructure** to study serverless stacks: **vHive**

→ vHive enables open serverless research at any scale

Used vHive to reveal a CPU bottleneck due to **lukewarm executions**

→ Large instruction footprints cannot be maintained on-chip under heavy function interleaving

→ Frequent off-chip misses for instructions expose the CPU to long-latency stalls

Jukebox: Record-and-replay instruction prefetcher for lukewarm serverless functions

→ Simple and effective solution for cold on-chip instruction state

→ Improves performance by 18% with 16KB of in-memory metadata per instance

Acknowledgements

Students & interns

Dmitrii Ustiugov (now @NTU-Singapore)
David Schall
Artemiy Margaritov
Shyam Jesalpura
Theodor Amariucai
Harshit Garg
Plamen Petrov
Michal Baczun
Yuchen Niu
Amory Hoste
Bora M. Alper

External collaborators

Rustem Feyzhanov (Instrumental)
Francisco Romero (Stanford)
Marios Kogias (Imperial)
Edouard Bugnion (EPFL)
Ana Klimovic (ETH)

Industry supporters

Join our Serverless Research Community

https://vhive-serverless.github.io

Thank you!

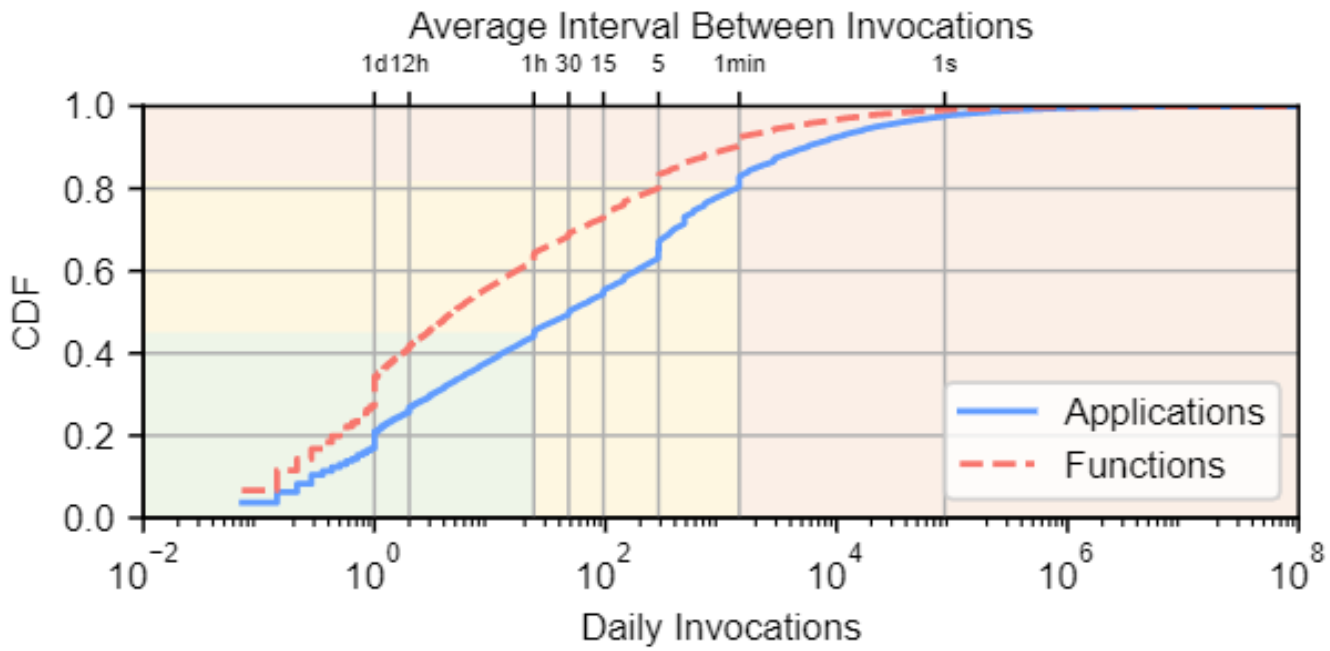
Questions?

EASE Lab

https://ease-lab.github.io

Backups

Serverless Workload Characteristics



Functions are short

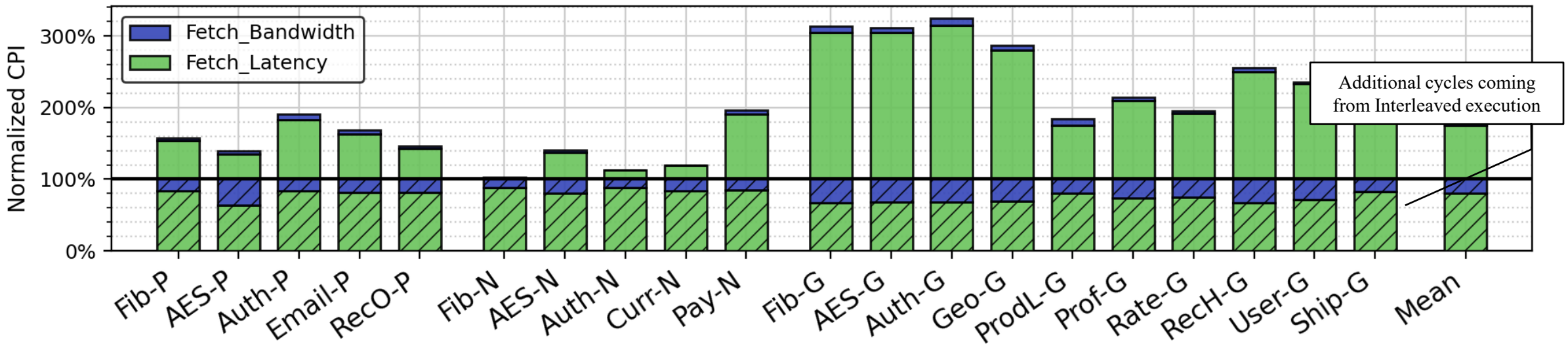
- Became significant shorter in last years

Functions are invoked infrequently

- Seconds to minutes.

Jukebox

Front-end in Focus



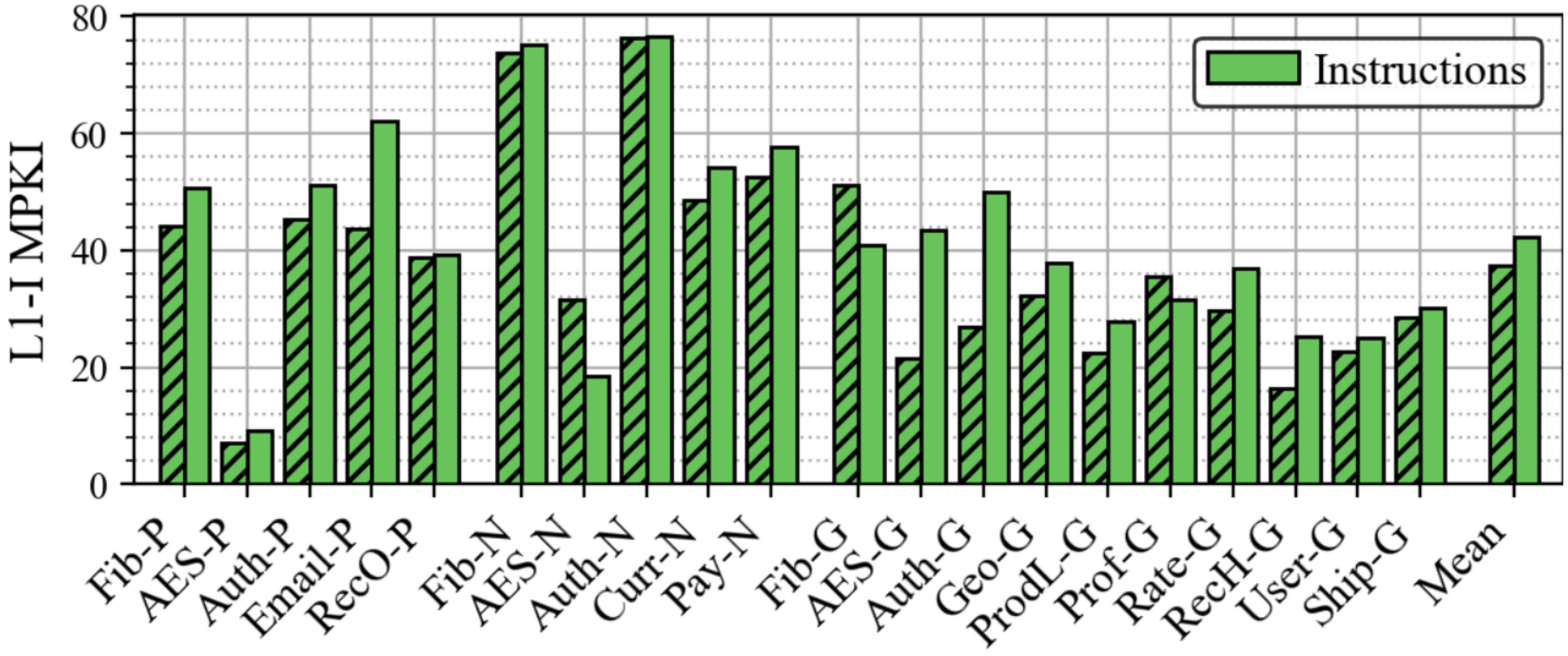
Fetch latency is the biggest source of stall cycles

Fetch latency **doubles** under interleaving

CPI normalized to Back-to-back execution

Instruction fetch latency a key performance bottleneck

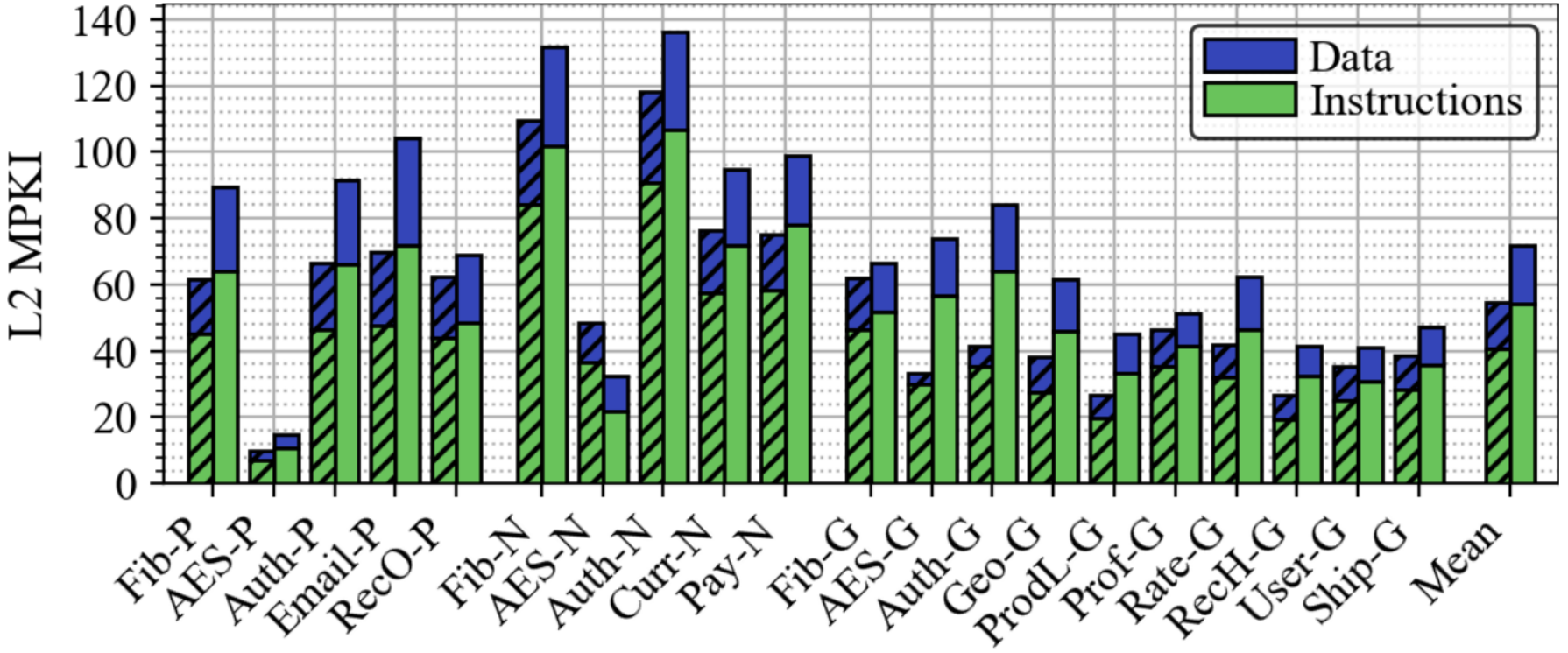
Instruction Misses L1-I



Serverless function execution suffer from large amount of instruction misses.

Note: I-cache misses is lower than L2-cache misses. This is due to how the hardware counters are implemented and is a very common case. Refer to this article for more information: http://sites.utexas.edu/jdm4372/tag/cache/

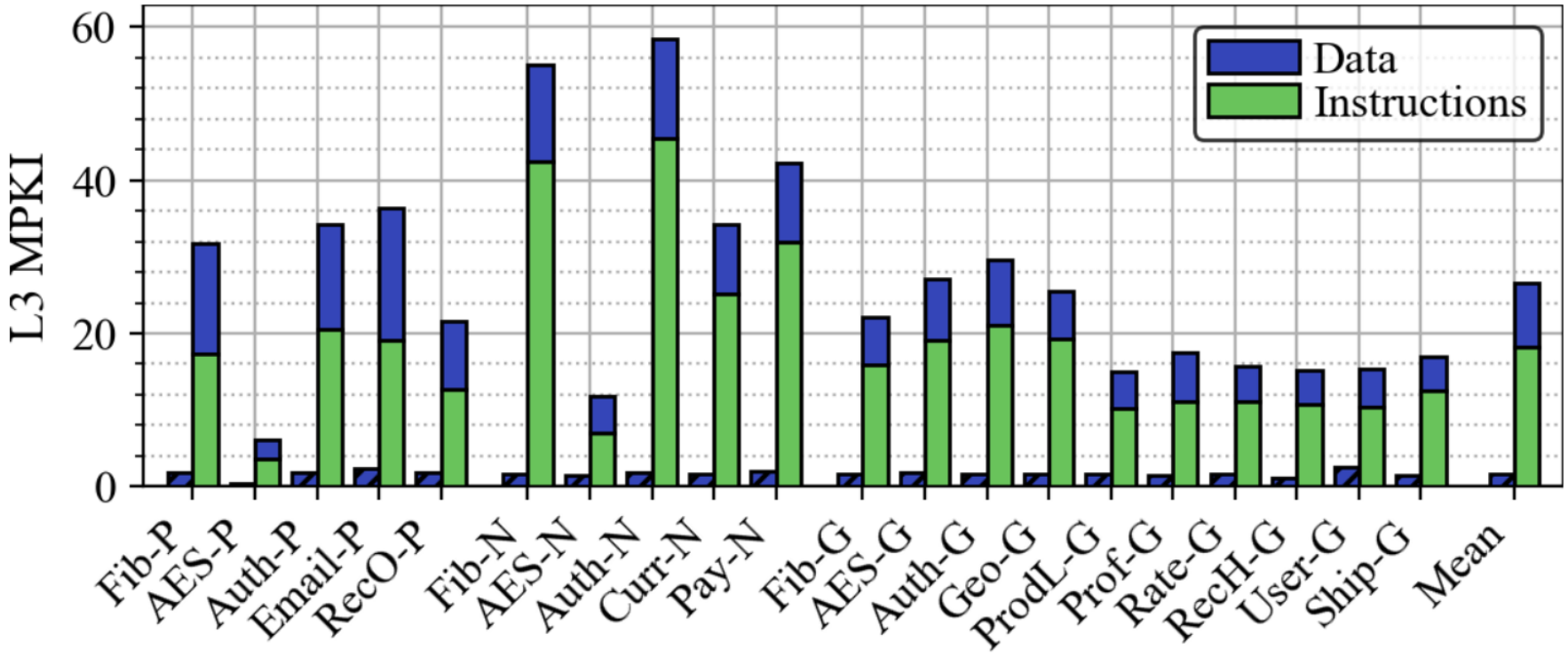
Instruction Misses L2



Serverless function execution suffer from large amount of instruction misses.

Note: I-cache misses is lower than L2-cache misses. This is due to how the hardware counters are implemented and is a very common case. Refer to this article for more information: http://sites.utexas.edu/jdm4372/tag/cache/

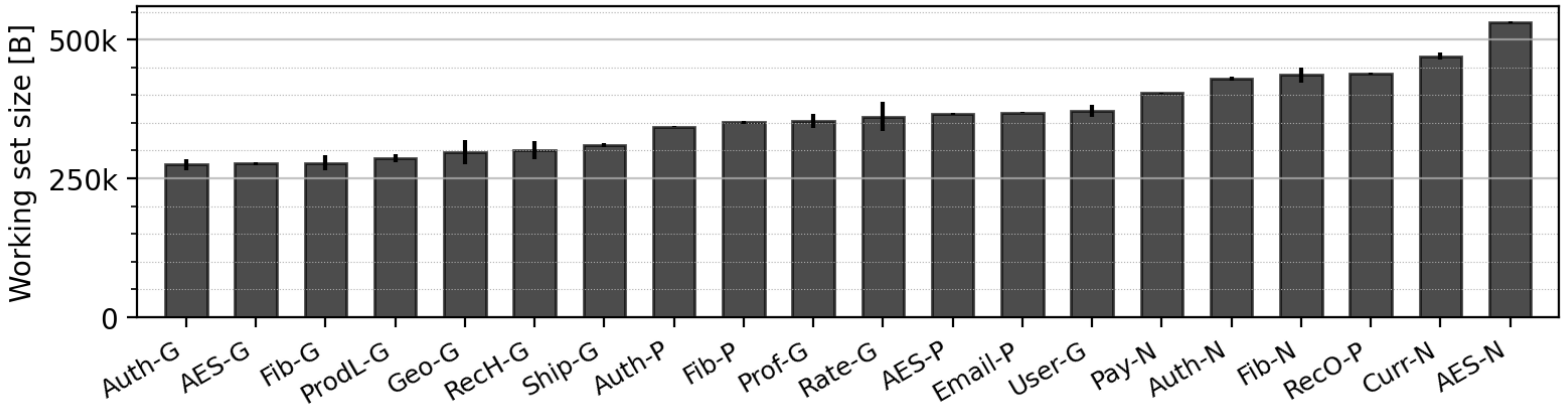
Instruction Misses L3



Serverless function execution suffer from large amount of instruction misses.

Note: I-cache misses is lower than L2-cache misses. This is due to how the hardware counters are implemented and is a very common case. Refer to this article for more information: http://sites.utexas.edu/jdm4372/tag/cache/

Instruction working set size



Large instruction working set despite short execution time

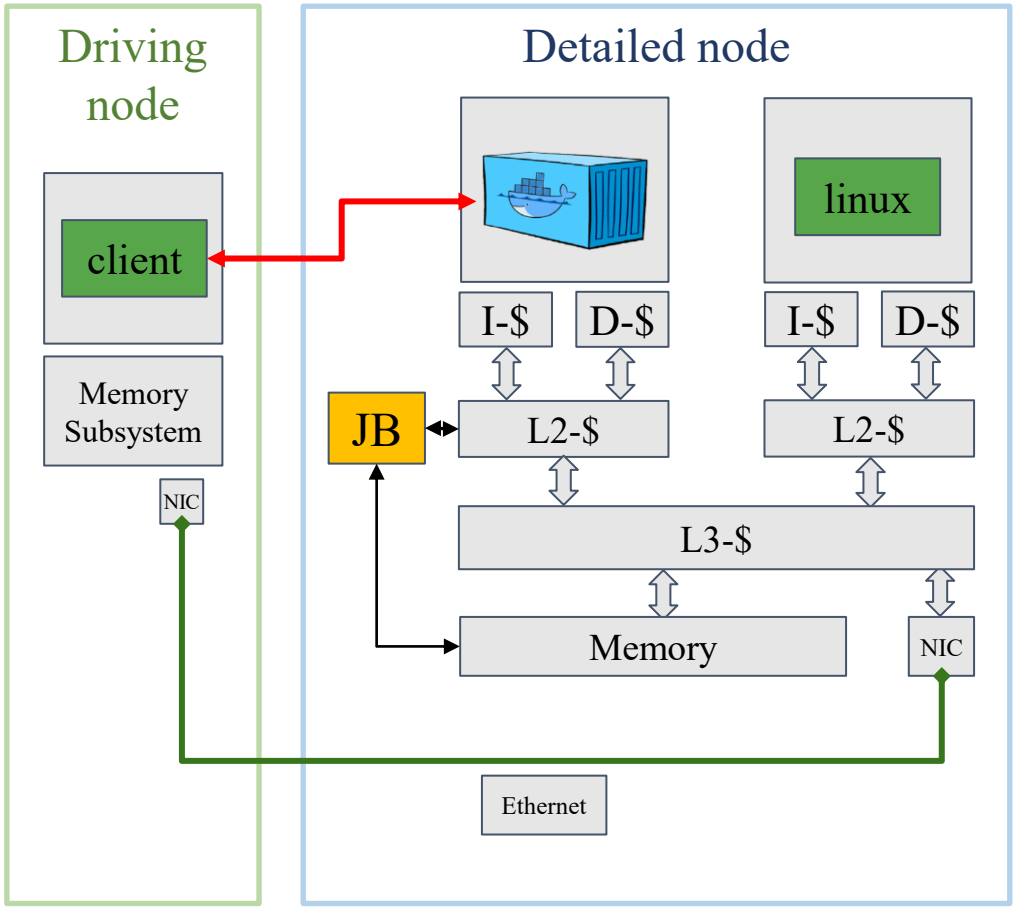
Similar instruction working set size over different invocations

Evaluation Infrastructure

Use **gem5** simulator for evaluating Jukebox

- Detailed model of the server node
  - *Dual core Skylake-like CPU model*
  - *32KB L1-I/D, 1MB L2/core, 8MB L3*
- Secondary node for driving invocations.
- Functions run in isolation
- **Cycle accurate** simulation of the **full system**
  - Exact same software stack as on real hardware

    (Ubuntu 20.04, kernel: 5.4, same container images)
  - **First support for containers in gem5**
    - Publicly available: https://github.com/vhive-serverless/vSwarm-u



Representative infrastructure for detailed evaluation