Architectural Implications of Function-as-a-Service Computing

Mohammad Shahrad
Princeton University
Princeton, USA
mshahrad@princeton.edu

Jonathan Balkind
Princeton University
Princeton, USA
jbalkind@princeton.edu

David Wentzlaff
Princeton University
Princeton, USA
wentzlaf@princeton.edu

ABSTRACT

Serverless computing is a rapidly growing cloud application model, popularized by Amazon's Lambda platform. Serverless cloud services provide fine-grained provisioning of resources, which scale automatically with user demand. Function-as-a-Service (FaaS) applications follow this serverless model, with the developer providing their application as a set of functions which are executed in response to a user- or system-generated event. Functions are designed to be short-lived and execute inside containers or virtual machines, introducing a range of system-level overheads. This paper studies the architectural implications of this emerging paradigm. Using the commercial-grade Apache OpenWhisk FaaS platform on real servers, this work investigates and identifies the architectural implications of FaaS serverless computing. The workloads, along with the way that FaaS inherently interleaves short functions from many tenants frustrates many of the locality-preserving architectural structures common in modern processors. In particular, we find that: 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, and IPC decreases by as much as 35% due to inter-function interference. We open-source FaaSProfiler, the FaaS testing and profiling platform that we developed for this work.

CCS CONCEPTS

• **Information systems** → **Enterprise resource planning**; **Computing platforms**; • **Computer systems organization** → **Architectures**.

KEYWORDS

serverless, function-as-a-service, faas, cloud, OpenWhisk, architecture

ACM Reference Format:
Mohammad Shahrad, Jonathan Balkind, and David Wentzlaff. 2019. Architectural Implications of Function-as-a-Service Computing. In *The 52nd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO-52), October 12–16, 2019, Columbus, OH, USA.* ACM, New York, NY, USA, 13 pages. https://doi.org/10.1145/3352460.3358296

MICRO-52, October 12–16, 2019, Columbus, OH, USA

ACM ISBN 978-1-4503-6938-1/19/10…$15.00
https://doi.org/10.1145/3352460.3358296

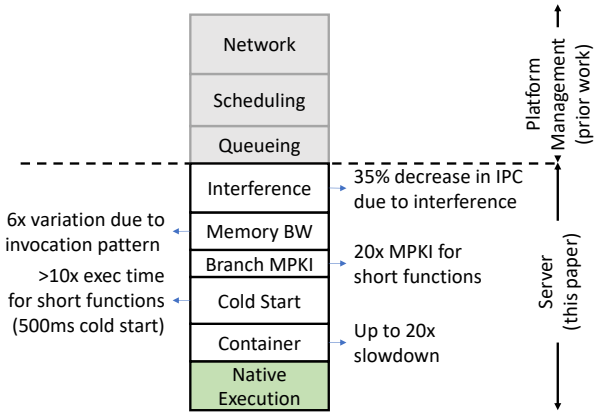

Figure 1: We characterize the server-level overheads of Function-as-a-Service applications, compared to native execution. This contrasts with prior work [2–5] which focused on platform-level or end-to-end issues, relying heavily on reverse engineering of commercial services' behavior.

1 INTRODUCTION

Serverless computing is a relatively new paradigm in cloud computing, first launched by Amazon's AWS Lambda [1] in November 2014. In serverless, the cloud provider manages the provisioning of resources for a service in a transparent, auto-scaling manner, without the developer in the loop. The developer is charged in a fine-grained way, proportional to the resources provisioned and used per request, with no cost levied when the service is idle (receiving no requests).

Function-as-a-Service (FaaS) is a key enabler of serverless computing. FaaS gets its name as it is a way to scale the execution of simple, standalone, developer-written functions, where state is not kept across function invocations. Functions in FaaS are event-driven, invoked by a user's HTTP request or another type of event created within the provider's platform. To match the rate of function invocations, the platform automatically scales the resources available to execute more instances of the developer's function in parallel.

The majority of serverless research has targeted mapping applications to this model, or has focused on scheduling policies or mechanisms for orchestrating FaaS setups. However, we are interested to know if the computational building blocks of FaaS, namely, commodity servers used in modern cloud data centers are suitable for such workloads. **FaaS applications differ from traditional cloud workloads in ways that may jeopardize common architectural wisdom**.

Firstly, FaaS functions are typically short-lived and priced in multiples of a 1ms, 10ms, or 100ms time quantum. Function execution time is usually also capped. Providers must rely on the

fine-grained interleaving of many short functions to achieve high throughput. **This fine-grained interleaving causes temporal locality-exploiting structures like branch predictors to underperform, raising questions about present-day computer architectures' ability to execute FaaS efficiently.** For instance, we observe a 20x increase in branch mispredictions per kilo-instruction (MPKI) when comparing our shortest functions to longer functions. This finding and others are shown in Figure 1.

Secondly, functions often run in a deeply virtualized manner: inside containers which might run within virtual machines. This causes a number of OS-level overheads. We find that flat overheads like the cold start latency of 500ms or more have an outsized impact on short functions.

Lastly, FaaS developers can write their functions in almost any language. Each platform has preferred languages, but developers may upload their own container to run a chosen application in the FaaS environment. Together, these features make for a highly dynamic system, with many, fast-changing components, which we have found bring new, unforeseen overheads and challenges for architectural optimization.

We provide the first server-level characterization of a Function-as-a-Service platform by carefully investigating a compute node of an open-source FaaS platform, Apache OpenWhisk (sold commercially as IBM Cloud Functions [6]). This contrasts with previous work solely focusing on black-box reverse-engineering of commercial FaaS systems [2–5]. With full control of our deployment, we precisely induce function co-location and access hardware performance counters, something impossible for an external black-box analysis. Our bottom-up approach enables us to decouple overheads in managing the cloud platform from the overheads caused in the server context. This reveals the architectural implications of FaaS computing.

We identify a range of new overheads that affect FaaS execution with short functions affected most acutely. Figure 1 summarizes these overheads in the context of the full stack of overheads.

Our contributions are as follows:

- We perform the first server-level characterization of a Function-as-a-Service platform to determine architectural and microarchitectural impacts.
- We develop and open-source FaaSProfiler, a new FaaS testing platform. This enables other researchers to conduct their own profiling with OpenWhisk. We also open-source benchmarks, measurement data, and scripts used in this paper.
- We characterize and attribute the individual contributions to server-level overheads, including containerization (20x slowdown), cold-start (>10x duration), and interference (35% IPC reduction) overheads.
- We discover that FaaS workloads can cause certain architectural components to underperform. This includes 20x more MPKI for branch predictors and 6x higher memory bandwidth under some invocation patterns.
- We identify a mismatch in demands between the FaaS provider, developer, and service end user, with respect to the present pricing model. This motivates further research into pricing, Service-Level Agreements (SLAs), and architectural methods to alleviate the mismatch.

2 BACKGROUND

The ideas behind serverless and FaaS have developed over a long period [7], but only formed into commercial services within the last few years. FaaS takes lessons from Infrastructure-as-a-Service (IaaS), Platform-as-a-Service (PaaS), and microservices, but brings with it completely new pricing, development, and provisioning models.

2.1 Differences with Other Cloud Models

FaaS is unlike traditional cloud models like IaaS and PaaS, and has significant differences with newer models such as microservices which have recently been thoroughly studied [8–10]. Differences of FaaS from prior models include:

- The FaaS developer does not provision or manage the servers that functions run on. This means there is no management of compute bottlenecks by developers.
- Functions are abstracted from machine type. Most providers do not guarantee a machine type, nor do they price differentially based on them.
- Functions are used to increase server utilization. As such, functions are often in contention with traditional applications using the same server.
- FaaS is priced at a fine granularity, based on resource utilization, not per API call. FaaS brings cloud computing closer to a compute-as-utility model.
- The developer is only charged for their function's execution time based on memory usage. This ignores platform (e.g. network, cluster-level scheduling, queuing) and system-level details (e.g. container management and OS scheduling) and motivates the provider to minimize these overheads.
- Similar to microservices, functions are typically very short (O(100ms)). This is in contrast to common cloud and processor architectures that expect long-running applications. The overheads of handling these short functions can further worsen the problem of low utilization in cloud servers [11–13]. Moreover, due to high parallelism, performance interference effects can be amplified.
- FaaS providers give SLAs only for availability.

The Cloud Native Computing Foundation (CNCF) divides serverless into FaaS and Backend-as-a-Service (BaaS) [14]. They define BaaS as "third-party API-based services that replace core subsets of functionality in an application. Because those APIs are provided as a service that auto-scales and operates transparently, this appears to the developer to be serverless." Many storage platforms, microservices, and traditional, more monolithic, cloud platforms are as such considered BaaS, because scaling, resource-provisioning, and operation are transparent to the API end-user. However, today they likely are not implemented using or priced like FaaS.

Isolation. FaaS generally makes use of containers provided by the developer (usually via a Docker [15] file) to run and handle each request. The provider may hide this for simple use cases, providing a portal to upload source files. Containers lack typical cloud isolation therefore many platforms run containers inside virtual machines, leading to higher overheads.

Metrics. The FaaS provider is unaware of functions' application-level performance metrics beyond throughput and execution time. As such, they do not guarantee metrics like 99th-percentile latency

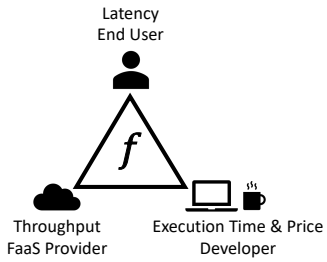

Figure 2: In FaaS, the stakeholders' demands directly compete. The end user desires low latency, the developer desires low execution time and thus cost, and the FaaS provider desires high throughput across their services.

as in more controlled settings like PaaS or microservices. This exposes the developer to platform-level behavior like high cold start times. This has motivated previous studies of the internals of FaaS platforms, invoking large numbers of different functions in different settings, in order to reverse-engineer the underlying system [3–5]. We go beyond this prior work, looking inside OpenWhisk as the provider to further break down the causes of variance in throughput, latency, and execution time.

Providers have just begun to give SLAs for FaaS [16]. These only cover uptime, falling in line with broader cloud SLAs at 99.95%. Beyond this, providers are making few guarantees to developers.

Pricing. FaaS is priced differently to existing cloud services, depending only on each function invocation's execution time and the function's memory limit (most often statically set by the developer). Charging only for compute time enables new applications that rely on massive parallelism to produce results at a high speed and low cost [17–19].

FaaS is priced in gigabyte-seconds (GB-s), with some providers having modifiers of gigahertz-seconds (GHz-s) or total invocations. A gigabyte-second is one second of execution, where the function is provided 1 gigabyte of memory. Usually a function has a static memory limit set in advance (of at least its true demand). In some cases, an increased memory limit entails a faster processor allocation (hence the additional gigahertz-seconds). With Apache OpenWhisk, a higher memory limit allocates more CPU time. The developer is charged for the time their functions execute, ignoring system-level processing, and container/VM setup or teardown.

Demands. In FaaS, the system's three stakeholders have different demands, as shown in Figure 2. For return on investment, the provider prefers high throughput, raising revenue per unit time and the utilization of their cloud infrastructure. The developer values low execution time most, reducing their compute cost. The end user, who makes the function invocations, wants low latency to run their application smoothly. We find that in our FaaS deployment, these three metrics are in competition, raising an economic quandary for the FaaS provider and questions about the effects of the present FaaS pricing model on the developer and end user.

2.2 Apache OpenWhisk

To analyze FaaS in depth, we need the visibility to understand system components and decouple software from architectural overheads. An open-source FaaS platform enables us to do this. To

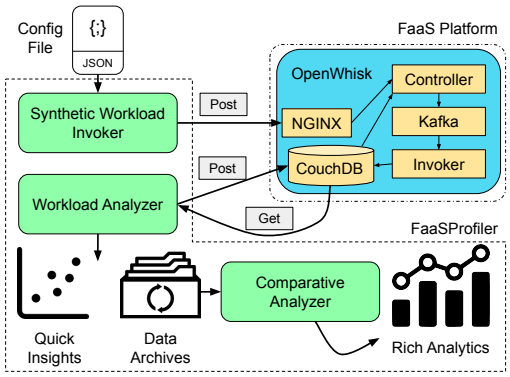

Figure 3: We build FaaSProfiler which interacts with OpenWhisk to run controlled tests and to profile various metrics.

maximize our results' relevance, we seek to study commercial systems. These requirements led us to Apache OpenWhisk [20], the basis of IBM Cloud Functions. It is the only complete FaaS platform open-sourced by a major provider and is straightforward to deploy. Finally, as OpenWhisk has been used in many recent studies [21–23], using it can help the reader to better relate the findings of this work to broader system trade-offs.

OpenWhisk uses Docker containers [15] without virtual machines to deploy functions, to reduce overhead. Practitioners are evaluating the differences and trade-offs between containerization and virtualization [24]. New virtual machines have been developed to better match the low startup latencies of containers. Examples include the production-grade Amazon Firecracker microVM [25, 26] and LightVM [27]. On the other hand, new containers have been introduced to decrease the overhead of containerization [28, 29].

The components of OpenWhisk are shown in Figure 3. HTTP requests intended to invoke a function (called an action in OpenWhisk) enter via an NGINX [30] reverse proxy. They are then forwarded to the Controller, which handles API requests like function invocations. The Controller checks the CouchDB database for authentication before loading action information from CouchDB. The Controller's Load Balancer then chooses a healthy and available Invoker to initiate the action and passes it the information via the Kafka publish-subscribe messaging component. The Invoker then invokes the action in a Docker container, taken from its pool. This container either comes from the developer, or is a language-specific container that the developer's code is injected into. On completion of the function invocation, the results and statistics about the execution are logged to CouchDB, and for blocking API calls, the results are returned to the end user.

OpenWhisk supports running functions in languages including Python, Node.js, Scala, Java, Go, Ruby, Swift, and PHP. Developers may provide their own Docker file to run function(s) of their choice.

3 METHODOLOGY

We use a server with an 8-core, 16-thread Intel Xeon E5-2620 v4 CPU which has 20MB of L3 cache. The server has 16GB of 2133MHz DDR4 RAM connected to a single channel, which we show has been sufficient for our experiments both regarding capacity and available bandwidth.

Listing 1: 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"
    }
}
```

We use the public OpenWhisk source code, as released on GitHub[1] and built from Git commit `084e5d3`. Our only changes from the default settings were to increase the limits on the number of concurrent invocations and invocations per minute, and to increase the memory allocated to the invoker, as discussed in Section 6.2. Finally, the test server's OS is Ubuntu 16.04.04 LTS.

3.1 FaaSProfiler

In this work, we introduce FaaSProfiler[2], a tool for testing and profiling FaaS platforms. We built FaaSProfiler based on the real needs and limitations we faced early on conducting our studies:

- **Arbitrary mix of functions and invocation patterns.** FaaSProfiler enables the description of various invocation patterns, function mixes, and activity windows in a clean, user-friendly format.
- **FaaS-testing not plug-and-play.** Each function should be invoked independently at the right time. Precisely invoking hundreds or thousands of functions per second needs a reliable, automated tool. We achieve this with FaaSProfiler.
- **Large amount of performance and profiling data.** FaaSProfiler enables fast analysis of performance profiling data (e.g., latency, execution time, wait time, etc.) together with resource profiling data (e.g L1-D MPKI, LLC misses, block I/O, etc.). The user can specify which parameters to profile and make use of the rich feature sets of open-source data analysis libraries like Python pandas [31].

Figure 3 shows FaaSProfiler and how it interacts with OpenWhisk via HTTP get/post. The user specifies the mix of function invocations in JSON format. Listing 1 shows an example JSON configuration. Here, the test is set to run for 25 seconds with a fixed random seed (110). Function invocations are non-blocking, where `function0` is invoked at 20 invocations per second (ips) between seconds 5 to 20 of the test with a Poisson distribution. This function is also invoked as `instance1` with 100ips between seconds 10 to 15 with a uniform distribution. Between seconds 1 to 24, `function1` is invoked at 15ips with a Poisson distribution. Note the image data file for `function1`. The binary data from the file is sent alongside the invocations. The example configuration also specifies an optional runtime profiling script (`MonitoringScript.sh`) to run concurrently at test start. This approach provides high flexibility and modularity for profiling. We utilize these runtime profiling scripts to use tools such as `perf` (Linux performance counters profiling), `pqos-msr` (Intel RDT Utility [32]), and `blktrace` (Linux block I/O tracing).

After specifying the config file, the user runs their experiments using the Synthetic Workload Invoker. Once complete, the Workload Analyzer can plot the results and archive them in standard Pandas data frames for later analysis and comparison of multiple tests using the Comparative Analyzer.

FaaSProfiler will provide researchers the ability to quickly and precisely profile FaaS platforms on real servers. It not only provides the entire testing and profiling environment, but also accelerates testing early-stage research ideas. After cloning, FaaSProfiler is set up with a single script. It is then ready to profile the user's functions. We also include a number of microbenchmarks and applications (discussed in Section 3.2).

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

Table 1: The list and description of FaaS applications we wrote or repurposed to use as benchmarks.

3.2 Benchmarks

Microbenchmarks. To investigate the overhead of FaaS function execution compared to native execution, we used a subset of the Python Performance Benchmark Suite [33]. We created OpenWhisk actions (functions) from the microbenchmarks, ignoring very similar microbenchmarks and those with a number of library dependencies, leaving us with a final set of 28. The same Python version was used throughout. For functions requiring libraries, we built unique minimal container images. The OpenWhisk action container includes code to receive and process the incoming HTTP requests to pass any arguments to the function that is being invoked, and to retrieve any results to send back to the caller.

Benchmark Applications. We have also written or repurposed five FaaS functions (detailed in Table 1) to use as representative FaaS benchmark applications. The functions are written in Python and NodeJS, and `ocr-img` calls the open-source Tesseract OCR [34] from NodeJS as an external binary. The containers are built either by injecting into the existing OpenWhisk containers for Python and NodeJS, or by creating a new container based on them when additional libraries or binaries are needed.

[1] https://github.com/apache/incubator-openwhisk
[2] http://parallel.princeton.edu/FaaSProfiler.html

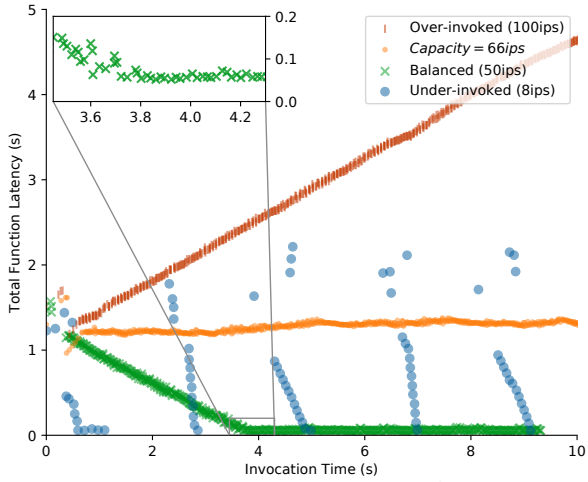

Figure 4: The invocation rate of a function (`json_dumps` here) determines its latency behavior. (Lower latency is better)

4 SYSTEM-LEVEL BEHAVIOR

Running functions natively is inherently different from provisioning functions in FaaS. For instance, the user directly starts a native function execution. However, function invocation in FaaS entails additional steps such as traversing load balancers and messaging layers. Even the very notion of performance is different. Runtime is a broadly accepted metric for functions running natively, but it is less clear what should contribute to a function performance metric for FaaS. This section aims to establish a common ground for the reader to better understand FaaS and to contextualize our later results.

4.1 Latency Modes and Server Capacity

Here, we show how the latency of function completions depend on invocation frequency in an interesting way. We demonstrate this with `json_dumps`, though we observe this behavior for all functions. Figure 4 shows the latency for four different *ips* rates, invoked in a uniform distribution for 10 seconds. Each rate has different latency variation behaviors which we call *latency modes*:

(1) **Over-invoked**: If no container is available for the invoker to provision to a function, the invocation is queued. When more functions are invoked than the server can execute (the red 100*ips* test in Figure 4), latency keeps increasing, due to the backlog of this internal queue. An active load balancer can avoid sending new invocations to over-invoked servers.

(2) **Under-invoked**: OpenWhisk pauses idle containers after a 50*ms* grace period to save memory. On the other hand, if invocations are queued and the server has available resources, new containers are started. Now, if a server is under-invoked (the blue 8*ips* test in Figure 4), containers become idle for long enough to be paused. This causes a latency ripple effect as invocations must wait periodically for container(s) for functions to be unpaused.

(3) **Balanced**: If the invocation rate is high enough to avoid pauses and low enough to stay within the server capacity, function invocation latency converges. For the green 50*ips* test in Figure 4, the latency converges to the execution time of roughly 55*ms* after about 3.75 seconds.

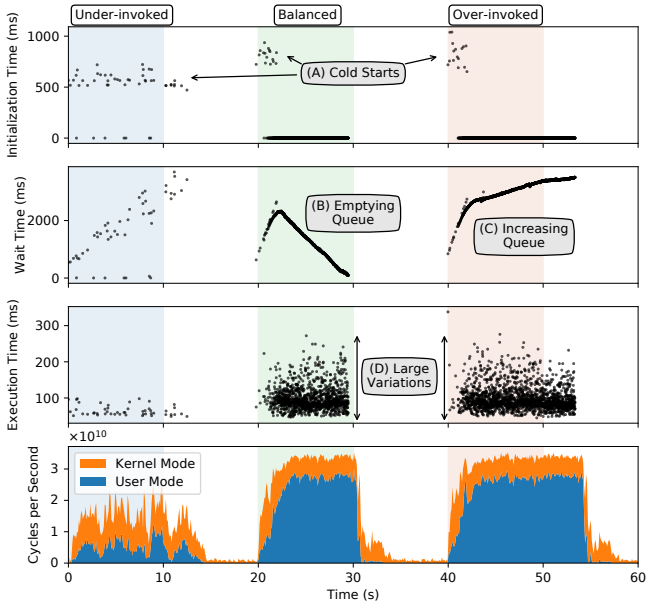

Figure 5: Function `json_dumps` invoked with different rates to study the latency modes. The breakdown of latency (made of initialization time, wait time, and execution time) is shown, as are the cycles per second spent in user and kernel modes. Kernel overhead is especially high for the under-invoked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

As shown in orange in Figure 4, the invocation rate can be set to match the server processing capacity, keeping the latency on the boundary of balanced and over-invoked modes. We use this number as a measure to capture the capacity of the server for a function; in this case for `json_dumps` it is 66ips.

4.2 Performance Breakdown

In Section 4.1, we showed how the latency of a function depends on the invocation rate. This latency comprises different components which we study here. We invoke `json_dumps` at 8ips (under-invoked), 50ips (balanced), and 100ips (over-invoked), each for a five-second phase. Between phases, there is a five-second idle period (no invocations). Figure 5 depicts the breakdown of latency components alongside CPU cycles spent in user and kernel modes.

Breakdown of function latency. The latency of a function invocation on a server (excluding network latency, and cluster-level scheduling and queueing) comprises of different components: (1) **Initialization time** or **cold start time** is spent preparing a function container. Warm functions skip initialization. (2) **Wait time** is spent waiting inside OpenWhisk before execution. (3) **Execution time** is the time taken for a container to run the function. All of these affect the end-to-end latency for the users. Therefore, delivering suitable FaaS offerings requires reducing cold starts or their cost, minimizing the queueing time for invocations, and keeping the function execution times low.

Cold starts are costly. We observe cold start time of at least 500ms and as high as 1000ms (marker A in Figure 5). This overhead occurs for the invocations that spin up new containers or un-pause paused containers. If containers remain live, there is no cold start.

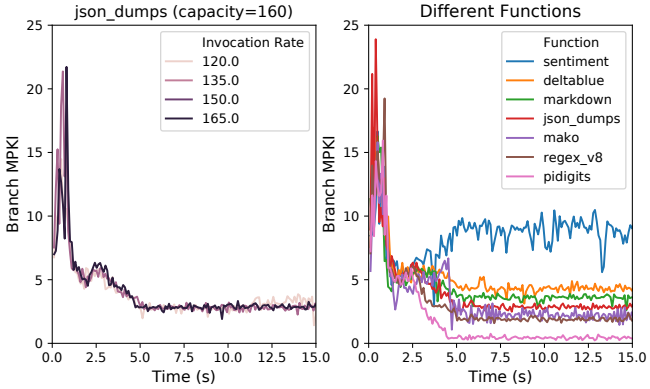

Figure 6: As long as function containers are alive, the invocation rate does not affect the branch MPKI (left). However, different functions consistently experienced different branch MPKI (right).

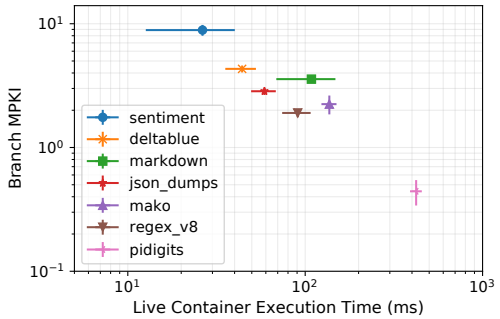

Figure 7: Functions with longer execution time have lower branch MPKI.

Wait time follows a queueing model. Function invocations are queued at the server-level to be run by worker container(s). While this local queue is different from cluster-level queues managed by load balancers, it still follows similar queuing principles that directly affect the QoS [35]. If the processing rate is higher than the invocation rate, the queue wait time decreases (balanced mode) as shown by marker B in Figure 5, and eventually flattens out (not pictured). However, if the invocation rate exceeds the processing capacity, there will be an accumulation of queueing delay and thus increasing wait time (over-invoked mode), as shown by marker C in Figure 5. When under-invoked, containers are paused/unpaused. Due to this varying processing capacity, the queue wait time varies.

Execution Time is variable. Marker D in Figure 5 shows high variability in execution time in the balanced and over-invoked modes when they reach the steady-state (once all containers have started). Such variations are mainly caused by involuntary context switches and get worsened by increased concurrency.

The OS kernel overhead is non-trivial. The kernel can consume a significant portion of active CPU cycles; especially when under-invoked, where containers are paused and unpaused regularly. The proportion of cycles spent in kernel mode, as compared to user mode is considerably higher when under-invoked. As the OS overhead is much less for native execution of functions, lowering the OS-related costs of virtualization for functions remains an active area of research [29, 36–38].

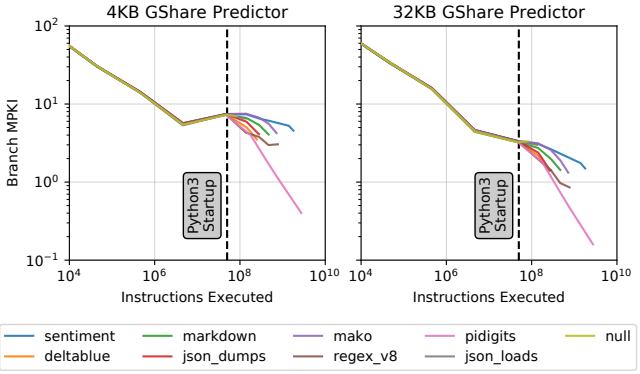

Figure 8: Simulated branch misprediction per kilo-instruction (MPKI) rates for 4KB and 32KB GShare predictors compared to execution time. Python startup overhead is significant for short functions, and MPKI is 16.18x (32KB) and 18.8x (4KB) higher for the shortest function compared to the longest.

5 COMPUTE AND MEMORY

To understand the architectural implications of FaaS, we examine which architectural components are affected the most by the fine-grained interleaving of many short functions. Here, we focus on the processor's primary locality-preserving mechanisms: branch predictors and caches. We find unique mismatches in the present-day design of branch prediction, LLC sizing, and memory bandwidth availability when they are used to run a FaaS environment. We then recommend how to improve the servers and broader cloud environment to better afford FaaS.

5.1 Branch Prediction

In this section, we explain how our series of experiments reveal insights into the branch prediction behavior of FaaS workloads. Specifically, we want to know to what extent branch predictors can handle short containerized function executions.

We started by invoking the `json_dumps` function with a high enough rate to ensure it remained in the balanced latency mode to keep all worker containers alive (i.e. at its capacity). This way, the container cold start code execution is eliminated from our profiling data. We observe that in the balanced mode, regardless of the invocation rate, the branch mis-predictions per kilo-instruction (MPKI) converges to the same value. This is shown in the left sub-figure of Figure 6.

After ensuring that the invocation rate within the balanced mode does not affect the branch MPKI values for a few functions, we compared the MPKI for different functions. To avoid the added complexity of a cross-language comparison, we chose from our python microbenchmarks and python applications. We observed that they have distinct branch MPKI convergence values, shown in the right sub-figure of Figure 6. At first look, one might think that these different MPKI values are due to different code for various functions. However, plotting the branch MPKI against the execution time, we came across the trend shown in Figure 7. Longer functions appear to have noticeably lower branch MPKI.

Observing this trend and the lack of more detailed visibility in our test machine motivated us to look deeper into understanding

this behavior. We wanted to understand the role of the language runtime's startup time for cold containers, and whether there is a clear loss in branch MPKI for shorter functions as compared to longer functions. We wrote a custom Pintool [39] to generate branch traces for some of our functions running outside of the containerized FaaS environment. Due to the limitations of Intel Pin, these traces do not include any branch information for execution inside the kernel. Additionally, these traces include the Python 3 language runtime startup. Using the simulation infrastructure provided as part of the Championship Branch Prediction 2016 (CBP-5), we simulated two GShare branch predictors [40] with Pattern History Tables of 4KB and 32KB in size.

The simulation results are shown in Figure 8. During Python 3 startup (indicated by the empty function, `null`, but expressed almost identically by all of our functions), MPKI is relatively high but decreases quickly. While the relationship of shorter functions having higher MPKI is intuitive, it is important to note that the traditional expectation is that programs will run for long enough to train the branch predictor (possibly across multiple OS scheduling quanta). In the FaaS environment, functions are short enough for MPKI to be noticeably affected by their length. This causes a difference in final MPKI between the shortest (`json_loads`) and longest (`pidigits`) functions of 18.8x for 32KB GShare and of 16.18x for 4KB GShare. Looking just at the data points for `pidigits`, we also see that it has a similarly high MPKI when it has executed the same number of instructions as `json_loads`, so this is not simply a matter of the shorter functions being harder to predict. Additionally, when a short function is infrequently invoked (so the predictor will not stay trained), the language runtime startup will make up a significant amount of the execution time. For a longer function like `pidigits` which takes over 300ms to execute, only 3% of the instructions are in the startup code. However, for the shortest function, `json_loads`, 60.9% of the instructions executed are from the startup code.

Implications. Our observation that present branch predictors experience higher MPKI for short functions motivates an investigation of branch predictor modifications to better target short-lived functions like FaaS. For instance, branch predictors that train quickly, or that retain state on a per-application basis. It also makes clear that FaaS function lifetime is short enough to harm temporal locality-related microarchitectural features.

5.2 Last-level Cache

To explore the impact of Last-level Cache (LLC) size on function cold start and execution time, we limited the amount of LLC accessible to CPU cores in hardware. This was feasible as our test server's CPU supports Intel Cache Allocation Technology (CAT) [41]. We controlled the LLC allocations using the Intel RDT Software Package [32]. We conducted experiments for each of our benchmark applications, during which we invoked each function with a low enough rate to experience cold start for every invocation.

As seen in Figure 9, LLC is not the performance bottleneck for these functions since reducing its size by almost 80% has no significant impact on their cold start or execution time. However, all of these functions experience significant slowdown with less than 2MB of LLC. As a reminder, a non-trivial amount of kernel code

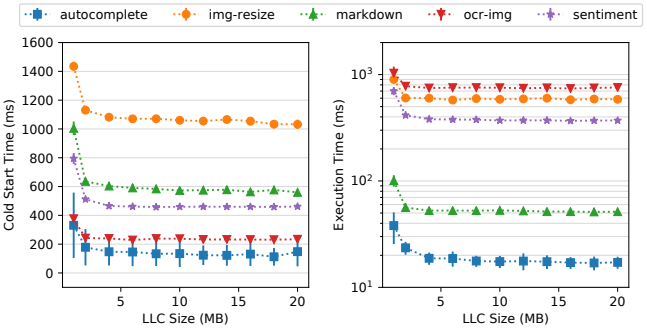

Figure 9: When invoked to cause cold starts, increasing LLC size beyond around 2MB does not significantly change the cold start latency and execution time of five functions.

executes on the server to enable OpenWhisk (Figure 5), thereby congesting the cache. As a side note, we observed that all Python 3 functions have the same cold start which is 3-5x higher than the NodeJS cold start.

Our observation regarding the low LLC requirement of FaaS workloads resonates well with the trend reported by prior studies on emerging cloud workloads [10, 42–44]. CloudSuite authors have shown that scale-out and server workloads have minimal performance sensitivity to LLC sizes beyond 4-6MB [43]. Analyzing modern latency-critical cloud workloads, Chen et al. [42] characterized them as "*not highly sensitive to LLC allocations especially at low load.*" Additionally, in the context of microservices, Gan et al. [10] have reported considerably lower LLC misses compared to traditional cloud applications. Considering that LLC takes a significant area on modern processors [44–46], our observation confirms the need to overcome this source of inefficiency in yet another class of cloud services.

Implications. Using current servers and in the short-term, cloud providers can utilize this ample LLC by partitioning the LLC generously to different system components, in an effort to mitigate LLC interference effects. However, in the long-term, providers might benefit from deploying processors with smaller LLCs or providing more processor cores, as has recently become popular for processors specifically designed for the cloud [47–49]. This would directly translate into lower TCO for providers, and thus cheaper services for clients.

5.3 Memory Bandwidth

We used Intel's Cache Monitoring Technology (CMT) and Memory Bandwidth Monitoring (MBM) [41] to monitor potential memory bandwidth contention. We invoked five different functions separately with their capacity rates on this server, under different cache sizes. The result is shown in Figure 10. Depending on the function and whether it uses the entire LLC, the memory bandwidth utilization can vary significantly – from 3.5% to 88% of the bandwidth limit of 17.07GB/s for our setup. Also, functions differed in their sensitivity to LLC size. But, under LLC scarcity, memory bandwidth was affected for all functions as the system started to thrash.

Values presented in Figure 10 correspond to invocation at capacity rates. We observed that although the total memory bandwidth usage at capacity is higher than balanced or under-invoked latency

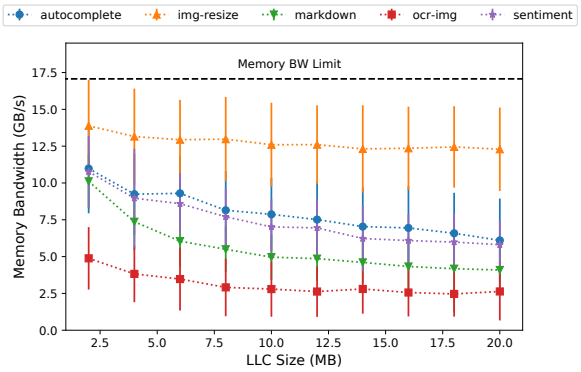

Figure 10: Memory bandwidth consumption varies a lot between functions. `img-resize` consumes the highest bandwidth (up to 88%) under default LLC size (20MB). Smaller LLC sizes translate into higher bandwidth usage. We run each function separately at its capacity.

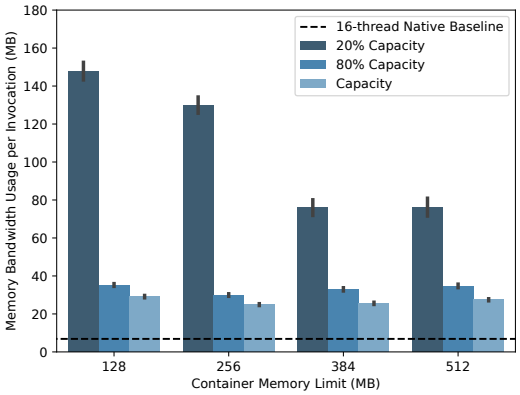

Figure 11: Under-invocation leads to cold starts, increasing the normalized memory bandwidth consumption per invocation. This behavior is independent of memory limit. We invoked different variants of `markdown` with their corresponding capacities to show this.

modes, per invocation bandwidth consumption follows the opposite trend. This is shown in Figure 11. We invoked four variants of `markdown` with 128MB, 256MB, 384MB, and 512MB of memory limit. These variants have capacities of 166ips, 160ips, 122ips, and 104ips, respectively. Each variant was invoked with a Poisson distribution at 20%, 80%, and 100% of its capacity. At 20% capacity, all variants experience many cold starts, and this significantly increases their normalized memory bandwidth usage.

Figure 11 also depicts the average per-invocation memory bandwidth usage for native execution of the `markdown` function. It is 3.8x lower than the lowest memory bandwidth usage of the corresponding OpenWhisk action. We measured this baseline by executing the function in sixteen concurrent processes, fully utilizing the sixteen hardware threads available on the server's CPU.

Implications. Knowing that memory bandwidth increases at cold-start time, memory bandwidth provisioning could be used to isolate newly starting containers. This would help to keep containers for other functions in their balanced mode. Additionally, prefetcher algorithms could be developed specifically for container cold-start

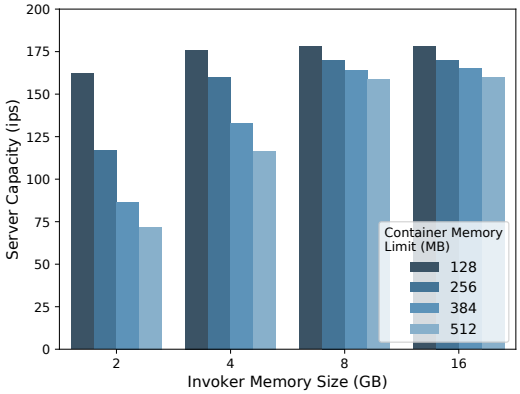

Figure 12: Invoker memory size determines the number of parallel containers, affecting the server capacity (`json_dumps` here). However, this parallelism has overheads which limit the capacity gains.

to ensure that the bandwidth is used most efficiently and that the function is able to move into its balanced mode as soon as possible.

6 PLACEMENT AND SCHEDULING

In this section, we take a step back, to see the implications of FaaS at a higher level. We explain how functions are scheduled inside a server and what it means for the ideal server size. We then present the consequences of this scheduling and how it can enable a FaaS provider to unfairly overcharge users mapped to an over-capacity server. Finally, we demonstrate intense interference effects in a busy FaaS server.

6.1 Invoker Scheduling

The Invoker is at the heart of OpenWhisk and is responsible for running containers. In OpenWhisk, the number of containers that can run at once (and their share of CPU time) is determined by the amount of memory allocated to the Invoker (known as its memory size) combined with each container's chosen memory limit.

To show this interplay, we determined the capacity of `json_dumps` under various container memory limits and Invoker memory sizes. The results are shown in Figure 12 – smaller function memory limits and larger Invoker memory sizes lead to higher capacity. We can see that the number of running containers is the primary factor affecting capacity, as doubling both the container memory limit and the Invoker memory size together results in the same capacity.

The interesting observation here is that for a fixed container memory limit, the gains of moving to larger invoker memory sizes quickly vanish. This is because other factors such as limitations on the available CPU time for a given function start to kick in. From another experiment we ran on the same function, we saw that doubling the Invoker memory size from 4GB to 8GB resulted in an average increase of 42.9% in latency and 70.5% in execution time. This is likely due to the overhead of managing and scheduling twice as many containers. We investigate this more in Section 6.2. Knowing this trade-off, we picked 4GB of Invoker memory to conduct our experiments in this work.

Implications. The behavior of the system as we varied Invoker memory size raises questions about OS scheduling for FaaS. Increasing the number of containers can improve server capacity, but

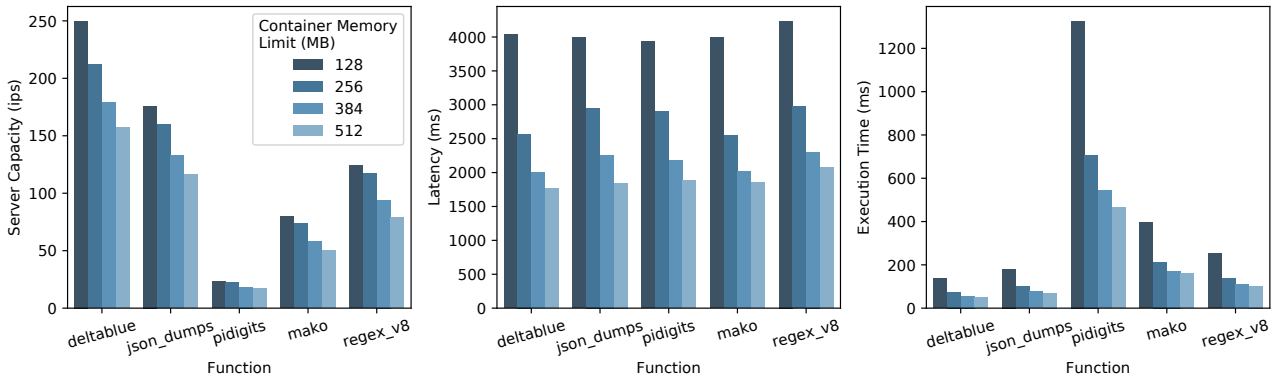

Figure 13: When invoked at capacity, reducing each function's memory limit from 512MB to 384MB and 256MB increases the latency and execution time roughly proportionally. However, decreasing the memory limit to 128MB increases capacity only slightly, while costing much more in latency and execution time.

is accompanied by significant increases in latency and execution time. Given that execution time for short functions is on the same order as the OS scheduling quantum, this motivates a FaaS-aware scheduler that would aim to complete the execution of a given function before invoking the scheduler again.

6.2 Balancing Demands

To ascertain the correct memory limit (and thus scheduling share) for several of our microbenchmark functions, we swept the space of function memory limits. Figure 13 shows the average measured capacity as well as latency and execution time at capacity for each sweep. While each metric increases when moving to smaller memory limits, the difference in magnitude of the increases raises a concerning imbalance between the demands of the FaaS provider (who values increasing capacity/throughput, but also profits from increasing execution time), the developer (who values decreasing execution time), and a potential individual end user (who values decreasing latency).

Looking deeper, we divided the increase in latency or execution time by the increase in capacity at the same step. This is the marginal increase in latency/execution time over the marginal increase in capacity for a step in memory limit. We see in Figure 13 that when stepping from 512MB to 384MB, and from 384MB to 256MB, the marginal capacity increase is close to proportional. However, when stepping to 128MB, the marginal capacity increase is only 69.4%-81.3% of the marginal latency increase and only 54.9%-64.5% of the marginal execution time increase. We chose a memory limit of 256MB as it remains within the reasonable regime of marginal increases of capacity compared to latency and execution time.

The imbalance comes as execution time (and thus price) and latency (the end user's experience) are lost in favor of the FaaS provider's preferred capacity maximization, while the provider guarantees the developer no QoS via the SLA. **The provider has the incentive to go over capacity, selling more function invocations per unit time on the same hardware, while also increasing functions' execution time, making the provider more money.** The execution time increase may not proportionally show in the latency, but it directly increases the cost of the service. Additionally, the extra wall-clock time charged to the developer may not actually be useful cycles. Our experiments show

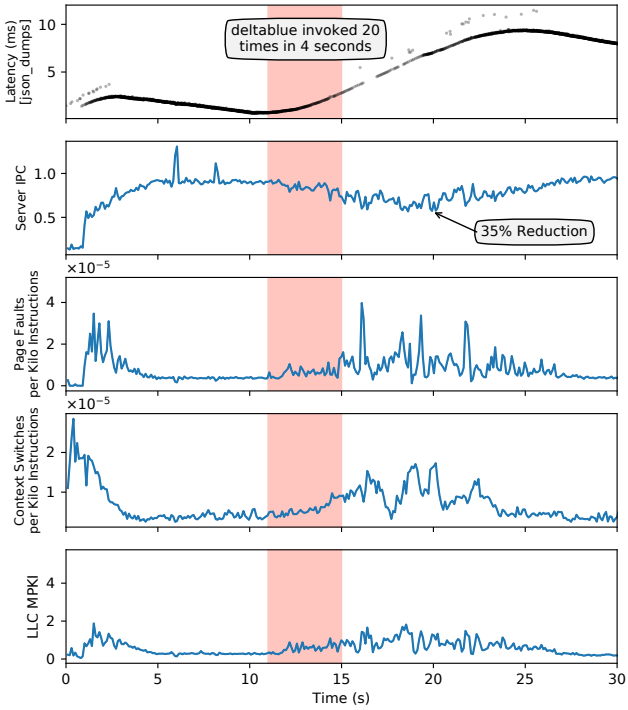

Figure 14: Interference effects are severe in FaaS. While `json_dumps` is invoked at 80% of its capacity (128ips), just 20 `deltablue` invocations in 4 seconds cause lingering effects that reduce IPC and increase page faults, context switches, and cache misses per kilo-instruction.

that those cycles are likely spent in OS scheduling, container management, page fault handling, or pipeline flushes due to branch mis-predictions.

Implications. Solving this pricing incentive phenomenon is beyond the scope of this paper. However, our findings motivate a search for solutions. Some possibilities are 1) pricing model changes, such as excluding system-level overheads or deploying incentive-compatible pricing schemes [50], 2) architectural changes, such as better isolation, and 3) SLA changes, structural guarantees made to the developer by the FaaS provider.

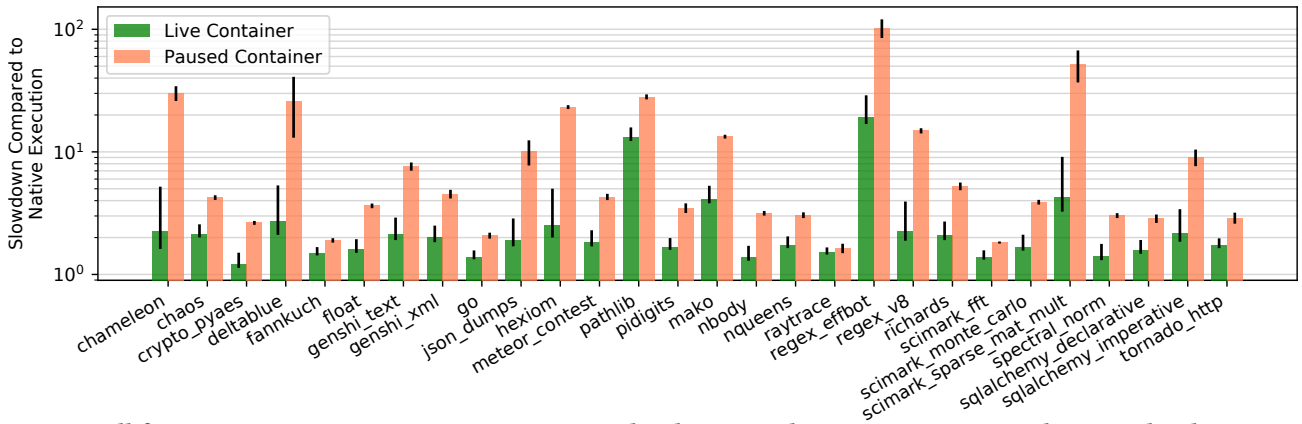

Figure 15: All functions experience a containerization overhead compared to native execution. These overheads are significantly higher for paused containers than for live containers. (Slowdown plotted as log, lower is better)

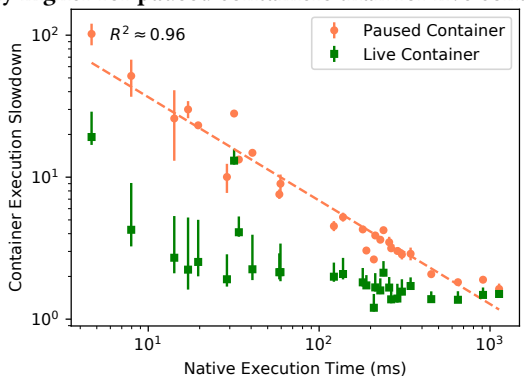

Figure 16: Shorter functions experience relatively higher slowdown. (Log-log plot, lower is better)

6.3 Interference

To characterize the interference effects within our system, we set up the function `json_dumps` to be invoked at 80% of its capacity (128 ips) for a 29 second period, putting it in what would normally be its balanced mode. At the 10th second, `deltablue` is invoked at a rate of 5 invocations per second for 4 seconds. Figure 14 shows this experiment, with `json_dumps` starting at the 1st second and `deltablue` starting at the 11th second. The figure shows latency for `json_dumps`, and the instructions per cycle (IPC), page faults per kilo-instruction, context switches per kilo-instruction, and LLC MPKI of the system. Looking at the latency, we can see that when `deltablue` begins to be invoked, `json_dumps` moves into its over-invoked mode, and takes some time to recover back to a balanced mode once `deltablue` is no longer being invoked. During this time window, we observe that IPC drops by 35% versus the balanced mode, and this correlates with significant increases in page faults, context switches, and LLC misses per kilo-instruction, which are all the overheads of bringing up new containers for `deltablue`.

Implications. A deeper architectural investigation of the interference phenomenon would shine light on the behavior we observe here. The increase in page faults per kilo-instruction canlikely be ameliorated by changes to the processor's TLBs. FaaS-specific LLC prefetching algorithms could also reduce LLC misses. Higher level isolation changes may also have positive impacts on all of the metrics we measured.

7 DISCUSSIONS

In this section, we discuss a few remaining aspects that should be taken into account when reasoning about the architectural implications of FaaS holistically. These include the overhead of containerization, broader differences of functions running natively versus in FaaS settings, and the impact of programming languages.

7.1 Overhead of Containers

While there have been numerous studies on the impact of containerization [51–53], we also characterize it in our setting to provide a complete view to the reader. Figure 15 shows the average slowdown experienced by different functions when running in Docker containers under OpenWhisk, compared to native execution (error bars indicate standard deviation). This slowdown is noticeable and depends on whether the action container is alive (green bars on left) or paused (orange bars on right). A paused container incurs considerably higher slowdown, as seen.

By default, OpenWhisk pauses idle containers after a 50*ms* grace period. While this number or the exact policy might vary from vendor to vendor, the issue is universal: containers cannot be kept alive forever and should be paused to release memory. In our experiments, we have observed that the additional latency due to paused containers is the dominant factor for performance variations. Fast container un-pausing, denser packing of live containers, and redefinition of the memory hierarchy to suit this setup are some of the potential architectural opportunities to tackle this issue.

We have observed a strong correlation between slowdown and function duration. In particular, there is a power-law relation between slowdown for paused action containers and the native execution time, as shown in Figure 16. **The shorter the function, the higher the slowdown**. This is due in part to fixed overheads being proportionally larger compared to the length of short functions. These slowdowns may necessitate consolidation of short functions by developers, or perhaps under the hood by the FaaS provider. Indeed, Akkus et al. [54] recently showed the advantages of application-level sandboxing for improving latency.

7.2 Native vs. in-FaaS Function Execution

Experiencing performance overheads due to containerization is not the only differences functions experience running on FaaS compared to native execution. We addressed many such differences

in the paper. As mentioned in Section 4.2, performance variability due to cold starts, orchestration, and queuing can contribute to QoS in FaaS. These overheads can become more problematic when function durations are shorter or when invocation patterns are burstier. Note that a fast burst can incur creating new containers and thus additional cold starts. Aside from the QoS aspects, containerized functions also consume more resources in FaaS settings. One example is the higher memory bandwidth usage, as shown in Section 5.3. In addition, provisioning additional isolation for security in FaaS further ramps up the resource usage [55]. Finally, the common practice of enforcing memory limits on FaaS functions makes them different from native functions. Memory limits in FaaS can be much lower than typical memory sizes of IaaS VMs. Therefore, a developer now needs to be more cautious about their function's maximum memory consumption.

7.3 The Role of Programming Language

We primarily used Python (and some NodeJS) functions to demonstrate system-level behavior. This is due to their wide support across every major platform. While we do not draw conclusions about the performance of different languages, we believe that our primary conclusions regarding overheads, latency modes, prediction performance, etc will hold, regardless of language. Other work has found variance in the cost of FaaS functions across different languages and FaaS platforms [4]. Going forward, providers will feel competitive pressure to improve the performance of the most heavily-used languages in order to lure customers to their platforms.

8 RELATED WORK

Studies demystifying serverless and FaaS. The majority of the related work on characterization of serverless/FaaS have been conducted from outside FaaS platforms [2–5, 56]. This means through careful testing, those researchers reverse-engineered what could be happening inside the black-box of commercial serverless systems. Despite providing valuable insights about FaaS offerings of providers such as AWS Lambda, Google Cloud Functions, Microsoft Azure Functions, and IBM Cloud Functions, those studies are bound by many practical restrictions. Lack of control over co-location [3], low visibility due to virtualization, and having no access to hardware performance counters are just a few such limitations. We are the first to take a bottom-up approach and conduct a server-level analysis of the FaaS model. This is vital, as we wanted to not only decouple network and server overheads, but also carefully understand overheads within a server. This enabled us to, for the first time, provide data about the architectural implications of FaaS.

Studies on architectural implications of emerging cloud models. Prior work has characterized emerging cloud models and workloads to understand inefficiencies in state-of-the-art server architectures. CloudSuite was introduced and has been used to study the architectural implications of scale-out workloads [43, 57]. It has influenced the design of new processor [58, 59] and accelerator [60] architectures for the cloud. A number of related works tackle characterizing resource consumption and performance breakdowns inside modern cloud data centers [61–65]. These studies are conducted at larger scale and have made valuable data publicly available to the research community. As microservices have gained interest recently, there have been studies discovering architectural implications [8] and proposing benchmark suites [10, 66] to enable further research. This paper is the first study to describe the architectural implications of FaaS applications and platforms. We also open source the testing and profiling tool we developed. FaaSProfiler will accelerate testing new ideas for other researchers and provide ground truth for architectural simulations by enabling the fast testing of a commercial-grade FaaS platform on local servers.

Profiling tools for the cloud. Researchers have developed numerous tools to profile and analyze cloud systems and workloads. One can classify those works based on their goal. There have been many works regarding tracing and diagnostics of cloud systems [67–71]. Another group of tools have been developed to conduct stress testing of cloud infrastructure [72, 73]. Profiling tools can be much more fine-grained in scope in order to conduct instruction-level introspection [39, 74–76]. Our tool, FaaSProfiler, is unique in that it is the first open testing and profiling tool to target a FaaS platform. Moreover, it flexibly enables the use of a wide range of available system profiling/tracing tools.

9 CONCLUSION

Function-as-a-Service (FaaS) serverless computing is an emerging service model in the cloud. The main body of FaaS research centers around cluster-level system management or building new applications using FaaS. However, serverless (while a nice name) is not server-less and functions still run on providers' servers. Understanding the implications of this new service model at the server-level is necessary for building next-generation cloud servers.

We provide the first server-level characterization of a Function-as-a-Service deployment on a commercial-grade platform. We find many new system-level and architectural insights on how short-lived deeply-virtualized functions can act against the common wisdom. In particular, architectural features that exploit temoporal locality and reuse are thwarted by the short function runtimes in FaaS. We see significant opportunity for computer architecture researchers to develop new mechanisms to address these challenges. Alongside this study, we open source a tool we developed, FaaSProfiler, which can accelerate testing and profiling FaaS platforms.

ACKNOWLEDGEMENTS

We thank Rodric Rabbah, Maria Gorlatova, Katie Lim, and Berkin Ilbeyi for constructive early-stage discussions. We also thank Alexey Lavrov, Amit Levy, Akshitha Sriraman, and anonymous reviewers for their valuable feedback on this work. This material is based on research sponsored by the NSF under Grants No. CCF-1822949 and CCF-1453112, Air Force Research Laboratory (AFRL) and Defense Advanced Research Projects Agency (DARPA) under agreement No. FA8650-18-2-7846, FA8650-18-2-7852, and FA8650-18-2-7862. The U.S. Government is authorized to reproduce and distribute reprints for Governmental purposes notwithstanding any copyright notation thereon. The views and conclusions contained herein are those of the authors and should not be interpreted as necessarily representing the official policies or endorsements, either expressed or implied, of Air Force Research Laboratory (AFRL) and Defense Advanced Research Projects Agency (DARPA), the NSF, or the U.S. Government. This work was partially supported by the AWS Cloud Credits for Research program.

REFERENCES

[1] "AWS Lambda." https://aws.amazon.com/lambda/. Accessed: 2019-8-27.

[2] G. McGrath and P. R. Brenner, "Serverless computing: Design, implementation, and performance," in *2017 IEEE 37th International Conference on Distributed Computing Systems Workshops (ICDCSW)*, pp. 405–410, June 2017.

[3] L. Wang, M. Li, Y. Zhang, T. Ristenpart, and M. Swift, "Peeking behind the curtains of serverless platforms," in *2018 USENIX Annual Technical Conference (ATC 18)*, (Boston, MA), pp. 133–146, USENIX Association, 2018.

[4] H. Lee, K. Satyam, and G. Fox, "Evaluation of production serverless computing environments," in *2018 IEEE 11th International Conference on Cloud Computing (CLOUD)*, vol. 00, pp. 442–450, Jul 2018.

[5] K. Figiela, A. Gajek, A. Zima, B. Obrok, and M. Malawski, "Performance evaluation of heterogeneous cloud functions," *Concurrency and Computation: Practice and Experience*, vol. 30, no. 23, p. e4792.

[6] IBM, "IBM Cloud Functions." https://www.ibm.com/cloud/functions. Accessed: 2019-08-30.

[7] E. van Eyk, L. Toader, S. Talluri, L. Versluis, A. Uta, and A. Iosup, "Serverless is more: From PaaS to present cloud computing," *IEEE Internet Computing*, vol. 22, pp. 8–17, Sep 2018.

[8] Y. Gan and C. Delimitrou, "The architectural implications of cloud microservices," *IEEE Computer Architecture Letters*, vol. 17, no. 2, pp. 155–158, 2018.

[9] A. Sriraman and T. F. Wenisch, "uTune: Auto-tuned threading for OLDI microservices," in *13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 18)*, (Carlsbad, CA), pp. 177–194, USENIX Association, 2018.

[10] Y. Gan, Y. Zhang, D. Cheng, A. Shetty, P. Rathi, N. Katarki, A. Bruno, J. Hu, B. Ritchken, B. Jackson, K. Hu, M. Pancholi, B. Clancy, C. Colen, F. Wen, C. Leung, S. Wang, L. Zaruvinsky, M. Espinosa, Y. He, and C. Delimitrou, "An open-source benchmark suite for microservices and their hardware-software implications for cloud and edge systems," in *Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '19, (New York, NY, USA), ACM, 2019.

[11] E. Cortez, A. Bonde, A. Muzio, M. Russinovich, M. Fontoura, and R. Bianchini, "Resource central: Understanding and predicting workloads for improved resource management in large cloud platforms," in *Proceedings of the 26th Symposium on Operating Systems Principles*, SOSP '17, (New York, NY, USA), pp. 153–167, ACM, 2017.

[12] M. Shahrad, C. Klein, L. Zheng, M. Chiang, E. Elmroth, and D. Wentzlaff, "Incentivizing self-capping to increase cloud utilization," in *Proceedings of the 2017 Symposium on Cloud Computing*, SoCC '17, (New York, NY, USA), pp. 52–65, ACM, 2017.

[13] Q. Liu and Z. Yu, "The elasticity and plasticity in semi-containerized co-locating cloud workload: A view from Alibaba trace," in *Proceedings of the ACM Symposium on Cloud Computing*, SoCC '18, (New York, NY, USA), pp. 347–360, ACM, 2018.

[14] CNCF Serverless Working Group, "Serverless whitepaper v1.0," tech. rep., Cloud Native Computing Foundation, March 2018.

[15] D. Merkel, "Docker: Lightweight linux containers for consistent development and deployment," *Linux J.*, vol. 2014, Mar. 2014.

[16] "AWS lambda announces service level agreement," tech. rep., Amazon Web Services, October 2018.

[17] S. Fouladi, R. S. Wahby, B. Shacklett, K. Balasubramaniam, W. Zeng, R. Bhalerao, A. Sivaraman, G. Porter, and K. Winstein, "Encoding, fast and slow: Low-latency video processing using thousands of tiny threads.," in *NSDI*, pp. 363–376, 2017.

[18] L. Ao, L. Izhikevich, G. M. Voelker, and G. Porter, "Sprocket: A serverless video processing framework," in *Proceedings of the ACM Symposium on Cloud Computing*, SoCC '18, (New York, NY, USA), pp. 263–274, ACM, 2018.

[19] L. Feng, P. Kudva, D. Da Silva, and J. Hu, "Exploring serverless computing for neural network training," in *2018 IEEE 11th International Conference on Cloud Computing (CLOUD)*, pp. 334–341, July 2018.

[20] Apache Software Foundation, "Apache OpenWhisk." https://openwhisk.apache.org. Accessed: 2019-08-30.

[21] I. Baldini, P. Cheng, S. J. Fink, N. Mitchell, V. Muthusamy, R. Rabbah, P. Suter, and O. Tardieu, "The serverless trilemma: Function composition for serverless computing," in *Proceedings of the 2017 ACM SIGPLAN International Symposium on New Ideas, New Paradigms, and Reflections on Programming and Software*, Onward! 2017, (New York, NY, USA), pp. 89–103, ACM, 2017.

[22] A. Mohan, H. Sane, K. Doshi, S. Edupuganti, N. Nayak, and V. Sukhomlinov, "Agile cold starts for scalable serverless," in *11th USENIX Workshop on Hot Topics in Cloud Computing (HotCloud 19)*, (Renton, WA), USENIX Association, July 2019.

[23] B. Trach, O. Oleksenko, F. Gregor, P. Bhatotia, and C. Fetzer, "Clemmys: Towards secure remote execution in FaaS," in *Proceedings of the 12th ACM International Conference on Systems and Storage*, SYSTOR '19, (New York, NY, USA), pp. 44–54, ACM, 2019.

[24] B. Ruan, H. Huang, S. Wu, and H. Jin, "A performance study of containers in cloud environment," in *Asia-Pacific Services Computing Conference*, pp. 343–356, Springer, 2016.

[25] J. Barr, "Firecracker - lightweight virtualization for serverless computing." https://aws.amazon.com/blogs/aws/firecracker-lightweight-virtualization-for-serverless-computing, November 2018.

[26] "Secure and fast microVMs for serverless computing." https://firecracker-microvm.github.io/. Accessed: 2019-8-27.

[27] F. Manco, C. Lupu, F. Schmidt, J. Mendes, S. Kuenzer, S. Sati, K. Yasukata, C. Raiciu, and F. Huici, "My VM is lighter (and safer) than your container," in *Proceedings of the 26th Symposium on Operating Systems Principles*, SOSP '17, (New York, NY, USA), pp. 218–233, ACM, 2017.

[28] J. Thalheim, P. Bhatotia, P. Fonseca, and B. Kasikci, "CNTR: Lightweight OS containers," in *2018 USENIX Annual Technical Conference (ATC 18)*, pp. 199–212, 2018.

[29] Z. Shen, Z. Sun, G.-E. Sela, E. Bagdasaryan, C. Delimitrou, R. Van Renesse, and H. Weatherspoon, "X-Containers: Breaking down barriers to improve performance and isolation of cloud-native containers," in *Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '19, (New York, NY, USA), pp. 121–135, ACM, 2019.

[30] W. Reese, "Nginx: the high-performance web server and reverse proxy," *Linux Journal*, vol. 2008, no. 173, p. 2, 2008.

[31] "pandas python data analysis library." https://pandas.pydata.org. Accessed: 2019-08-30.

[32] "Intel® RDT Software Package." https://github.com/intel/intel-cmt-cat. Accessed: 2019-8-27.

[33] V. Stinner, "The Python Performance Benchmark Suite, Version 0.7.0." https://pyperformance.readthedocs.io. Accessed: 2019-8-27.

[34] "Tesseract Open Source OCR Engine." https://github.com/tesseract-ocr/tesseract. Accessed: 2019-8-27.

[35] A. Mirhosseini and T. F. Wenisch, "The queuing-first approach for tail management of interactive services," *IEEE Micro*, vol. 39, pp. 55–64, July 2019.

[36] R. Koller and D. Williams, "Will serverless end the dominance of Linux in the cloud?," in *Proceedings of the 16th Workshop on Hot Topics in Operating Systems*, HotOS '17, (New York, NY, USA), pp. 169–173, ACM, 2017.

[37] H. Fingler, A. Akshintala, and C. J. Rossbach, "USETL: Unikernels for serverless extract transform and load why should you settle for less?," in *Proceedings of the 10th ACM SIGOPS Asia-Pacific Workshop on Systems*, APSys '19, (New York, NY, USA), pp. 23–30, ACM, 2019.

[38] E. G. Young, P. Zhu, T. Caraza-Harter, A. C. Arpaci-Dusseau, and R. H. Arpaci-Dusseau, "The true cost of containing: A gVisor case study," in *11th USENIX Workshop on Hot Topics in Cloud Computing (HotCloud 19)*, (Renton, WA), USENIX Association, July 2019.

[39] C.-K. Luk, R. Cohn, R. Muth, H. Patil, A. Klauser, G. Lowney, S. Wallace, V. J. Reddi, and K. Hazelwood, "Pin: Building customized program analysis tools with dynamic instrumentation," in *Proceedings of the 2005 ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '05, (New York, NY, USA), pp. 190–200, ACM, 2005.

[40] S. McFarling, "Combining branch predictors," Tech. Rep. TN-36, Digital Western Research Laboratory, June 1993.

[41] A. Herdrich, E. Verplanke, P. Autee, R. Illikkal, C. Gianos, R. Singhal, and R. Iyer, "Cache QoS: From concept to reality in the Intel® Xeon® processor E5-2600 v3 product family," in *High Performance Computer Architecture (HPCA), 2016 IEEE International Symposium on*, pp. 657–668, IEEE, 2016.

[42] S. Chen, C. Delimitrou, and J. F. Martínez, "PARTIES: QoS-aware resource partitioning for multiple interactive services," in *Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '19, (New York, NY, USA), ACM, 2019.

[43] M. Ferdman, A. Adileh, O. Kocberber, S. Volos, M. Alisafaee, D. Jevdjic, C. Kaynak, A. D. Popescu, A. Ailamaki, and B. Falsafi, "Clearing the clouds: A study of emerging scale-out workloads on modern hardware," in *Proceedings of the Seventeenth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS XVII, (New York, NY, USA), pp. 37–48, ACM, 2012.

[44] A. Shahab, M. Zhu, A. Margaritov, and B. Grot, "Farewell my shared LLC! a case for private die-stacked DRAM caches for servers," in *2018 51st Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pp. 559–572, Oct 2018.

[45] A. Jaleel, J. Nuzman, A. Moga, S. C. Steely, and J. Emer, "High performing cache hierarchies for server workloads: Relaxing inclusion to capture the latency benefits of exclusive caches," in *High Performance Computer Architecture (HPCA), 2015 IEEE 21st International Symposium on*, pp. 343–353, IEEE, 2015.

[46] J. Zhan, O. Kayiran, G. H. Loh, C. R. Das, and Y. Xie, "OSCAR: Orchestrating STT-RAM cache traffic for heterogeneous CPU-GPU architectures," in *2016 49th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pp. 1–13, Oct 2016.

[47] M. McKeown, Y. Fu, T. Nguyen, Y. Zhou, J. Balkind, A. Lavrov, M. Shahrad, S. Payne, and D. Wentzlaff, "Piton: A manycore processor for multitenant clouds," *IEEE Micro*, vol. 37, pp. 70–80, Mar 2017.

[48] M. McKeown, A. Lavrov, M. Shahrad, P. J. Jackson, Y. Fu, J. Balkind, T. M. Nguyen, K. Lim, Y. Zhou, and D. Wentzlaff, "Power and energy characterization of an open source 25-core manycore processor," in *2018 IEEE International Symposium on High Performance Computer Architecture (HPCA)*, pp. 762–775, Feb 2018.

[49] J. Balkind, M. McKeown, Y. Fu, T. Nguyen, Y. Zhou, A. Lavrov, M. Shahrad, A. Fuchs, S. Payne, X. Liang, M. Matl, and D. Wentzlaff, "OpenPiton: An open source manycore research framework," in *Proceedings of the Twenty-First International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '16, (New York, NY, USA), pp. 217–232, ACM, 2016.

[50] M. Shahrad and D. Wentzlaff, "Availability Knob: Flexible user-defined availability in the cloud," in *Proceedings of the Seventh ACM Symposium on Cloud Computing*, SoCC '16, (New York, NY, USA), pp. 42–56, ACM, 2016.

[51] W. Felter, A. Ferreira, R. Rajamony, and J. Rubio, "An updated performance comparison of virtual machines and Linux containers," in *2015 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*, pp. 171–172, March 2015.

[52] E. Casalicchio and V. Perciballi, "Measuring docker performance: What a mess!!!," in *Proceedings of the 8th ACM/SPEC on International Conference on Performance Engineering Companion*, ICPE '17 Companion, (New York, NY, USA), pp. 11–16, ACM, 2017.

[53] Z. Li, M. Kihl, Q. Lu, and J. A. Andersson, "Performance overhead comparison between hypervisor and container based virtualization," in *2017 IEEE 31st International Conference on Advanced Information Networking and Applications (AINA)*, pp. 955–962, March 2017.

[54] I. E. Akkus, R. Chen, I. Rimac, M. Stein, K. Satzke, A. Beck, P. Aditya, and V. Hilt, "SAND: towards high-performance serverless computing," in *Proceedings of the USENIX Annual Technical Conference (ATC 18)*, 2018.

[55] S. Brenner and R. Kapitza, "Trust more, serverless," in *Proceedings of the 12th ACM International Conference on Systems and Storage*, pp. 33–43, ACM, 2019.

[56] M. Gorlatova, H. Inaltekin, and M. Chiang, "Characterizing task completion latencies in fog computing," *arXiv preprint arXiv:1811.02638*, 2018.

[57] M. Ferdman, A. Adileh, O. Kocberber, S. Volos, M. Alisafaee, D. Jevdic, C. Kaynak, A. D. Popescu, A. Ailamaki, and B. Falsafi, "Quantifying the mismatch between emerging scale-out applications and modern processors," *ACM Transactions on Computer Systems*, vol. 30, no. 4, p. 24, 2012.

[58] P. Lotfi-Kamran, B. Grot, M. Ferdman, S. Volos, O. Kocberber, J. Picorel, A. Adileh, D. Jevdjic, S. Idgunji, E. Ozer, and B. Falsafi, "Scale-out processors," *Proceedings of the 39th Annual International Symposium on Computer Architecture*, p. 12, 2012.

[59] J. Kim, E. Teran, P. V. Gratz, D. A. Jiménez, S. H. Pugsley, and C. Wilkerson, "Kill the program counter: Reconstructing program behavior in the processor cache hierarchy," in *Proceedings of the Twenty-Second International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '17, (New York, NY, USA), pp. 737–749, ACM, 2017.

[60] J. Ahn, S. Hong, S. Yoo, O. Mutlu, and K. Choi, "A scalable processing-in-memory accelerator for parallel graph processing," in *Proceedings of the 42nd Annual International Symposium on Computer Architecture*, ISCA '15, (New York, NY, USA), pp. 105–117, ACM, 2015.

[61] K. Lim, P. Ranganathan, J. Chang, C. Patel, T. Mudge, and S. Reinhardt, "Understanding and designing new server architectures for emerging warehouse-computing environments," in *Proceedings of the 35th Annual International Symposium on Computer Architecture*, ISCA '08, pp. 315–326, 2008.

[62] Z. Jia, L. Wang, J. Zhan, L. Zhang, and C. Luo, "Characterizing data analysis workloads in data centers," in *2013 IEEE International Symposium on Workload Characterization (IISWC)*, pp. 66–76, Sept 2013.

[63] S. Kanev, J. P. Darago, K. Hazelwood, P. Ranganathan, T. Moseley, G.-Y. Wei, and D. Brooks, "Profiling a warehouse-scale computer," in *Proceedings of the 42Nd Annual International Symposium on Computer Architecture*, ISCA '15, pp. 158–169, 2015.

[64] L. A. Barroso, M. Marty, D. A. Patterson, and P. Ranganathan, "Attack of the killer microseconds.," *Commun. ACM*, vol. 60, no. 4, pp. 48–54, 2017.

[65] K. Hazelwood, S. Bird, D. Brooks, S. Chintala, U. Diril, D. Dzhulgakov, M. Fawzy, B. Jia, Y. Jia, A. Kalro, J. Law, K. Lee, J. Lu, P. Noordhuis, M. Smelyanskiy, L. Xiong, and X. Wang, "Applied machine learning at Facebook: A datacenter infrastructure perspective," in *2018 IEEE International Symposium on High Performance Computer Architecture (HPCA)*, pp. 620–629, Feb 2018.

[66] A. Sriraman and T. F. Wenisch, "uSuite: A benchmark suite for microservices," in *2018 IEEE International Symposium on Workload Characterization (IISWC)*, 2018.

[67] C. Reiss, A. Tumanov, G. R. Ganger, R. H. Katz, and M. A. Kozuch, "Heterogeneity and dynamicity of clouds at scale: Google trace analysis," in *Proceedings of the Third ACM Symposium on Cloud Computing*, SoCC '12, (New York, NY, USA), pp. 7:1–7:13, ACM, 2012.

[68] M. Chow, D. Meisner, J. Flinn, D. Peek, and T. F. Wenisch, "The mystery machine: End-to-end performance analysis of large-scale internet services," in *11th USENIX Symposium on Operating Systems Design and Implementation (OSDI 14)*, pp. 217–231, 2014.

[69] T. Xu, X. Jin, P. Huang, Y. Zhou, S. Lu, L. Jin, and S. Pasupathy, "Early detection of configuration errors to reduce failure damage," in *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI 16)*, pp. 619–634, 2016.

[70] H. Baek, A. Srivastava, and J. Van der Merwe, "CloudSight: A tenant-oriented transparency framework for cross-layer cloud troubleshooting," in *Proceedings of the 17th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing*, CCGrid '17, (Piscataway, NJ, USA), pp. 268–273, IEEE Press, 2017.

[71] Y. Gan, Y. Zhang, K. Hu, Y. He, M. Pancholi, D. Cheng, and C. Delimitrou, "Seer: Leveraging big data to navigate the complexity of performance debugging in cloud microservices," in *Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '19, (New York, NY, USA), ACM, 2019.

[72] F. Chen, J. Grundy, J. Schneider, Y. Yang, and Q. He, "StressCloud: A tool for analysing performance and energy consumption of cloud applications," in *2015 IEEE/ACM 37th IEEE International Conference on Software Engineering*, vol. 2, pp. 721–724, May 2015.

[73] A. Alourani, M. A. N. Bikas, and M. Grechanik, "Search-based stress testing the elastic resource provisioning for cloud-based applications," in *International Symposium on Search Based Software Engineering*, pp. 149–165, Springer, 2018.

[74] N. Nethercote and J. Seward, "Valgrind: A framework for heavyweight dynamic binary instrumentation," in *Proceedings of the 28th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '07, (New York, NY, USA), pp. 89–100, ACM, 2007.

[75] X. Tong, J. Luo, and A. Moshovos, "QTrace: An interface for customizable full system instrumentation," in *2013 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*, pp. 132–133, IEEE, 2013.

[76] N. Chachmon, D. Richins, R. Cohn, M. Christensson, W. Cui, and V. J. Reddi, "Simulation and analysis engine for scale-out workloads," in *Proceedings of the 2016 International Conference on Supercomputing*, ICS '16, (New York, NY, USA), pp. 22:1–22:13, ACM, 2016.