Architectural Implications of FaaS Computing

Mohammad Shahrad, Jonathan Balkind, and David Wentzlaff

Wednesday, October 30, 2019

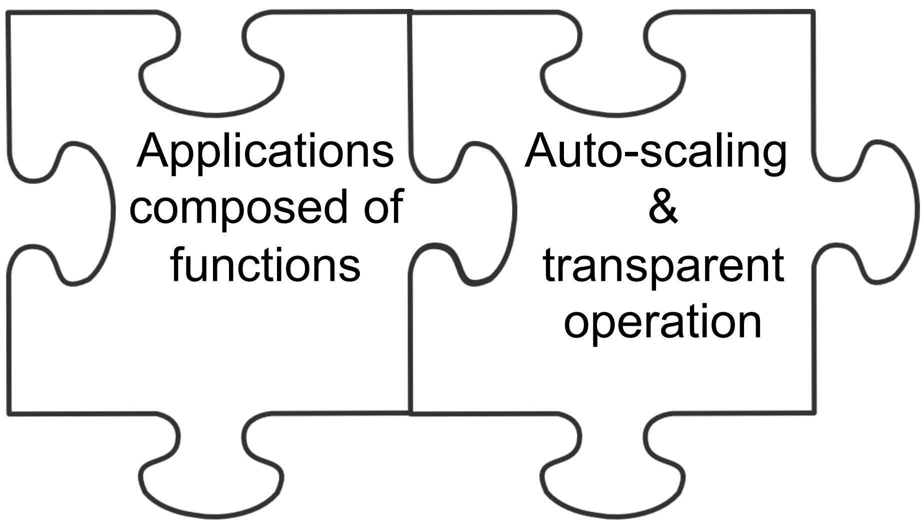
@MShahrad

https://mshahrad.github.io/

PRINCETON UNIVERSITY

PRINCETON School of Engineering and Applied Science

Function-as-a-Service (FaaS)
Serverless Computing



Amazon Lambda

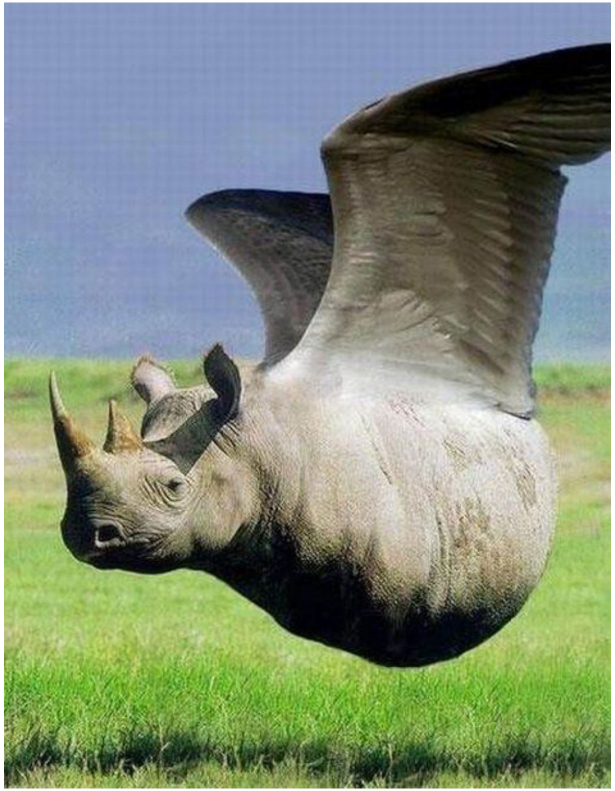
Azure Functions

Google Cloud Functions

IBM Cloud Functions

FaaS is like a flying rhino!

- Neither a bird (native function)
  - Too much overhead compared to native function execution
- Nor a rhino (VM)
  - Being small and short-lived makes them hard to provision



Source: https://www.flickr.com/photos/1grandpoobah/7902346828/in/photostream/

FaaS Differs From Prior Cloud Offerings

Just a few:

- Short function executions
- High concurrency (with inefficient isolation)
- Fine-grained pricing based on execution time, memory, and request counts
- Developer has less control on provisioning

Prior Work

External characterization and reverse-engineering

Building new applications / mapping existing applications

Better isolation/virtualization mechanisms
(safe containers, light virtual machines)

Our Initial Goal: New Architectural Features to Better Support FaaS

No Benchmark

↓

Let's gather some!

No Clear Testing & Profiling Methodology

↓

Let's build one!

Too Complex of a System

↓

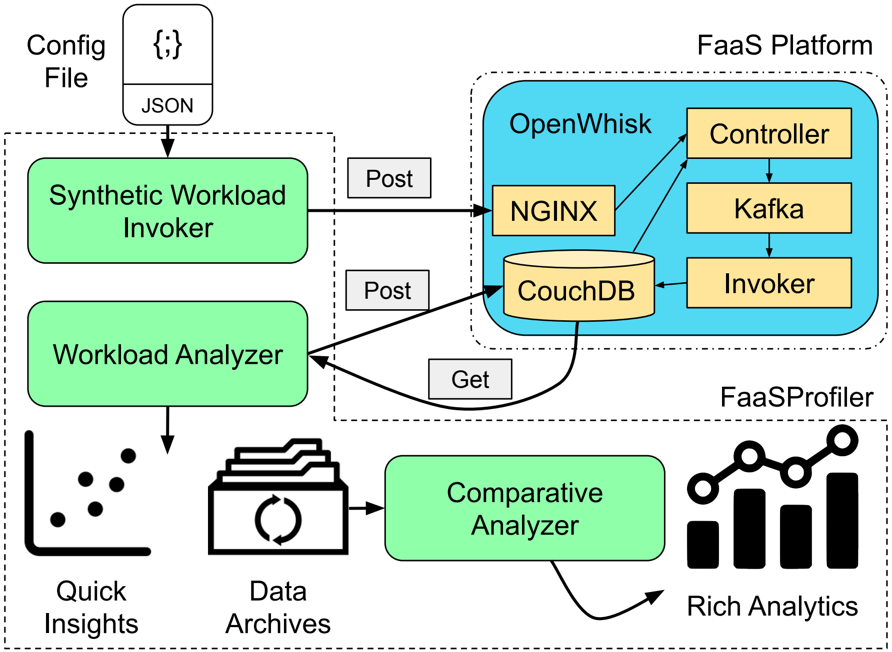
Let's try to understand it first!

Diving deep into an open source serverless platform.

- A complete open-sourced industry-grade (IBM) FaaS platform
- Functions run in containers
- Functions can be in Python, Node.js, Scala, Java, Go, Ruby, Swift, PHP, .Net, and Rust
- Or the developer can provide a Docker container

APACHE OpenWhisk™

We built FaaSProfiler for testing and profiling.



- Automated function invocations (single JSON file):
  - Synthetic distributions
  - Specified traces
- Uses standard profiling tools:
  Perf, PQoS, Blktrace, etc.
- Easy analysis and comparison

https://github.com/PrincetonUniversity/faas-profiler

Benchmarks and Test Setup

Benchmarks:

- 5 representative applications
- 28 Python microbenchmarks

Test server:

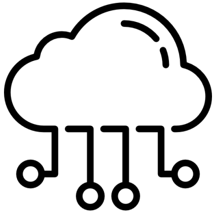
- Intel Xeon E5-2620 v4
- 8-cores, 16-threads
- 20MB Last-Level Cache
- 16GB 2133MHz DDR4 (single-channel)

FaaS Benchmark	Runtime
`autocomplete`	NodeJS
`markdown-to-HTML`	Python
`img-resize`	NodeJS
`sentiment-analysis`	Python
`ocr-img`	NodeJS + binary

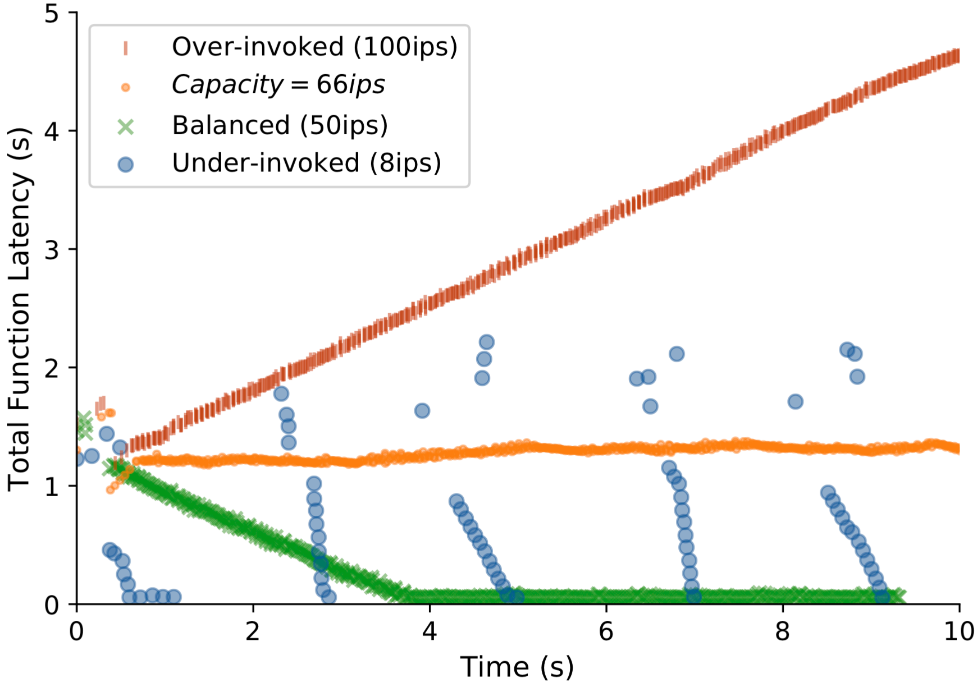
Understanding The Performance Criteria

For native functions, <u>execution time</u> is an accepted measure of performance.

How about for FaaS functions?

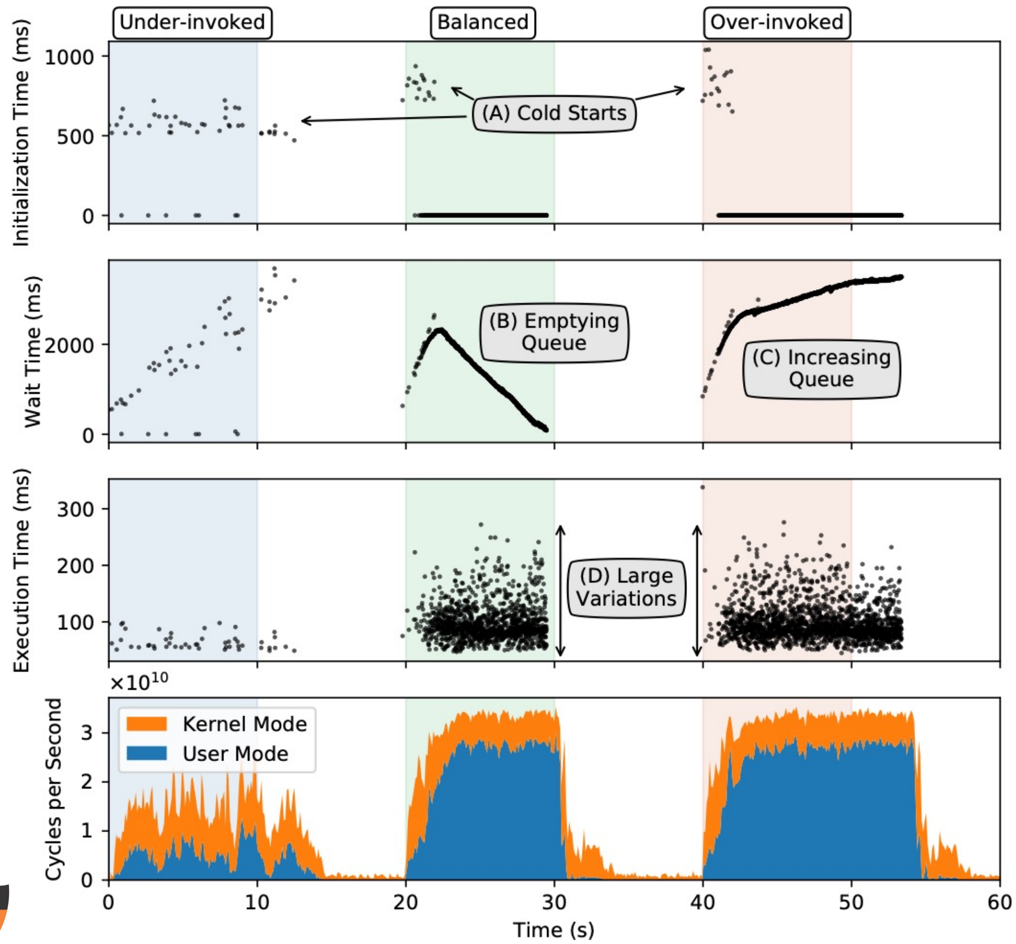
Execution Time	Latency	Throughput
		
Developer	End User	Provider

Server Capacity & Latency Modes



Over-invoked	Input > Capacity
Capacity	Input = Capacity
Balanced	Input < Capacity
Under-invoked	Input << Capacity

Breakdown of Latency



1. Initialization Time

2. Wait Time in the Queue

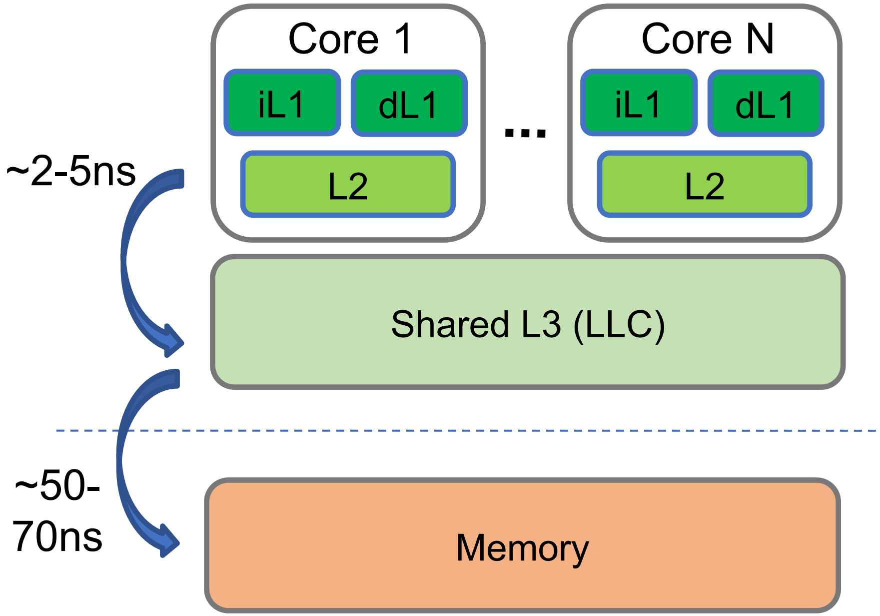
3. Execution Time

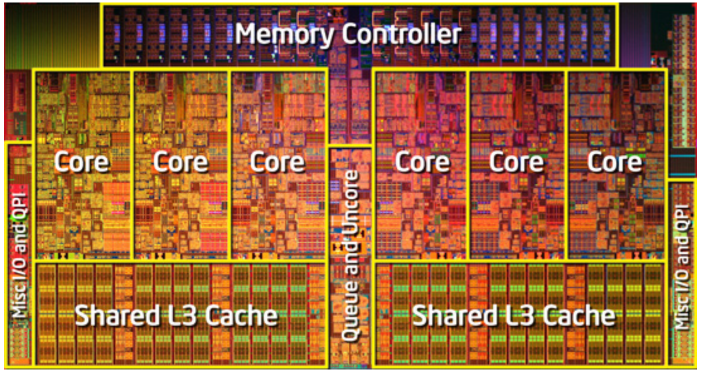
Non-trivial overhead

Interesting Architectural Findings

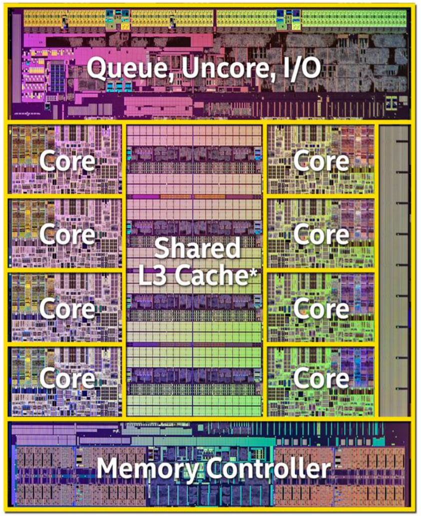
1. Last-level Cache (LLC) Requirement
2. Branch Prediction
3. Memory Bandwidth Consumption

Last-Level Cache (LLC)





https://www.anandtech.com/show/2960/2



http://www.guru3d.com/articles-pages/core-i7-5960x-5930k-and-5820k-processor-review,2.html

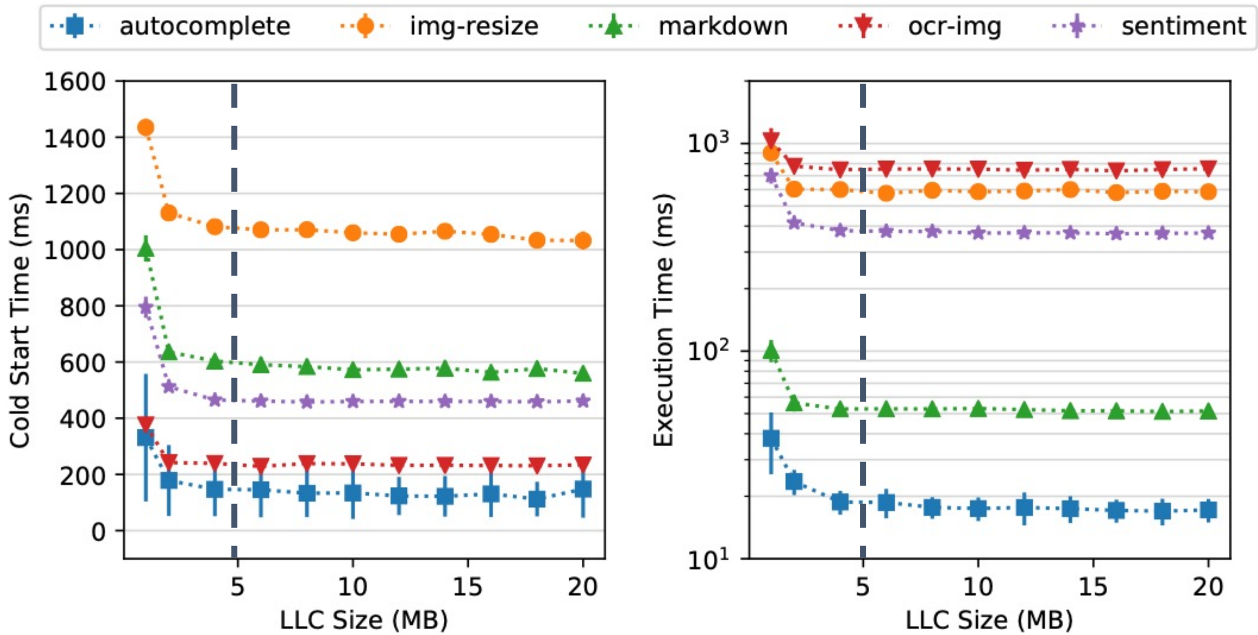
Critical for Performance

Vary Expensive (SRAM on CPU)

We observed low LLC requirements.

Used Intel Cache Allocation Technology (CAT).

Low Demand from Platform Components



Low Demand from Functions

Others have also reported decreasing LLC requirement for emerging cloud workloads.

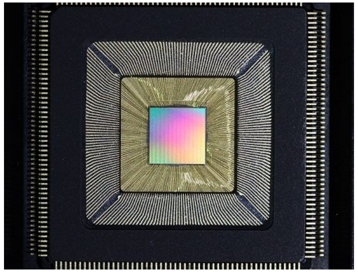
- Scale out workloads [Ferdman et al., ASPLOS ‘12]
- Latency-critical cloud workloads [Chen et al., ASPLOS ‘19]
- Microservices [Gen et al., ASPLOS ‘19]

Short-term Opportunity

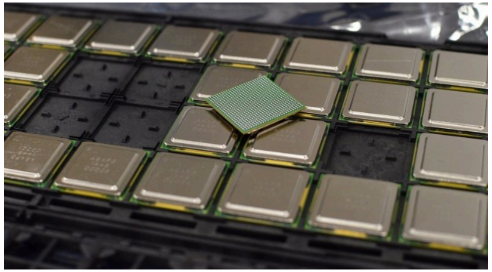
Partition the LLC in favor of cache-sensitive workloads.

Long-term Opportunity

More cores, less LLC



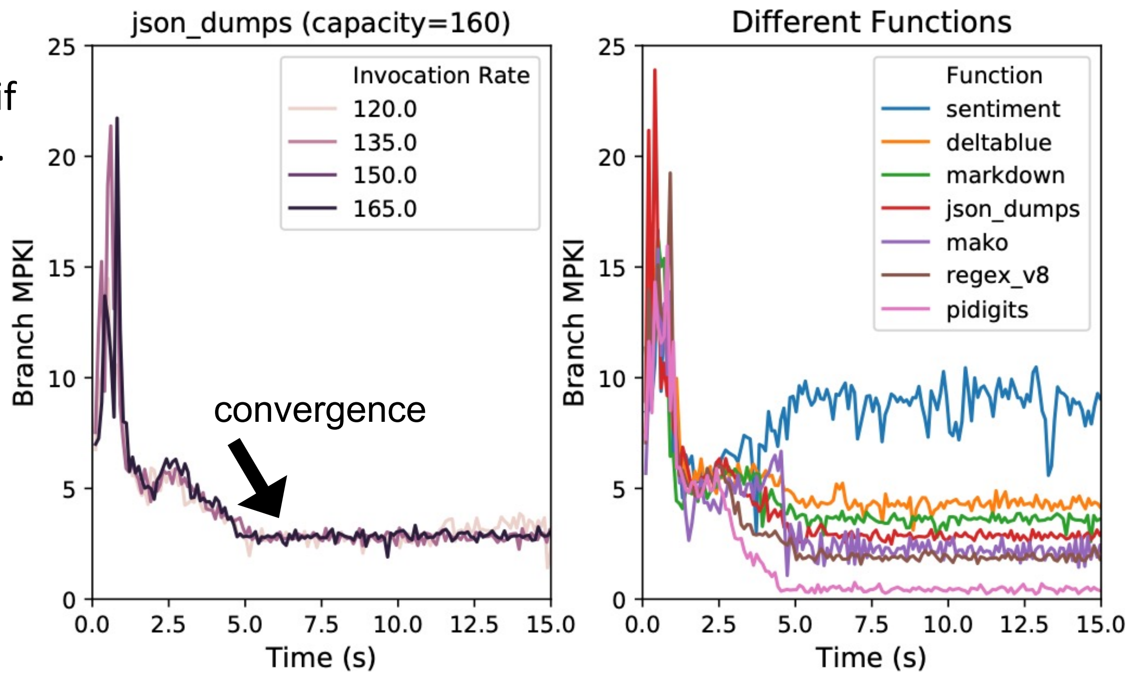
Princeton Piton Processor



UC Davis KiloCore

Branch Prediction Performance

MPKI does not vary with invocation rate if containers kept alive.

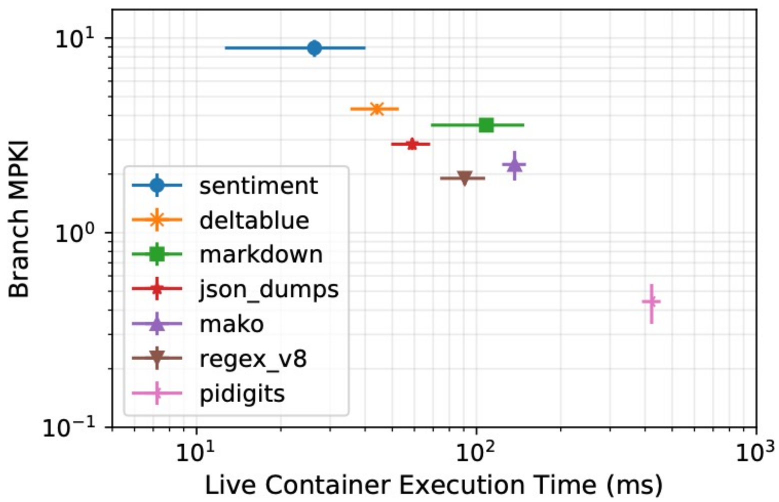


Functions have a distinct behavior.

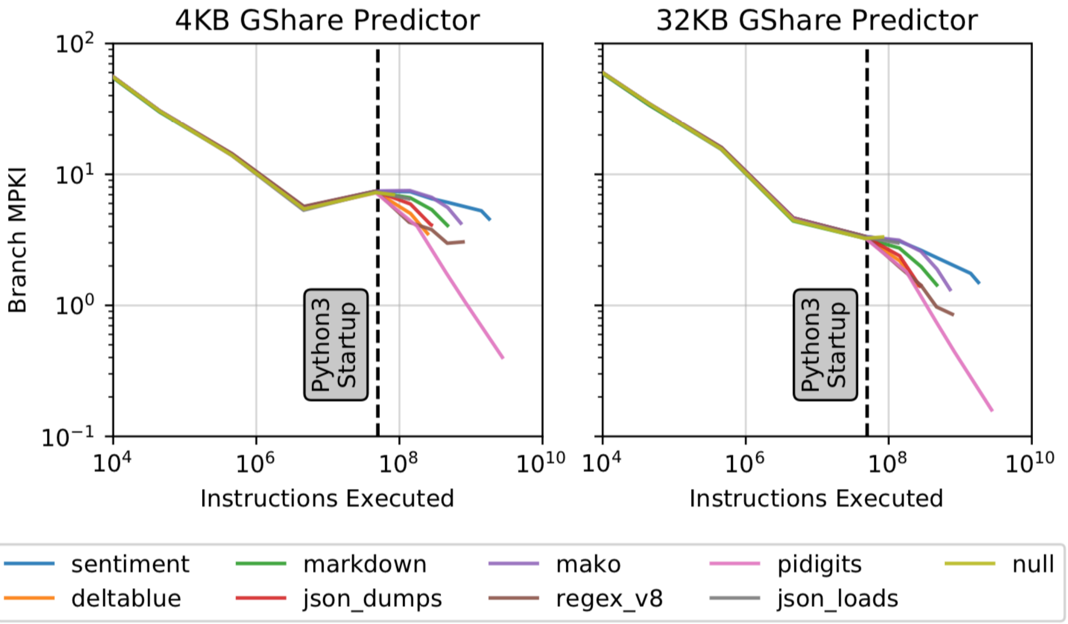
MPKI: Misses per Kilo Instructions

Longer execution helps with branch misses.

Shorter functions have higher MPKI.
[~20x variations]



Simulations revealed the reason.



Short FaaS Function Lifetimes vs. Conventional Microarchitectural Expectation

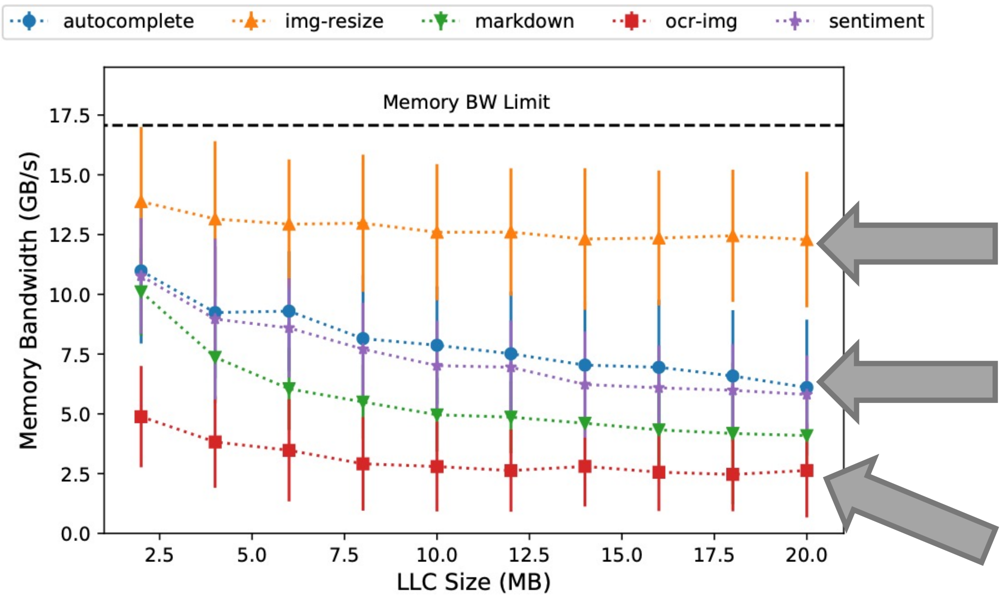
- Conventional expectation: programs run for long enough to train the predictors.
- Short deeply-virtualized functions are not a good fit to this model.

Opportunity

Revised branch predictors for:

- Retaining prediction states at the container- or application-level
- Faster training

Memory Bandwidth Consumption



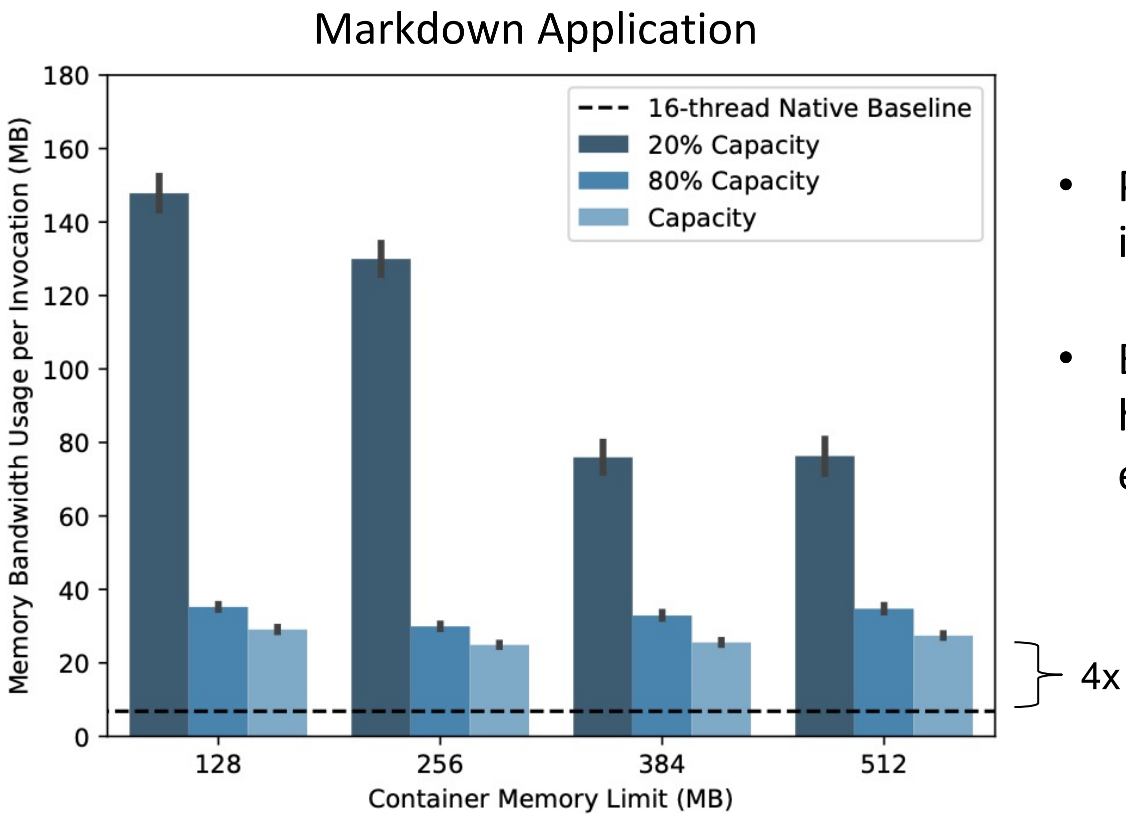
- Heavy payload
- Short execution time

- Light payload
- Very short execution time

- Medium payload
- Long execution time

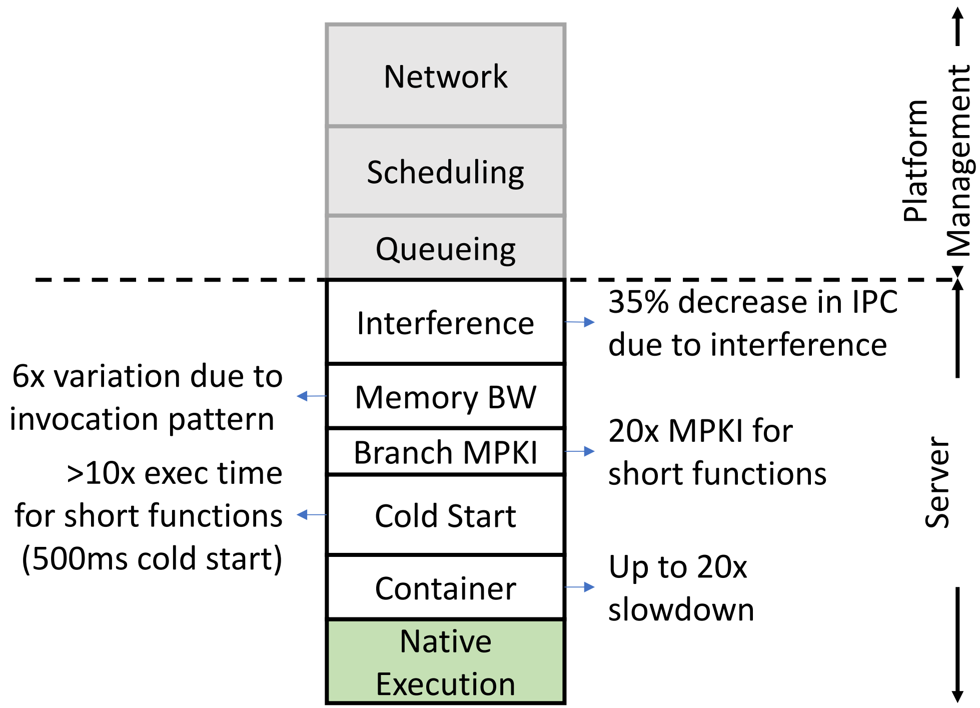
Various demands make it hard to co-locate.

Per-Invocation Memory Bandwidth Usage



- Pausing/unpausing containers increases the bandwidth usage
- Bandwidth usage noticeably higher compared to native executions

The server behavior should be carefully taken into account when designing new services.



Paper PDF

FaaSProfiler

PRINCETON UNIVERSITY

PRINCETON
School of Engineering and Applied Science