Architectural Implications of Function-as-a-Service Computing

Micro 19’

Authors: Mohammad Shahrad, Jonathan Balkind, David Wentzlaff

Motivation of this investigation

- Short-lived function causes locality-exploiting structures like branch predictors to underperform
- Deeply-virtualized environment comes with overheads
- FaaS platform brings new, unforseen overheads and challenges for architectural optimization

Takeaways

- FaaS containerization brings up to 20x slowdown compared to native execution
- cold-start can be over 10x a short function’s execution time
- branch mispredictions per kilo-instruction are 20x higher for short functions
- memory bandwidth increases by 6x due to the invocation pattern
- PC decreases by as much as 35% due to inter-function interference

Overheads breakdown

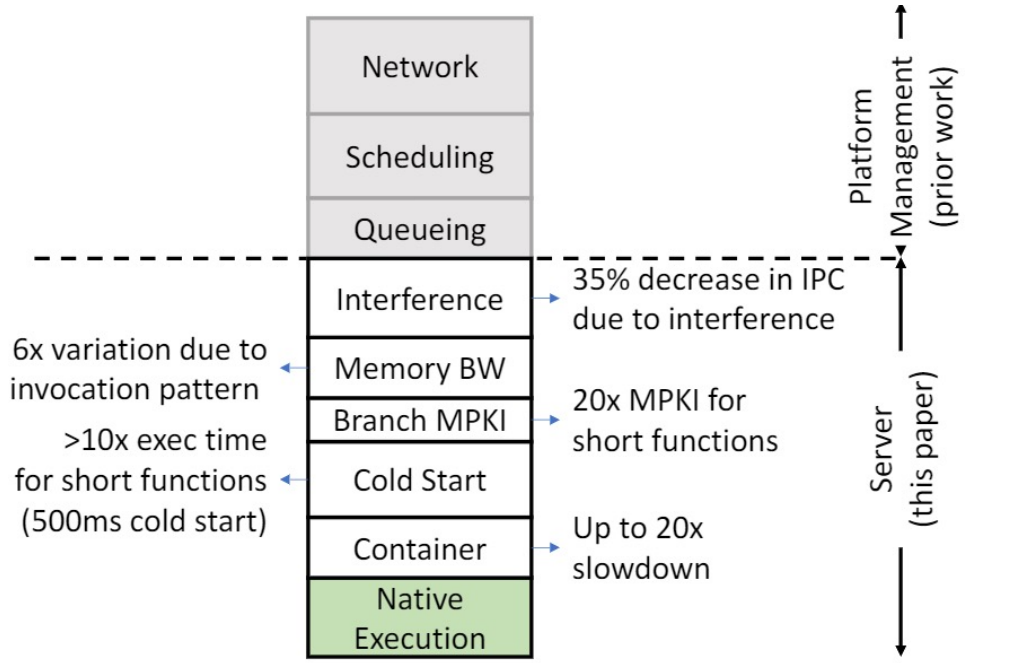


Figure 1: We characterize the server-level overheads of Function-as-a-Service applications, compared to native execution. This contrasts with prior work [2–5] which focused on platform-level or end-to-end issues, relying heavily on reverse engineering of commercial services' behavior.

Experiment environment

CPU: Intel Xeon E5-2620 v4, 8-core, 16-thread, 20MB of L3 cache.

Memory: 16GB of 2133MHz DDR4 RAM connected to a single channel

Platform: Apache OpenWhisk

OS: Ubuntu 16.04.04 LTS

FaaSProfiler

- Arbitrary mix of functions and invocation patterns
- Large amount of performance and profiling data

PaaSProfiler and Open Whisk

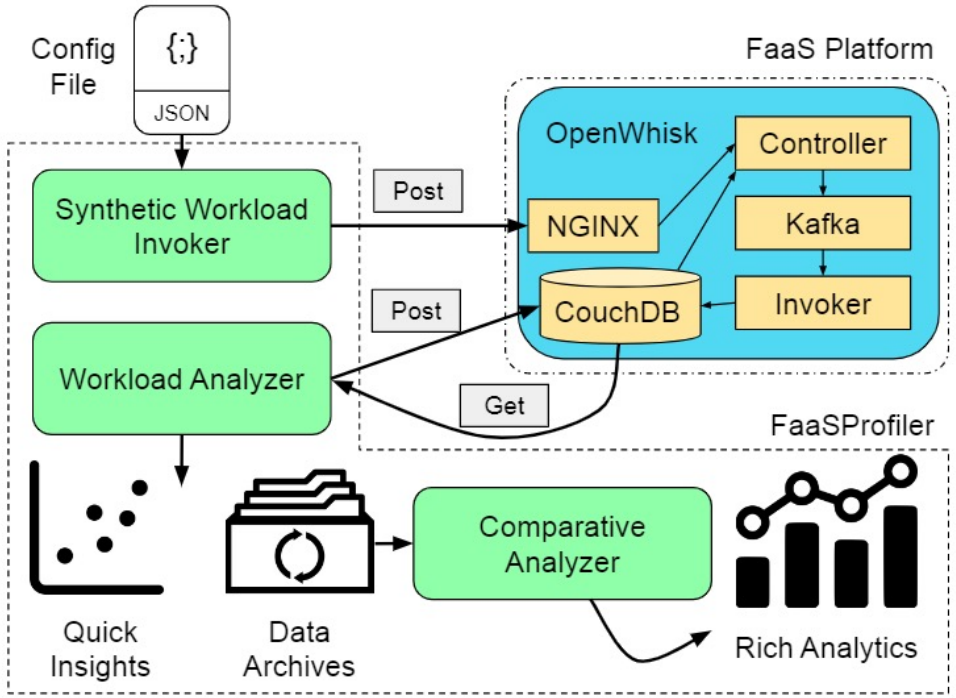


Figure 3: We build FaaSProfiler which interacts with Open-Whisk to run controlled tests and to profile various metrics.

Example Workload Configuration JSON

```
{
    "test_name": "sample_test",
    "test_duration_in_seconds": 25,
    "random_seed": 110,
    "blocking_cli": false,
    "instances":{
        "instance0":{
            "application": "function0",
            "distribution": "Poisson",
            "rate": 20,
            "activity_window": [5, 20]
        },
        "instance1":{
            "application": "function0",
            "distribution": "Uniform",
            "rate": 100,
            "activity_window": [10, 15]
        },
        "instance2":{
            "application": "function1",
            "data_file": "~/image.jpg",
            "distribution": "Poisson",
            "rate": 15,
            "activity_window": [1, 24]
        }
    },
    "perf_monitoring":{
        "runtime_script": "MonitoringScript.sh"
    }
}
```

Benchmarks

- Subset of Python Performance Benchmark Suite
- Five Faas functions

Five Faas functions

Application	Description	Runtime
autocomplete	Autocomplete a user string from a corpus	NodeJS
markdown	Renders Markdown text to HTML	Python
img-resize	Resizes an image to several icons	NodeJS
sentiment	Sentiment analysis of given text	Python
ocr-img	Find text in user image using Tesseract OCR	NodeJS + binary

Latency Modes and Server Capacity

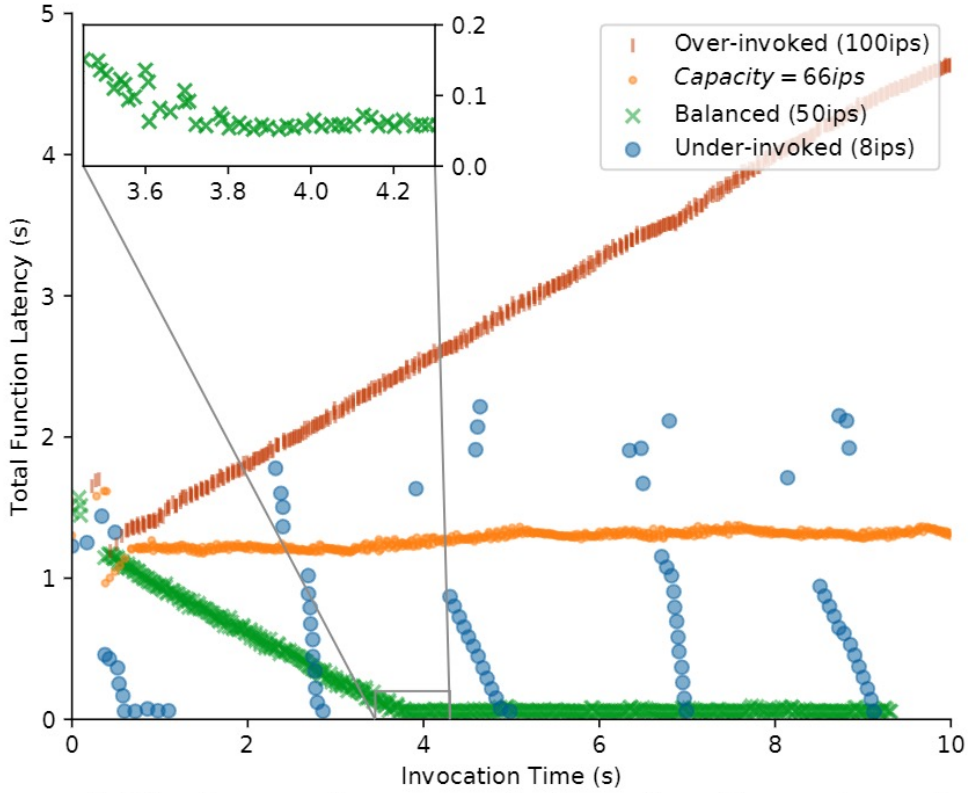


Figure 4: The invocation rate of a function (`json_dumps` here) determines its latency behavior. (Lower latency is better)

Latency Breakdown

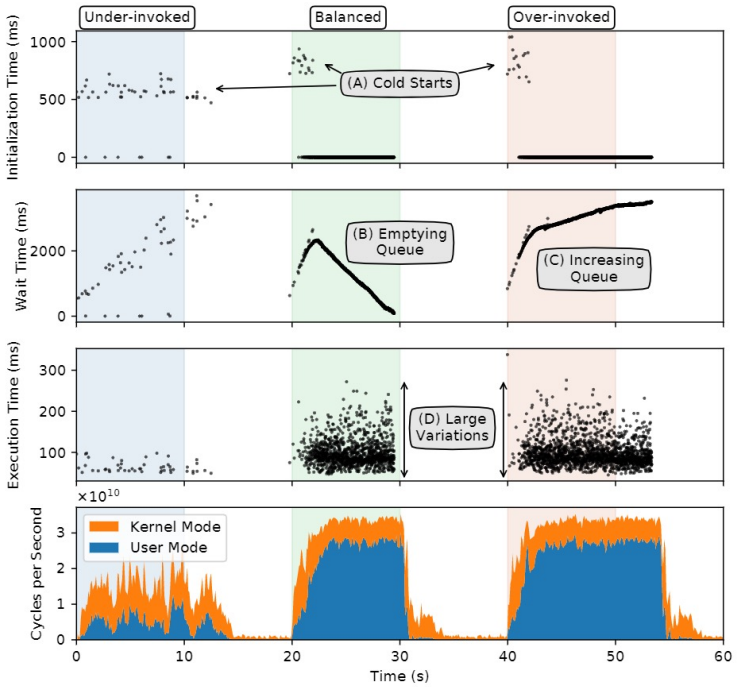


Figure 5: Function `json_dumps` invoked with different rates to study the latency modes. The breakdown of latency (made of initialization time, wait time, and execution time) is shown, as are the cycles per second spent in user and kernel modes. Kernel overhead is especially high for the under-invoked case.

Branch prediction

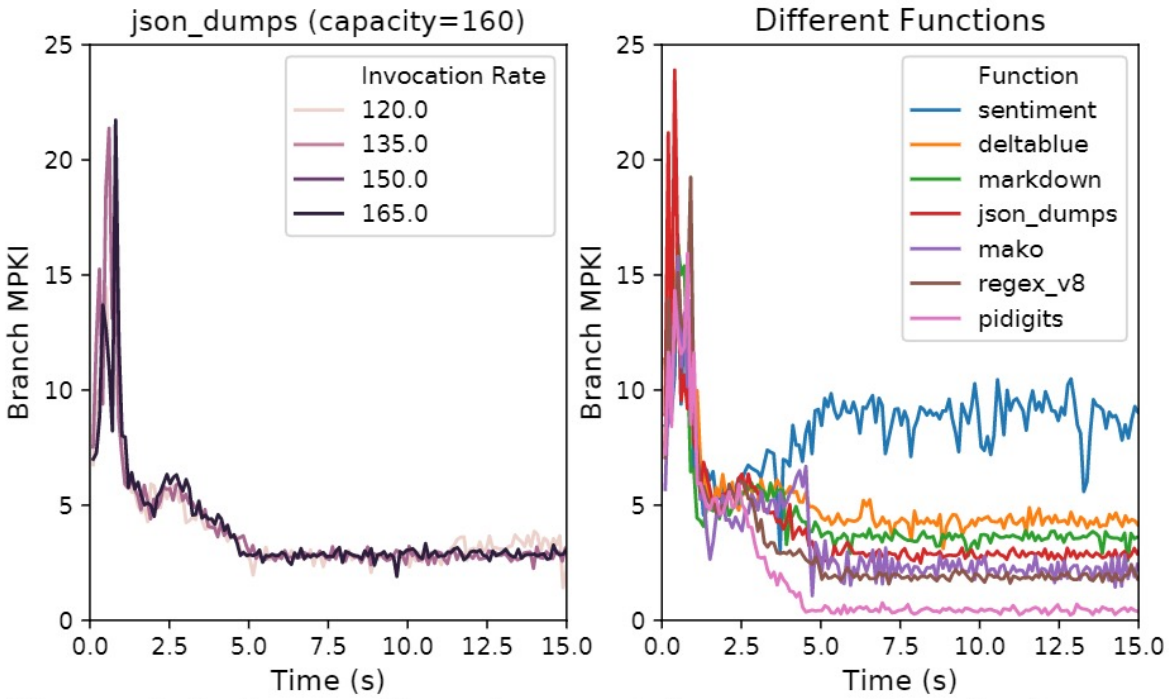


Figure 6: As long as function containers are alive, the invocation rate does not affect the branch MPKI (left). However, different functions consistently experienced different branch MPKI (right).

Branch prediction

Figure 7: Functions with longer execution time have lower branch MPKI.

Branch prediction

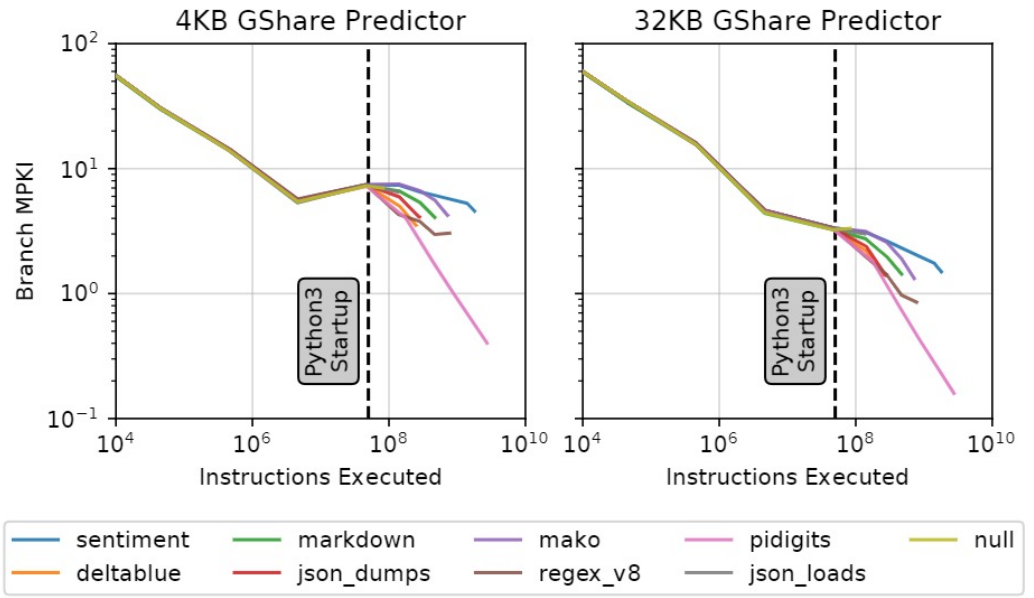


Figure 8: Simulated branch misprediction per kilo-instruction (MPKI) rates for 4KB and 32KB GShare predictors compared to execution time. Python startup overhead is significant for short functions, and MPKI is 16.18x (32KB) and 18.8x (4KB) higher for the shortest function compared to the longest.

Last-level Cache

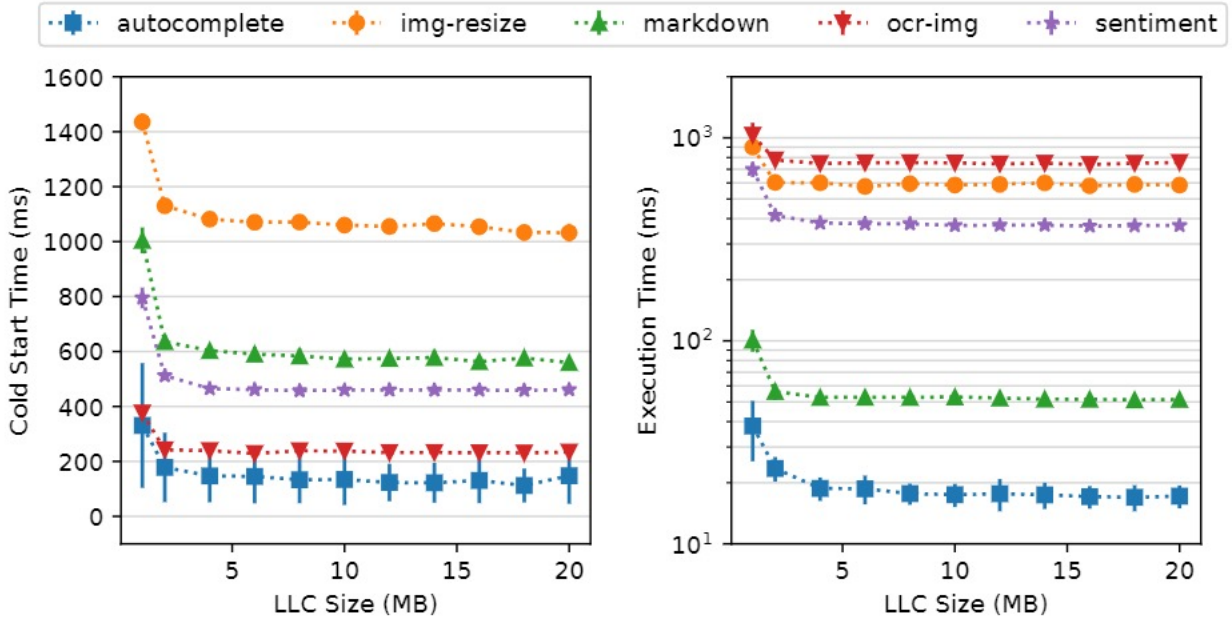
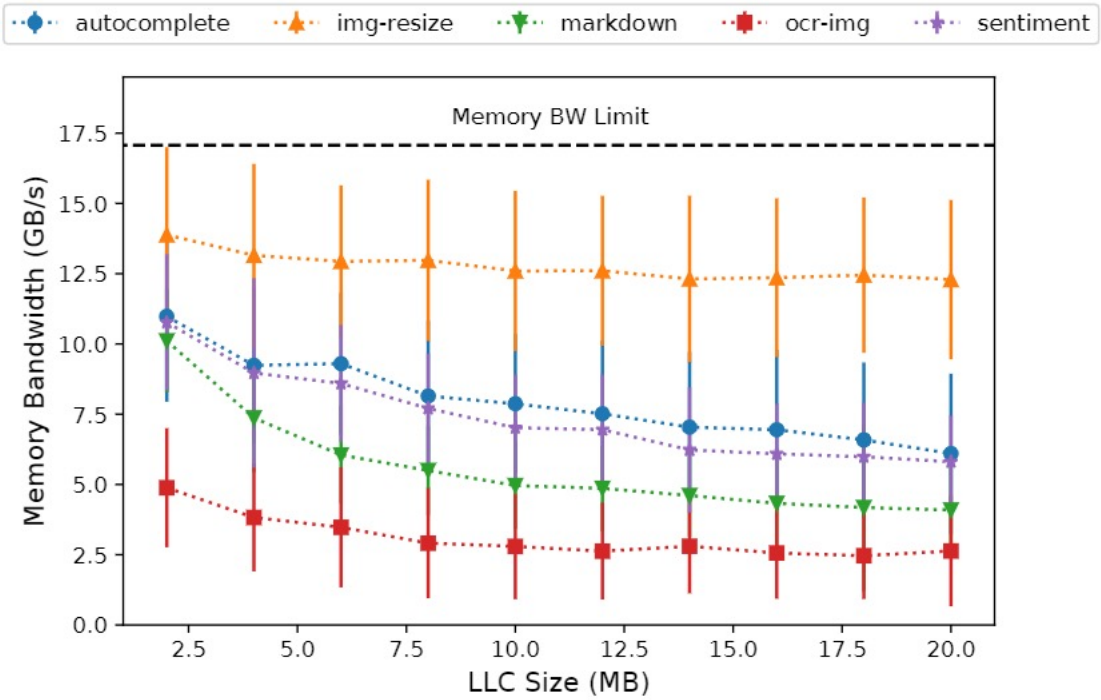


Figure 9: When invoked to cause cold starts, increasing LLC size beyond around 2MB does not significantly change the cold start latency and execution time of five functions.

Memory Bandwidth



Memory Bandwidth

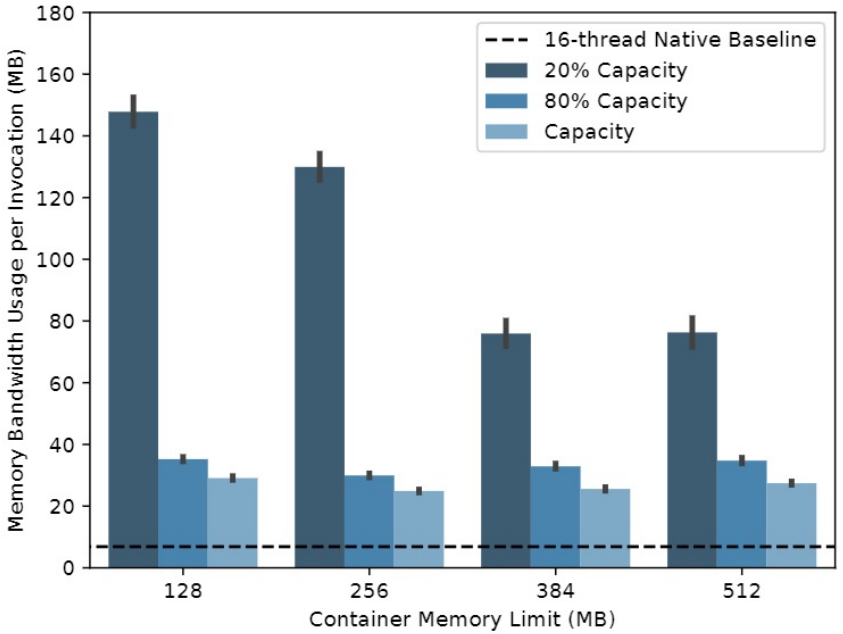


Figure 11: Under-invocation leads to cold starts, increasing the normalized memory bandwidth consumption per invocation. This behavior is independent of memory limit. We invoked different variants of `markdown` with their corresponding capacities to show this.

Invoker Scheduling

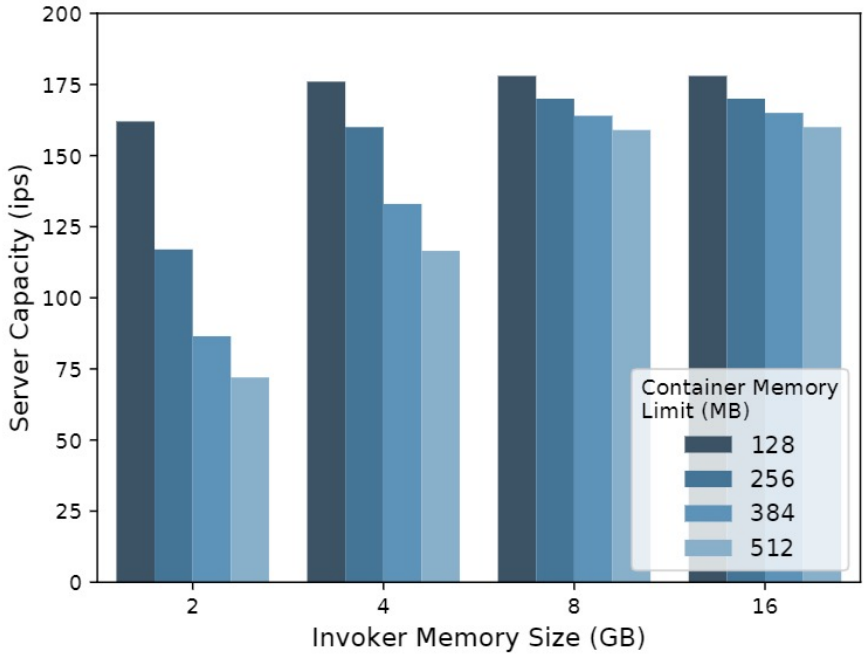


Figure 12: Invoker memory size determines the number of parallel containers, affecting the server capacity (`json_dumps` here). However, this parallelism has overheads which limit the capacity gains.

Invoker Scheduling

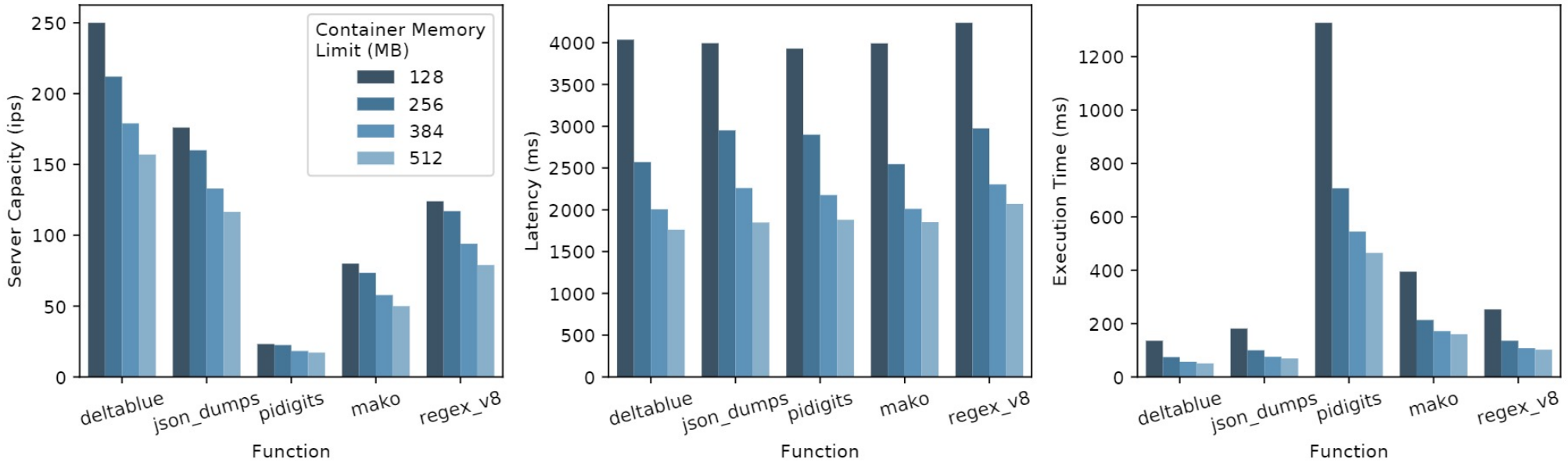


Figure 13: When invoked at capacity, reducing each function's memory limit from 512MB to 384MB and 256MB increases the latency and execution time roughly proportionally. However, decreasing the memory limit to 128MB increases capacity only slightly, while costing much more in latency and execution time.

Interference

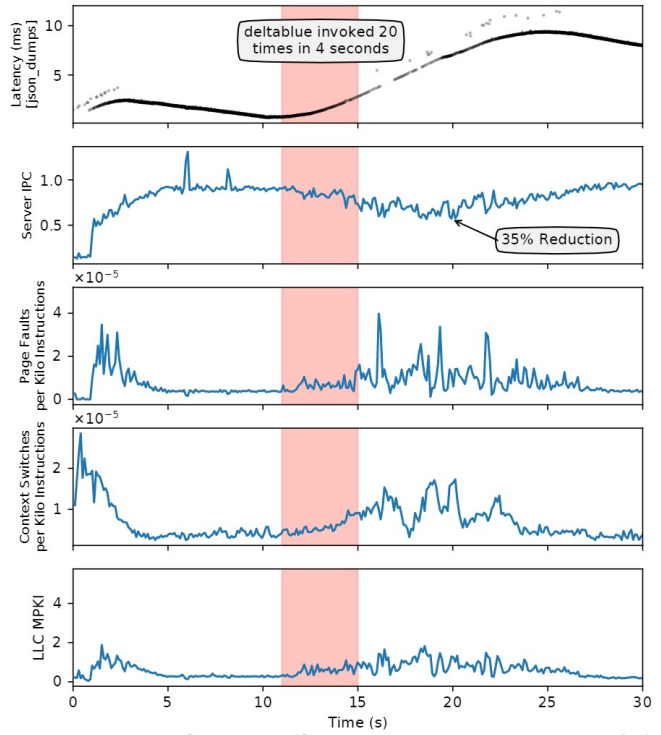


Figure 14: Interference effects are severe in FaaS. While `json_dumps` is invoked at 80% of its capacity (128ips), just 20 `deltablue` invocations in 4 seconds cause lingering effects that reduce IPC and increase page faults, context switches, and cache misses per kilo-instruction.

Overhead of Containers

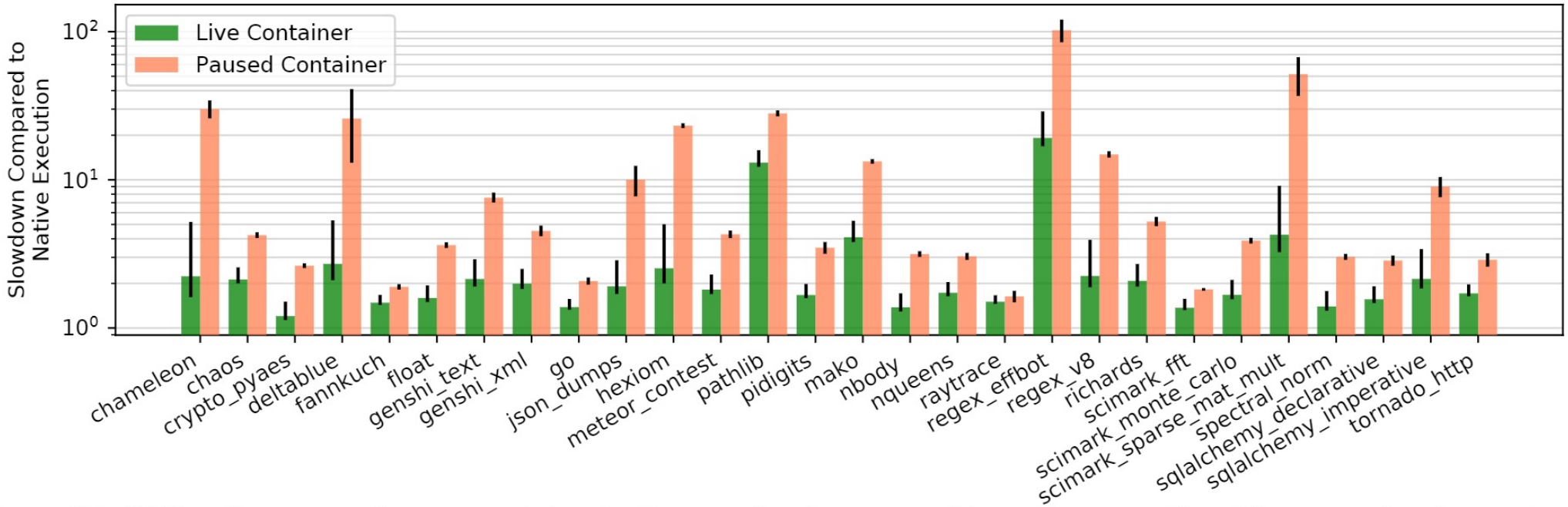


Figure 15: All functions experience a containerization overhead compared to native execution. These overheads are significantly higher for paused containers than for live containers. (Slowdown plotted as log, lower is better)

Overhead of Containers

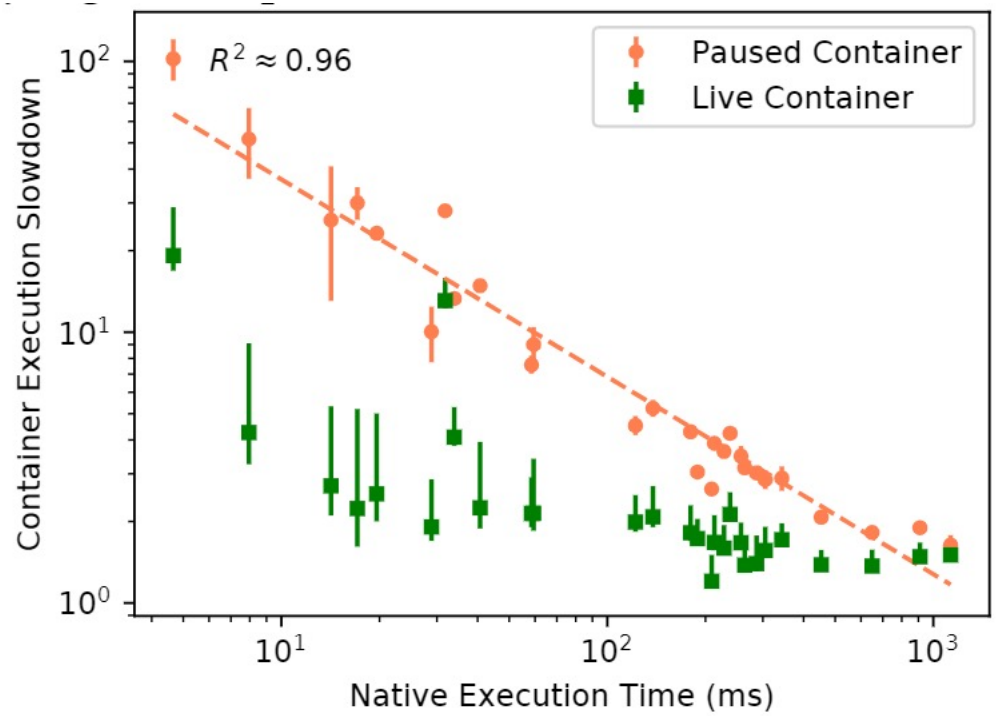


Figure 16: Shorter functions experience relatively higher slowdown. (Log-log plot, lower is better)