Understanding, Predicting and Scheduling Serverless Workloads under Partial Interference

Laiping Zhao
laiping@tju.edu.cn
State Key Laboratory of Communication Content Cognition, Colleage of Intelligence & Computing(CIC), Tianjin University

Yanan Yang
ynyang@tju.edu.cn
CIC, Tianjin University, Tianjin Key Lab. of Advanced Networking(TANKLAB)

Yiming Li
l_ym@tju.edu.cn
CIC, Tianjin University, TANKLAB

Xian Zhou
logzx2019@tju.edu.cn
CIC, Tianjin University, TANKLAB

Keqiu Li
keqiu@tju.edu.cn
CIC, Tianjin University, TANKLAB

Abstract

Interference among distributed cloud applications can be classified into three types: *full*, *partial* and *zero*. While prior research merely focused on *full interference*, the *partial interference* that occurs at parts of applications is far more common yet still lacks in-depth study. Serverless computing that structures applications into small-sized, short-lived functions further exacerbate *partial interference*. We characterize the features of partial interference in serverless as exhibiting *high volatility*, *spatial-temporal variation*, and *propagation*. Given these observations, we propose an incremental learning predictor, named *Gsight*, which can achieve high precision by harnessing the spatial-temporal overlap codes and profiles of functions via an end-to-end call path. Experimental results show that *Gsight* can achieve an average error of 1.71%. Its convergence speed is at least 3× faster than that in a serverful system. A scheduling case study shows that the proposed method can improve function density by ≥18.79% while guaranteeing the quality of service (QoS).

CCS Concepts: • **Computer systems organization** → **Cloud computing**.

Keywords: Serverless, Partial Interference, Performance Prediction, Resource Utilization

ACM Reference Format:
Laiping Zhao, Yanan Yang, Yiming Li, Xian Zhou, and Keqiu Li. 2021. Understanding, Predicting and Scheduling Serverless Workloads under Partial Interference. In *The International Conference for High Performance Computing, Networking, Storage and Analysis (SC '21), November 14–19, 2021, St. Louis, MO, USA.* ACM, New York, NY, USA, 13 pages. https://doi.org/10.1145/3458817.3476215

1 Introduction

Serverless computing has grown rapidly in recent years due to its low-cost and management-free operation properties. Many applications are being deployed in commercial serverless platform [9, 10, 14, 22, 26, 39, 40, 48, 50]. We summarize and categorize these applications into three dimensions (as in Table 1): *scheduled-background* (BG), *short-term computing* (SC) and *latency-sensitive* (LS).

In any category, **the *functions* are usually *small in size* and relatively *short-lived.*** *Small-sized* functions increase the colocating density. For example, AWS Lambda configures each instance with only 0.125-3GB of memory and 0.5GB of local storage [1]. A server with 256GB memory can accommodate hundreds of such functions. *Short-lived* functions exacerbate the dynamicity of system runtime states. These functions can be triggered by associated events at any time and are released after completion. For example, BG or LS functions usually process a request within seconds or milliseconds. Even for SC applications that share similar workloads with traditional serverful best effort tasks (BEs) [33], their maximum processing times are also limited to 900 seconds with AWS Lambda [1].

High-density, highly dynamic serverless workloads make it more challenging to colocate workloads with QoS guarantees (e.g., latency). First, while it is possible to isolate traditional coarse-grained components from coscheduled workloads using resource partitioning tools (e.g., Intel Cache Allocation Technology (CAT), Memory Bandwidth Allocation (MBA)), the many small-sized functions in serverless must share limited cores, memory bandwidth and LLC to improve resource efficiency. This inevitably causes interference. Second, the high dynamicity of functions likely make resource allocation approaches that rely on reactive methods (e.g., [6, 33, 34, 41, 44, 65]) less efficient by causing them to

SC '21, November 14–19, 2021, St. Louis, MO, USA

ACM ISBN 978-1-4503-8442-1/21/11…$15.00
https://doi.org/10.1145/3458817.3476215

Table 1. Serverless workload survey. BG: Scheduled Background; SC: Short-term Computing; LS: Latency Sensitive

	Description	**ServerlessExamples**
BG	triggered or scheduled intermittently, and needs to be run from time to time without any latency requirements.	IoT data collection, monitoring
SC	minute-level processing times; millisecond changes in completion times are trivial.	Bigdata[22, 26, 40], linearAlge[50]
LS	frequent invocations; millisecond increases in latency lead to nontrivial user experience degradation.	Websearch[9], e-com[39], SN[10]

miss the optimal tuning opportunities. For example, when a reactive approach adds cache ways to the coscheduled functions due to an observed increase in cache misses during the previous time window, the actual LLC demand may instantly decrease due to the release of functions therein.

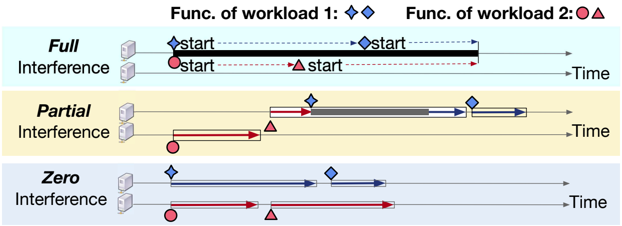


Figure 1. Various interference scenarios. *Full* indicates that the workloads share the same server; *Partial* indicates that the workloads intersect at partial servers; and *Zero* indicates that there is no interference between workloads.

Current resource management capabilities in open source serverless frameworks (e.g., OpenFaaS [42], OpenWhisk [43]) rely heavily on Kubernetes [27], which is a general framework that does not provide explicit optimization for colocation and QoS guarantees. The traditional *prediction-based* approaches (Table 2) that forecast mutual interference based on workload profiles can be adapted into a serverless system by treating each function as a separate application. However, the *diversity* across a set of user functions makes the profiling phase costly. Moreover, the accuracy of the predictions of such approaches can be compromised by *partial interference* (Figure 1): *interference occurs only at some, but not all, of the workloads.* Compared with serverful workloads, small-sized serverless functions are more prone to *partial interference*. For example, scheduling *n* functions of a single application following the *balancedResourceAllocation* priority (i.e., a default scheduling policy in Kubernetes [28]) would spread them across at most *n* servers. The interference experienced by any one of them may generate quite different impacts on the overall performance, leading prior monolithic predictors (e.g., [5, 11, 35]) that did not distinguish functions to failures.

In this work, we characterize the features of *partial interference*, which include the following: (1) **High volatility:** Partial interference may either have a negligible impact on the end-to-end QoS when it occurs on a noncritical path or manifest an effect similar to full interference. (2) **Spatial variation:** Two function call paths may intersect at arbitrary locations. The number of partial colocations increases exponentially with the number of servers and functions, resulting in different levels of performance degradation due to the *inconsistent sensitivities of functions* [65]. (3) **Temporal variation:** The different phase overlaps of two colocated functions can also generate different types of partial interference, as both the invocation frequency and run program segments are time-varying. (4) **Propagation:** Although functions are stateless in serverless, the impact on the overall QoS is not equivalent to a simple summation of the partial interference at each function, as hotspots propagate across dependent functions [17]. Hence, we need to explore a holistic prediction model that takes partial interference into account.

Although *partial interference* makes it difficult to predict the QoS of colocated workloads, we also observe the **predictability** feature: diverse yet small-sized functions can expose more information about the inherent workload structure, and rich details at the function level can help to improve prediction accuracy. Given this finding, we present *Gsight*, an accurate performance predictor for colocated serverless workloads. *Gsight* identifies the temporality and spatiality of partial interference and encodes both the "function-server mappings" and the "time overlap" information in a prediction model (§ 3.3). To reduce the profiling cost of diverse functions, our approach only requires the solo-run profiles of colocated functions and employs an incremental learning model that can converge quickly using the accurate function profiles. *Gsight* treats applications as black-boxes and uses only system-layer and microarchitecture-layer metrics to estimate performance degradation. Hence, it is application-agnostic and can be used in public clouds where the administrator cannot monitor the application-level metrics.

In summary, we provide the following contributions:

- We observe the effects of *partial interference* among colocated distributed workloads and characterize this interference using spatial-temporal overlap coding.
- We observe that function-level profiling enables rapid convergence and highly accurate prediction.
- We design a prototype, *Gsight*, that employs incremental learning to accurately predict the QoS for any colocation of BG, SC and LS workloads. We demonstrate its effectiveness via a scheduling use case.
- We present a detailed evaluation of *Gsight* and demonstrate its high accuracy regarding performance prediction and its significant resource utilization improvement.

2 Understanding Partial Interference

In this section, we characterize partial interference, and evaluate how serverless workloads aggravate such interference. **Methodology** We create multiple partial interference scenarios by colocating BG, SC and LS workloads arbitrarily to study their impacts on application performance. As BG functions usually have very lenient performance requirements, we mainly study the tail latency and end-to-end average

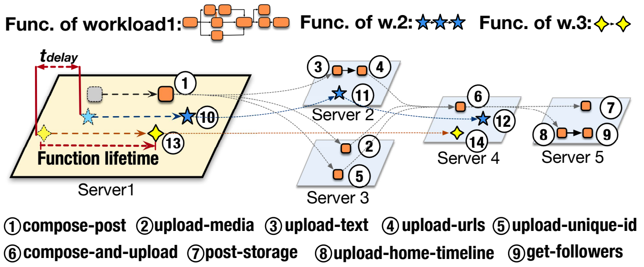
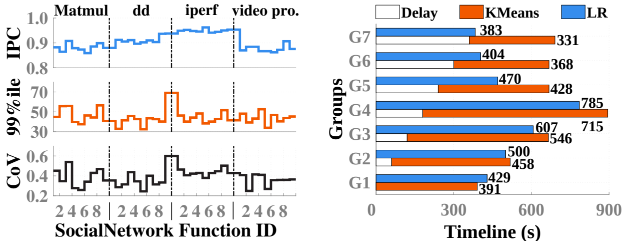

Func. of workload1:
Func. of w.2:
Func. of w.3:
t_{delay}
Function lifetime
Server1
Server 2
Server 3
Server 4
Server 5
①
②
③
④
⑤
⑥
⑦
⑧
⑨
⑩
⑪
⑫
⑬
⑭
①compose-post ②upload-media ③upload-text ④upload-urls ⑤upload-unique-id
⑥compose-and-upload ⑦post-storage ⑧upload-home-timeline ⑨get-followers

Figure 2. Examples of function invocations. Workload 1(*social network*) consists of a message-posting request and invokes 9 functions (①-⑨); Workload 2 consists of 3 functions (⑩-⑫); Workload 3 consists of 2 functions (⑬-⑭).

instructions per cycle (IPC) of LS workloads and the job completion times (JCTs) of SC workloads under partial interference. We choose applications in FunctionBench [25] as BG or SC functions, because their execution generally takes several minutes (except *float operation*). For LS workloads, we choose the *social network* in DeathStarBench [16] and adapt it into serverless functions as in [61]. Figure 2 shows an example function call path of *message posting* in the *social network*, which consists of nine functions over multiple branches. All benchmarks are implemented and deployed on OpenFaaS [42], an open source serverless framework.

2.1 Observations

Observation 1: *High volatility: Serverless functions are diverse in terms of execution behavior and resource consumption, making partial interference more volatile.*



(a) Partial interference scenarios (b) Inter. at ⑥compose-and-upload

Figure 3. (a): Both the 99*th* percentile latency (ms) and IPC of message-posting vary significantly under different partial interferences. (b) The JCTs of colocated LogisticRegression (LR) and Kmeans when they are started at different times.

Formulating an application into a collection of functions enables the independent scheduling of each function. Their independent deployment creates more partial interference opportunities, as every function can be colocated with another from an arbitrary application. Because of the diversity of functions in terms of execution behaviors and resource consumption, the partial interference can have quite different impacts on performance, either as strong as full interference or as weak as zero interference.

We select three microbenchmarks (*matrix multiplication*, *dd* and *iperf*) and one application (*video processing*) from FunctionBench [25] and colocate them with every function of the *social network* separately, generating 36 partial interference scenarios. In particular, *matrix multiplication*, *dd* and *iperf* are CPU-intensive, disk I/O-intensive and network-intensive applications, respectively. The *video processing* application generates high pressure on CPU and memory and medium pressure on disk I/O and network. Figure 3(a) shows how the 99*th* percentile latency, the coefficient of variance (CoV) of the latencies and the IPC vary over the partial interference scenarios. We see that the impact of partial interference exhibits higher volatility under different scenarios: colocating *matrix multiplication* or *video processing* with *social network* decreases the IPC significantly, while *iperf* does not impact IPC greatly due to its network-intensive nature. The difference in 99*th* percentile latency among these scenarios reaches 7×.

Observation 2: *Spatial variation: Serverless functions are small in size and stateless, making partial interference spatially varied.*

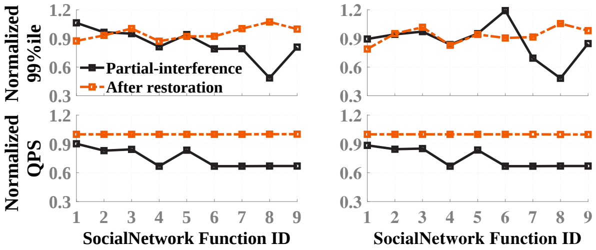
In terms of memory allocation, 90% of the Azure functions never request greater than 400MB, and 50% of the application runtime is allocated at most 170MB [49]. These small-sized functions cause high-density deployment at each physical server, inevitably aggravating partial interference. Moreover, functions are stateless and spread over multiple machines in the cluster, leading to various interference locations.

The end-to-end performance of a workflow varies significantly even under the contention from the same workload due to the inconsistent interference tolerance abilities of functions (Figure 3(a)). For example, the colocation of *matrix multiplication* with the ⑨*get-followers* function generates 99*th* percentile latency that is 3× higher than that obtained with ①*compose-post*. Furthermore, the end-to-end performance relies on the function call path structure. Interference on the critical path (e.g., ①→②→⑥→⑧→⑨) generates a much more severe impact than interference on the noncritical path (e.g., ③,④,⑤,⑦). Hence, to provide SLA guarantees under partial interference, the colocation scheme should be aware of spatially-varied interference.

Observation 3: *Temporal variation: Serverless functions are short-lived, making partial interference temporally varied.*

The characterization of Azure Function workloads shows that 50% of the invocations are executed for less than 1 second on average, and 96% of functions take less than 60 seconds on average [49]. Short-lived functions cause colocated functions to overlap in an arbitrarily short time period. Besides, although the current Azure has a low invocation frequency for many of their functions [49], we expect that the overall invocation frequency will continue to rise as the technology matures. These frequent, short invocations make partial interference more temporally varied.

To evaluate temporally-varied partial interference, we use two additional SC workloads: *Logistic Regression* processes 4 million example data with a size of 15GB, and *KMeans* clusters two partitions of 4 million points with a size of 15GB [30]. Both workloads employ 60 instances, and are bound in the same socket. Figure 3(b) shows the timelines of their executions with different start delays. In configurations $\{g_1, ..., g_7\}$, the start delay of *KMeans* gradually increases from 0 to 360 seconds at a step size of 60 seconds. We see that the JCT of *LogisticRegression* increases from 429 to 785 seconds from g_1 to g_4, implying that the later phase of the map and the shuffle phase are more sensitive to interference. In g_5-g_7, the JCT of *LogisticRegression* becomes much smaller as the time overlap becomes shorter. *KMeans* exhibits similar performance, and the maximum difference in JCT is more than 2×. Hence, to provide SLA guarantees under partial interference, the colocation scheme should be aware of such temporally-varied interference.



(a) Inter. at ①compose-post (b) Inter. at ⑥compose-and-upload

Figure 4. Performance changes achieved by all nine functions under partial interference or after local control. (a): Interference or control occurs at ①compose-post. (b) Interference or control occurs at ⑥compose-and-upload.

Observation 4: *Hotspot propagation: Partial interference triggers a chain reaction across the function call path and leads to diametrically opposite effects.*

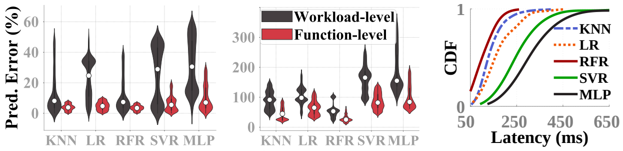
We are surprised to observe that partial interference affects the performance of functions in opposite ways: while the performance of colocated functions is degraded by interference, the subsequent functions' performance on the call path instead improves. For example, Figure 4(a) shows that all functions except ① have lower 99*th* percentile latencies after we create interference at function ①. Likewise, when function ⑥ is interfered (Figure 4(b)), the 99*th* percentile latencies of the other functions become lower than before.

There are two reasons for this fact: (1) Interference causes saturation in the colocated functions, resulting in increased local latency. In a *sequence chain*, the queries per second (QPS) of the subsequent invoked functions decreases due to the waiting on blocking requests at the previous function. Hence, their local latencies also decrease. In the case of a *nested chain* [58], the hotspot propagates to its upstream functions as the caller has to wait until the return of the callees [16]. (2) Existing frameworks like OpenFaaS [42] and OpenWhisk [43] share a same gateway design: all function invocations are received by a frontend gateway, and then forwarded to independent backends where function code executes. Managing the waiting queue of a saturated function would consume many resources of the gateway, thereby degrading the invocation speeds of all other functions. Henceforth, the impact on functions ⑦-⑨ is more pronounced than that on ①-⑤ (Figure 4(b)). The colocation scheme should consider the end-to-end call path of each invocation to guarantee SLA.

Observation 5: *Restoring propagation: The local control of partial interference suffers from impact propagation.*

The dotted lines in Figure 4 also show how local isolation control restores the latencies of all functions. After we move the corunner to another server socket, not only the interfered function (i.e., ①compose-post in Figure 4(a) or ⑥compose-post in Figure 4(b)) but also the other functions have their invocation frequencies restored. While local interference control is helpful for decreasing the 99*th* percentile latency of the local function, the other functions' latencies increase due to the restored invocations.

Observation 6: *Predictability: Accurate partial interference prediction is enabled by function-level profiles, thereby improving the QoS of workloads.*



(a) End-to-end ave. IPC (b) 99%ile latency (c) Actual latency

Figure 5. (a) and (b) violin plots show the probability density of the prediction errors. For predicting IPCs or tail latencies, *function-level* profiles produce much lower medians and variances than monolithic *workload-level* profiles. (c) Highly accurate prediction improves the QoS of *social network*.

Formulating a complex workload into small functions exposes the inner structures of the workload, and profiling at the *function-level* also becomes easier due to the containerized deployment. Since *function-level* profiles keep much richer execution information than *workload-level* profiles [23], it becomes possible to build a performance predictor that is both accurate and lightweight.

We compare the prediction accuracies achieved by *workload-level* and *function-level* profiling by feeding their profiles into the same machine learning model. In particular, we train the learning model using traces of multifunction workloads (*feature-generation* [25] and *e-commerce* [36]), and evaluate the prediction errors using *social network*. In *workload-level* profiling, functions are integrated together into a single container as a monolithic workload. Partial interference is generated by a random combination of the other workloads in

Table 2. A comparison between Gsight and previously developed methods. *FPS* (frames per second) is a metric for games. *M*: the number of LS workloads; *N*: the number of SC/BG workloads. *B*: the number of microbenchmarks. *CS*: the number of corunning workloads built from a training set. *D*: the one-time cost for building the initial training dataset.

		CLITE [45]	GAugur [31]	SMiTe [63]	Prophet [5]	2-phase [64]	Paragon [11]	Bubbleup[35]	Pythia [55]	ESP [37]	Gsight[this paper]
Resource	Multi-server						✔		✔	✔	✔
	Multi-dimensional Res.	✔	✔	✔	GPU	✔	✔			✔	✔
	Temporally-varied partial										✔
	Spatially-varied partial										✔
Workload	> 2 workloads	✔	✔		✔	✔	✔		✔	✔	✔
	LS+LS	✔	✔								✔
	LS+SC/BG	✔		✔	✔			✔	✔		✔
	SC+SC/BG	✔				✔	✔			✔	✔
	Functions										✔
Prediction	End-to-end model										✔
	QoS metric	-	FPS	IPC	Time	JCT	Interference	Time, th.p.	IPC	JCT	IPC, JCT
	Prediction error (%)	-	5	1.79	5.1-5.9	0.3	5.3	0.7-2.2	3.4	> 3	1.67
Overhead	Profiling cost	$\leq 20(M+N)$	$B(M+N)$	$B(M+N)$	$M+N$	$CS(M+N)$	$2(M+N)$	$BM+N$	$BM+N$	$M+N$	$D+(M+N)$
Date	Year	2020	2019	2014	2017	2016	2013	2011	2018	2017	2020

FunctionBench [25]. All training data are collected once per second for five minutes, following the design of our *Gsight* approach. Figure 5 shows violin plots of the distributions of the prediction errors yielded by various learning models (including *K-Nearest Neighbor (KNN) Regression*, *Logistic Regression (LR)*, *Random Forest Regression (RFR)*, *Support Vector Regression (SVR)*, and a *Multi-layer Perceptron Neural Network (MLP)*). As shown, whether predicting IPCs or tail latencies, the predicting models trained using *function-level* profiles produce an average median that is 2× lower than that obtained using *workload-level* profiles. Their maximal difference can reach 4×. Furthermore, *function-level* profile-based predicting is more stable. Its variance is on average 13× (up to 42×) lower than that of *workload-level* profiling. An accurate predicting model enables better QoS during actual execution. Figure 5(c) shows that under the same settings as those in § 6, our *Gsight* employing *RFR* generates a much lower the 99th percentile latency than the others.

2.2 Implications

Performance prediction for serverless workloads is challenged by highly volatile partial interference (**Observation 1**). To provide an accurate prediction, the utilized prediction method should be able to address the *observations* above. First, it should harness the spatially- and temporally-varied overlap among functions for predicting (**Observations 2 and 3**). Second, it should be a holistic method that captures the end-to-end performance variation caused by partial interference (**Observations 4 and 5**). Third, it should exploit the profiles of functions extensively to achieve high precision (**Observation 6**).

Prior work on predictors focused on full interference and serverful workloads (Table 2), but the features of partial interference were left unaddressed. Applying such predictors directly to serverless workloads would produce considerable errors (§ 6). While reactive approaches of job scheduling and resource allocation suffer due to the high dynamicity of functions, it would be particularly desirable to have a novel predictor for serverless workloads.

3 Predicting with Gsight

3.1 Design

The insight of *Gsight* is that the QoS prediction accuracy under partial interference can be significantly improved through a "spatial-temporal interference"-aware incremental learning model, which can converge quickly by training on the profiles of functions along an end-to-end call path.

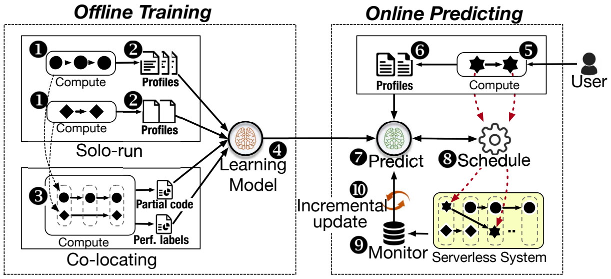


Figure 6. The design of *Gsight*. The learning model takes function profiles on the request call path and the partial interference code as input and produces QoS prediction results.

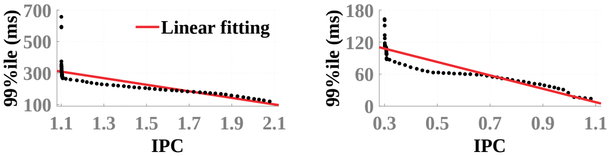
Figure 6 highlights the design overview of *Gsight*. The core of the predictor is an *incremental learning* model❹, which is trained by learning on both offline data(❷❸) and sequentially-available online data(❾❿). The *incremental learning* method can continuously improve the prediction accuracy during system operation; thus, it is particularly suitable for production systems. However, a less-trained model in the initial stage would generate poor predictions (i.e., "underfitting"). We mitigate this problem using accurate function-level profiles and an elaborately designed predicting model. First, each function of each workload is characterized under its solo-run❶, generating a profile❷ containing both system-layer and microarchitecture-layer metrics. Then, the workloads are colocated under partial interference to generate performance labels❸. Each partial interference scenario corresponds to a *partial interference code*, which consists of two vectors capturing the spatial overlap and temporal overlap among functions. In particular, the spatial overlap vector indicates the codeployment locations of functions, and the temporal overlap vector represents the time overlap among functions. *Function-level profiles*, the *partial interference code* and *labels* constitute the initial training dataset.

In the online phase, after a user submits a new serverless workload❺, we first profile its functions under a solo-run❻. Then, the scheduling algorithm❽ searches for its optimal placement by calling the prediction model❼ iteratively. In each iteration, the prediction model❼ infers its QoS by taking the solo-run profiles of all workloads and their partial interference codes. After the functions are deployed in the system, the real performance of the system is monitored❾ and is used to update the learning model incrementally❿.

3.2 Profiling

Interference profiling can be conducted in two ways: the *pairwise colocating way* measures the direct interference between two colocated workloads in terms of how much the QoS metric degrades [33]; the *microbenchmark way* colocates tailored microbenchmarks with functions for testing their sensitivity [31, 35, 55], then estimates the QoS based on the obtained microbenchmarking results. However, due to their high profiling costs, neither of them is adopted by *Gsight*. We instead design a *solo-run way* that collects the runtime status information of functions on dedicated physical servers and estimates the interference based on their resource utilization profiles. Although the *solo-run way* does not characterize the interference directly, (1) it avoids the profiling cost generated by pairwise co-locations or microbenchmarks, thus reducing the profiling overhead significantly; (2) a high prediction precision can be achieved by the elaborate design of the prediction model and the accurate function-level profiles.

We adopt a *non-intrusive* method that only monitors the *system-layer* and *microarchitecture-layer* metrics of the application and does not require instrumentation or modifications to application source code. In the *system layer*, we collect not only the configurations of resource allocation but also their actual utilization ratios, including the number of CPU cores, memory, network, memory bandwidth, and disk I/O. In the *microarchitecture layer*, we record the instruction execution-related measures of the CPU, which represent the execution status of the workload. In particular, we consider two important factors in the system design: parallelism and locality. Parallelism includes instruction-level-parallelism (ILP) and memory-level-parallelism (MemLP), where ILP measures the efficiency of the pipeline, and MemLP is a measure of the efficiency of concurrent access such that a higher MemLP value implies higher concurrency. The locality metric represents the locality of instructions and data measured by cache misses, including the cache and translation lookaside buffer (TLB). We collect the number of misses per thousand instructions (MPKI).



(a) Social network (b) E-commerce

Figure 7. The latency-IPC curve for LS services under various partial interference.

To train the machine learning model, we further collect the actual performance of workloads as labels for the training dataset. For example, *IPC* and *tail latency* metrics are used for LSes, and *JCT* metric is used for SCs. By creating partial interference scenarios through varying the QPS of LS workloads and the temporal or spatial overlap among colocated workloads (§ 6), we record the corresponding 99th percentile latencies and IPCs and show their statistical relations in Figure 7. There exists a "knee" in the latency-IPC correlation curve: the 99th percentile latency has a strong correlation with the IPC after the knee but varies significantly before that. This implies that a well-trained IPC prediction model would also have high accuracy when predicting tail latency after the knee but would perform differently before that the knee. As the tail latency varies under the same configuration and the IPC measurements are more immune to system noise [24, 55, 57, 63], our evaluations in § 6 show that the prediction model achieves higher accuracy for the IPC than the tail latency, and the tail latency prediction error falls from 28.6% to 18.7% after removing low IPC samples.

Table 3. Correlation between metrics and performance

Metric	Pearson	Spearman	Metric	Pearson	Spearman
Branch MPKI	-0.60	-0.72	L1I MPKI	0.38	0.45
Context-switches	0.96	0.96	L2 MPKI	0.54	0.81
MLP	0.02	-0.03	L3 MPKI	0.54	0.78
L1D MPKI	-0.37	-0.56	DTLB MPKI	-0.75	-0.85
ITLB MKPI	-0.38	-0.54	IPC	0.85	0.89
CPU utilization	0.81	0.82	LLC	0.83	0.84
Memory utilization	0.11	0.19	Memory IO	0.04	0.05
Network bandwidth	0.94	0.94	Disk IO	0.08	0.08
transmit(TX)	-0.16	-0.19	CPU frequency	-0.57	-0.68
receive(RX)	-0.60	-0.61	-	-	-

Incorporating all the available systems and microarchitecture metrics into the model is problematic because irrelevant metrics can easily result in overfitting, causing the model to convey poor accuracy. Moreover, a large number of input dimensions also leads to a long prediction time. Thus, performance metrics that are highly correlated with inherent characteristics or interference must be used. To do so, we use the *Pearson Correlation Coefficient* [2] and *Spearman Correlation Coefficient* [47] to evaluate the correlations between the target QoS and the performance metrics. The larger each coefficient is, the more the metrics are correlated with the performance. As shown in Table 3, we list the average correlations between the metrics and performance separately, and for conducted experiments, we exclude the ones with absolute correlation values less than 0.1. Finally, we choose the remaining 16 metrics as the inputs for the incremental learning model. For a distributed workload consisting of multiple functions, we collect the measures of all its functions.

3.3 Incremental Learning

An accurate prediction model can be obtained after conducting training with a sufficient number of samples. However, the collection of a large number of traces (including both the solo-run profiles and the corun labels) at the beginning is difficult. It is also not feasible to learn a prediction model in just one training step due to the continuous evolution of workloads. To solve the problem, we adopt the *online incremental learning* method for training the prediction model: we first prepare a small dataset of workload metrics and their corun QoS. We train the learning model using this dataset and use it for performance prediction thereafter. During the execution period, we extend the dataset with the newly generated metrics and their actual corun QoS repeatedly. The learning model is updated by the new data for better prediction accuracy.

We design a regression model (denoted by *RM*), which can predict the QoS of an arbitrary number of colocated workloads under partial interference. The model for predicting application A's performance under the interference of $B, C, ...$ (i.e., $P_{A\cup\{B,C,...\}}$) is described as follows:

$$P_{A\cup\{B,C,...\}} = RM(\underbrace{R_A, R_B, R_C, ...}_{AllocatedRes}, \underbrace{U_A, U_B, U_C, ...}_{Utilization}, \underbrace{D_A, D_B, D_C, ...}_{Delay}, \underbrace{T_A, T_B, T_C, ...}_{Lifetime}), \tag{1}$$

where $(R_A, R_B, R_C, ...)$ represents the resource allocation vectors configured for the workloads $(A, B, C, ...)$, $(U_A, U_B, U_C, ...)$ denotes the actual utilization ratios of various resources and performance monitoring counters collected in the *system layer* and *microarchitecture layer* as in Table 3, $(D_A, D_B, D_C, ...)$ is a start delay vector, and $(T_A, T_B, T_C, ...)$ represents the lifetime lengths of solo-run SC/BG workloads, which are set to 0 when utilizing LS workloads.

Temporal overlap coding: The start delay vector (D_A, D_B, $D_C, ...$) codes the time overlap among workloads. In particular, we have $D_A = 0$, and $(D_B, D_C, ..)$ represents the start delay of workload $(B, C, ...)$ compared to that of A. That is, workload A is ahead of i at time D_i if $D_i > 0, \forall i \in \{B, C, ...\}$, or $|D_i|$ later than i if $D_i < 0, \forall i \in \{B, C, ...\}$.

For the example in Figure 2, since workload #2 is ahead of workload #3 with respect to t_{delay}, we have the following temporal overlap code for workloads #2 and #3: $(D_2, D_3) = (0, t_{delay})$.

Spatial overlap coding: Both R_i and U_i ($\forall i \in \{A, B, C, ...\}$) are two-dimensional vectors that are specially designed for incorporating the spatial overlap information among application functions. In particular, we have:

$$U_i = \begin{vmatrix} u^1_{i1} & u^2_{i1} & ... & u^{16}_{i1} \\ u^1_{i2} & u^2_{i2} & ... & u^{16}_{i2} \\ ... & ... & ... & ... \\ u^1_{iS} & u^2_{iS} & ... & u^{16}_{iS} \end{vmatrix}, \tag{2}$$

where S denotes the number of servers in the system, and u^k_{il} $(1 \le l \le S, 1 \le k \le 16)$ represents the kth metric collected for i's function on server l. If there exists no function on l, then $(u^1_{il}, u^2_{il}, ..., u^{16}_{il})$ are all set to 0s. Since all matrices of U_i ($\forall i \in \{A, B, C, ...\}$) have the same numbers of rows and columns, functions from different workloads that share the same number of rows are implied to be colocated on the same server. In this way, we are able to encode the spatial overlap in the prediction model. If there exists more than one function located on the same server, we aggregate their metrics together, generating a "virtual larger function". Likewise, the vector R_i is designed in the same way.

$$\begin{vmatrix} u^1_{1,1} & .. & u^{16}_{1,1} \\ u^1_{1,\{34\}} & .. & u^{16}_{1,\{34\}} \\ u^1_{1,\{25\}} & .. & u^{16}_{1,\{25\}} \\ u^1_{1,6} & .. & u^{16}_{1,6} \\ u^1_{1,\{789\}} & .. & u^{16}_{1,\{789\}} \end{vmatrix} \tag{3}$$

$$\begin{vmatrix} u^1_{2,10} & .. & u^{16}_{2,10} \\ u^1_{2,11} & .. & u^{16}_{2,11} \\ 0 & .. & 0 \\ u^1_{2,12} & .. & u^{16}_{2,12} \\ 0 & ... & 0 \end{vmatrix} \tag{4}$$

$$\begin{vmatrix} u^1_{3,13} & .. & u^{16}_{3,13} \\ 0 & .. & 0 \\ 0 & .. & 0 \\ u^1_{3,14} & .. & u^{16}_{3,14} \\ 0 & ... & 0 \end{vmatrix} \tag{5}$$

For the three workloads in Figure 2, their spatial overlap codes (U_1, U_2, U_3) are shown in Matrices (3-5), respectively. As functions ①, ⑩ and ⑬ are deployed on the same server #1, they are all located in the first row vectors in U_1, U_2 and U_3, respectively. Furthermore, since ③ and ④ of workload #1 are deployed together on server #2, we simulate a virtual function "{34}" and measure the average of each metric (out of 16). This is also applicable to "{25}" and "{789}". The third and fifth row vectors of U_2 are 0s because functions of workload #2 are not deployed on server #3 and #5.

The prediction model has different forms for the different combinations of LS, SC and BG workloads.

LS+LS workloads: If there are *no BG and SC jobs* involved in the colocation process, we have $D_i = 0$ and $T_i = 0$ ($\forall i \in \{A, B, C, ...\}$); because the LS workloads are invoked repeatedly, the main interference factor is the QPS instead of the start delay. Moreover, $P_{A\cup\{B,C,...\}}$ represents the IPC or tail latency of A.

LS+SC/BG workloads: If both LS and SC (or BG) workloads are present in the co-locating, then the start delay of the first arriving SC (or BG) is set to 0, and the delays of the other SCs (or BGs) are set by comparing their arrival times to the first workload. Furthermore, we have $D_i = 0$ and $T_i = 0$ for i belonging to LS workloads. For example, *message-posting* in Figure 2 is an LS workload, and we have $D_1 = 0$ and $T_1 = 0$ for it.

SC+SC/BG workloads: If there are *no LS workloads* involved in colocation, then we have that the start delay of the first arriving SC (or BG) is equal to 0 and that $T_i \neq 0$ ($\forall i \in \{A, B, C, ...\}$). Moreover, $P_{A\cup\{B,C,...\}}$ represents the JCT of A.

BG+BG workloads: If there exists only BG workloads in the colocating, we do not call the prediction model because the performance requirements of BGs are lenient.

3.4 Learning Model

The prediction model can be constructed using either traditional machine learning (ML) or deep learning (DL) algorithms. Although deep learning algorithms may yield higher accuracy, they usually require many more training samples than traditional machine learning. Hence, we choose the traditional ML approach, particularly *Incremental RFR (IRFR)*, to build our model, because *RFR* is simple, easy to implement and suitable for various high-dimensional, various types of continuous variables [7]. *RFR* is an ensemble model that constructs a multitude of decision trees and combines their outputs together for classification and regression [32]. It is robust against the overfitting of the training data because it incorporates randomness: each decision tree is trained on *a random subspace of the data*, and only *a random subset features* are selected. Our evaluations in § 6 show that *IRFR* achieves high prediction accuracy (the error ≤ 1.71%), outperforming the other representative machine learning algorithms, including *incremental KNN (IKNN)* [8], *incremental LR (ILR)* [29], *incremental SVR (ISVR)* [51], and *incremental MLP (IMLP)* [19]. We evaluate the impurity-based importance of the 16 metrics in *RFR* model and find that in addition to *disk IO*, all other metrics are informative (Figure 8).

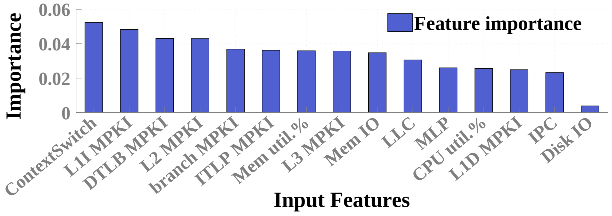


Figure 8. The impurity-based importance of the 16 metrics.

4 Scheduling using Gsight

We present a function scheduling case study to validate the effectiveness of *Gsight*. The scheduling algorithm aims to maximize resource efficiency by deploying function instances on a minimum number of active servers while guaranteeing the QoS of colocated workloads. This mechanism is activated whenever a new workload is submitted or a previously submitted workload scales beyond the current function instances. When the invocation load varies but does not yet cause scaling-out operations, it is also possible to further optimize resource efficiency by rescheduling the existing instances.

Suppose that there are S servers in the computing system, and the newly arrived workload consists of M functions; then, the maximum search space for deployment is S^M. Although *Gsight*'s inference efficiency is high, the brute force approach that explores all possibilities in the search space is rather time consuming, due to the largeness of the search space. Its time complexity is $O(PS^M)$, where P refers to the inference time for a colocation setting. To reduce the scheduling time, we design a *binary search-based scheduling* algorithm, which attempts a half spatial overlap if the full overlap schedule violates the SLA. Hence, it can reduce the time complexity to $O(MP\log S)$. In each attempt, since it is still costly to evaluate all the possible overlaps even with the constraint of the half overlap, we instead check only one configuration (i.e., by scheduling the function with maximum resource requirements to the server with the most available resources) to see if it can satisfy the SLA requirement.

5 Implementation

5.1 OpenFaaS Implementation

We implement *Gsight* and integrate it into OpenFaaS, an event-driven serverless computing platform based on Kubernetes [27]. Our framework adopts the controller-agent architecture: a centralized *Gsight controller* is deployed in the master node, and each slave node is equipped with a *Gsight agent*. The *Gsight controller* is a submodule of the OpenFaaS controller that monitors the function statuses and makes scheduling decisions. It consumes around 0.4 cores and 600MB of memory. Each *Gsight agent* collects the microarchitecture metrics yielded by function instances using tools such as perf (Linux performance counters profiling) and pqos-msr (Intel RDT Utility) and receives instructions from the controller for allocating resources (e.g., CPU cores, LLC, memory bandwidth).

5.2 Function Startup Latency

In serverless computing, a cold start happens when an inactive function is invoked. The startup process for deploying the function along with any dependencies it may have generates high latency for function execution. It also causes resource contention on other colocated functions. To evaluate such interference, we can just treat the startup process as an ordinary phase of the function execution and the startup profile is a part of the function profile. If an invocation experiences a cold start, the predictor utilizes function profiles containing the startup phase. Otherwise, if the cold start is avoided by pre-warmed functions or customized guest kernels [13], the predictor takes function profiles without startup phase as input. In this way, the QoS can still be predicted accurately under the startup interference.

6 Evaluation

6.1 Methodology

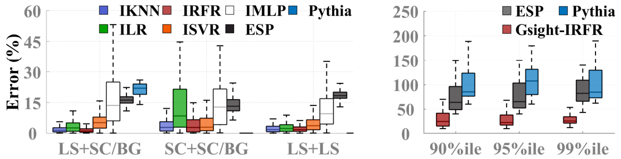
Table 4. Experimental testbed configuration.

Component	Specification	Component	Specification
CPU model	Intel Xeon E7-4820v4	Shared LLC Size	25MB
Number of sockets	4	Memory Capacity	256GB
Processor BaseFreq.	2.00 GHz	Operating System	Ubuntu 14.04.5LTS
Threads	80 (40 physical cores)	SSD Capacity	960GB
Private L1&L2 Cache	64 KB and 256 KB	Number of Nodes	8

Testbed and Workloads: Table 4 summarizes our testbed platform used for evaluating *Gsight*. We use the workloads in ServerlessBench [58] and FunctionBench [25] for evaluation. As they were originally implemented on OpenWhisk [43] or AWS Lambda [1], we redevelop and deploy them on OpenFaaS[42]. Similar to [21, 61], we also port the *social network* [16] and *e-commerce* [36] to OpenFaaS. Each microservice in *social network* (or *e-commerce*) is transformed into one or more functions according to its features.

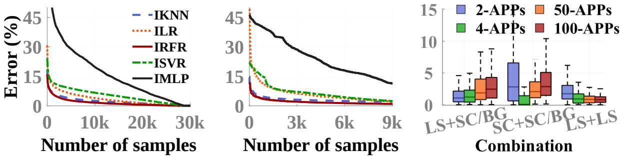
These services are triggered by dynamic invocations simulated using the production trace from the Azure Function [49], where the invocations per hour illustrate diurnal and weekly patterns. We observe that cold starts occur frequently during the request load rises, i.e., an average of 8 cold starts per minute are observed under our settings.

Competing Predicting and Scheduling Policies: Table 2 shows previously developed interference prediction methods. We compare *Gsight* with *ESP* [37] and *Pythia* [55] because they are the state-of-the-art approaches, sharing the same QoS metrics as ours. For the scheduling comparison, while *Pythia* employs the *Best Fit* algorithm that places the workload on the server with the smallest amount of headroom, we further design a *Worst Fit* algorithm that always schedules functions with maximum resource requirement to the server with the maximum amount of available resources until an SLA violation occurs.



(a) IPC (b) Tail latency

Figure 9. The prediction errors of (a) IPC and (b) tail latency.



(a) Serverful (b) Serverless (c) Multi-apps

Figure 10. The prediction error: (a) The incremental updating process for serverful workloads. (b) The incremental updating process for serverless workloads. (c) The distribution of prediction error under multiworkload colocations.

6.2 Prediction Error

The prediction error is defined as $|\hat{P} - P|/P$, where $\hat{P}$ and P denote the predicted QoS and the actual value, respectively. Any slight decrease in prediction error would significantly lower the amount of resources reserved for colocated workloads significantly, saving millions of dollars when building a large-scale datacenter for cloud providers. Next, we investigate how the learning model, workload types, function-level profiles and number of colocated workloads affect the prediction accuracy and convergence speed of *Gsight*.

High prediction accuracy: *Gsight* employing the *IRFR* learning model is highly accurate in predicting the performance of either LS or SC workloads. We first evaluate the prediction errors under different learning models and workload colocations (including LS+LS, LS+SC/BG and SC+SC/BG). Figure 9 shows the prediction errors with respect to IPC (Figure 9(a)) and tail latency (Figure 9(b)) for the various models. We make four main observations. First, *IRFR* is the most suitable learning model for *Gsight*. While the mean prediction errors of *IKNN*, *IRFR* and *ISVR* are less than 6% in all experiments, the prediction error of IPC generated by *IRFR* is as low as 1.71% in the LS+BG/SC colocation, outperforming the other models significantly. Second, *Pythia* and *ESP* produce higher prediction errors than the others in all experiments because *Pythia* is not able to handle the propagation effect of partial interference, and *ESP* only uses four microarchitecture metrics (i.e., the IPC, L2 access rate, L3 access rate and memory bandwidth) during model training. Third, *Gsight* achieves high prediction accuracies in all colocations. Even in the worst case (SC+SC/BG), *IRFR*'s prediction error is still less than 5%. Fourth, it is more challenging to predict tail latency than IPC. Although *Gsight* is more accurate than *Pythia* and *ESP*, its prediction error is still 28.6% when predicting tail latency (Figure 9(b)).

Fast and stable convergence: Function-level profiles enable *Gsight* to converge quickly. Its precision is also stable after convergence. To evaluate the convergence speed, we train the same learning model using serverful and serverless samples. In particular, serverful samples are generated at the workload-level by running serverful benchmarks on our testbed, including Sparkbench [30], bigdatabench [54], Redis [46], Apache Solr [3] and MongoDB [38]. We collect 37,053 samples in total. Serverless samples are collected at the function-level from serverless benchmarks.

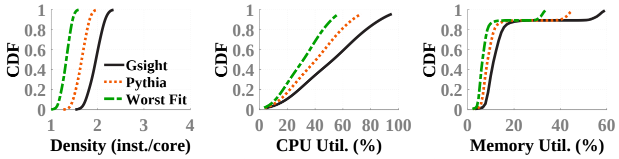
We observe that *Gsight* converges much faster in a serverless system than that in a serverful system. In particular, the prediction errors in the serverless system after 1,000, 2,000, and 3,000 samples are just 3.41%, 2.55% and 2.09%, respectively, while they are 6.5%, 4.74% and 3.75% in the serverful system (Figure 10(a)). Moreover, *IRFR* must be updated with 3500, 5450 and 7430 samples to achieve the same prediction error as that obtained when trained with 1,000, 2,000, and 3,000 samples in a serverless system. Hence, function-level profiles in a serverless system help to greatly improve the convergence speed of *Gsight*, which is at least 3× faster than in a serverful system. Figure 10(b) also shows that the prediction errors produced by *IRFR* after 3,000 samples are kept below 2.09%. It even approaches to 1% after 9,000 samples. Hence, the precision is also stable after convergence.

The number of colocated workloads does not affect the prediction precision of *Gsight*. We further evaluate the prediction accuracy of *Gsight* by varying the number colocated workloads. Figure 10(c) shows that an increase in the number of workloads does not degrade the accuracy much. The prediction error is always less than 3% in any combinations. In particular, the average error even becomes lower when colocating many LS+LS workloads.

We also evaluate how quickly *Gsight* will recover if the workload drastically differs from the training data. We divide the training data into two groups: *I/O intensive* (e.g., social network) and *CPU intensive* (e.g., ML serving). Figure 13 shows that *IRFR* trained using *I/O intensive* samples produces an error of 43.9% for predicting the IPC of *CPU intensive*, since *CPU intensive* workloads are about 1.6× of the IPC of *I/O intensive* ones. However, the error declines to 4.6% quickly after the incremental update with 1,000 samples.

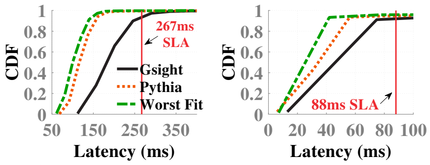
6.3 Scheduling Results

***Gsight scheduling* further improves the density of workloads in a high-density serverless system.** *Function density* measures the number of function instances deployed per core in the system. It dynamically changes with a varied invocation load due to the autoscaling capability of OpenFaaS. Figure 11(a) shows the CDF of the function density recorded over time. We see that, benefiting from accurate predictions, *Gsight* improves the function density by averages of 18.79% and by 48.48% over those of *Pythia* and *Worst Fit*, respectively. This demonstrates that *Gsight* is particularly suitable for high-density serverless systems.



(a) Function density (b) CPU utilization (c) Memory utilization

Figure 11. Scheduling results in terms of function density, CPU and memory utilization.



(a) Social network (b) E-commerce

Figure 12. The CDF of 99th percentile latency for an LS workload.

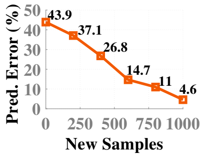


Figure 13. Model recovery under new samples.

A higher function density leads to higher resource utilization. Figure 11(b) shows that *Gsight* can improve CPU utilization by 30.02% and 67.51% on average compared with that of *Pythia* and *Worst Fit*, respectively. Likewise, Figure 11(c) shows that *Gsight* can improve memory utilization by 31.04% and 76.91% on average.

***Gsight* scheduling can guarantee the SLA of workloads.** For LS workloads, we define their SLAs based on the following principle: *each LS runs at its maximum allowable request load without interference over 30 minutes, and we record the 99th percentile latency per second and set the average as the SLA*. For example, the 99th percentile latencies stated in SLAs of *social network* and *e-commerce* are 267ms and 88ms, respectively. As the IPC prediction model exhibits smaller errors than the tail latency model in Figure 9, we adopt the IPC model for scheduling by transforming the tail latency in SLA into IPC according to their correlation curve (using the average if there are multiple IPCs; Figure 7). Figure 12 shows that *Gsight* still guarantees their SLAs 95.39% and 93.33% of the time, respectively. This is because a weak guarantee mainly occurs when the IPC is low, which is true for just 4.1% of all samples (Figure 7). Note that *Gsight* is orthogonal to the buffer-based or reactive-control tail latency optimization approaches [6, 20, 65], which suggests that a stronger SLA guarantee can be achieved when integrating them together.

6.4 Overhead and Scalability

Offline profiling cost: The solo-run way of profiling generates a cost of $O(D + (M + N))$, where M represents the number of LS workloads, N represents the number of SC/BG workloads, and D is the one-time cost for building the initial training dataset. We develop an open-loop load generator, which can test each LS workload under various access loads and generate profiles within 5 minutes. In a real system, the profiling can also finish in a short time if using the load generator. Otherwise, it would last one period (e.g., a day in a diurnal pattern). Since LS workloads are invoked repeatedly, one profiling step for them can be reused for later invocations. Each SC or BG is profiled by running it once in a dedicated environment. Fortunately, the majority of BGs and a large proportion of SCs (≥ 40% of bigdata applications [15, 59]) are recurring. Collecting the initial training dataset with 3,000 samples takes us less than 2 person-hours, including deployment, monitoring, and training. In particular, the training process takes only < 10 minutes.

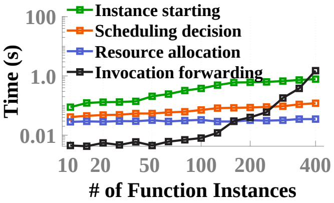


Figure 14. The overhead of the scheduling process.

The number of dimensions in *RM* varies with the number of colocated workloads. For training the *IRFR* model, we simply fix the number of workloads in $P_{A\cup\{B,C,...\}}$ as n, i.e., the maximum allowable colocations in the system. If the actual colocations are less than n, we use zeros to pad out the values. As both the start delay vector and the lifetime vector have n dimensions, and both the resource allocation vector and utilization vector have $16nS$ dimensions, the overall number of dimensions of each piece of training data is $32nS + 2n$. In our experiments, we set $n = 10$.

Online running cost and scalability: The online scheduling process consists of four steps: *invocation forwarding*, *scheduling decision making*, *instance starting* and *resource allocation*. Figure 14 shows how the overhead increases with the number of function instances. We see that the *scheduling decision making* is highly efficient and only takes a few milliseconds when scheduling different numbers of instances. In particular, each inference takes an average of 3.48 milliseconds and each incremental update takes an average of 24.784 milliseconds. The most time is spent on starting instances.

Hence, it is necessary to accelerate the startup process to improve the time efficiency (e.g., [13]). The invocation forwarding module of OpenFaaS is stable and efficient when the number of instances is less than 110, but slows down rapidly after 120 instances due to the bottleneck of the gateway. Hence, a scalable gateway is also required for better scalability (e.g., [21]).

In a large-scale system, the number of colocations experienced by a workload is generally limited. For example, since a vast majority of workflows contain fewer than 10 functions [49], the number of physical servers on which a workflow is deployed is also typically fewer than 10. In this case, the number of dimensions (i.e., $32nS+2n$) of the predicting model could be fewer than a few hundreds and *Gsight* can still have a high predicting efficiency. In the worst case, if a workflow consists of a lot of functions and spans over hundreds or thousands of servers, *Gsight* may do not scale up well due to the massive dimensions and scheduling search space. Policies like dimensionality reduction (e.g., PCA) and hierarchy scheduling can be explored to improve *Gsight* in future work.

7 Related Work

Regarding serverful systems, interference has been studied extensively, and predicting-based colocation can improve resource utilization while guaranteeing the tail latency of LS workloads. Table 2 summarizes prior approaches for predicting workload performance under interference. These methods commonly rely on workload-level profiling to improve the prediction precision. For example, the "sensitivity curve transformation" methods (e.g., [35], [57], [63], [55], [31]) characterize interference sensitivity using microbenchmark rulers. Quasar [12] uses collaborative filtering techniques for prediction. Other predictors try to identify the specific features that are relevant to performance, e.g., query-level predictions [4], the number of cycles-per-instruction (CPI) [62], and cluster utilization counters[56]. For predicting SC's performance under colocation, prior work evaluated interference at shared LLC, instruction execution units, I/O or memory bandwidth [11, 18, 52] and employed techniques such as collaborative filtering [11], statistical methods [53], regression analysis [64], and machine learning [37, 60] to predict slowdown. For colocated LS workloads, GAugur [31] supports the efficient colocation of online games using a machine learning-based prediction model.

In summary, prior work has explored the design space of interference predictors. While they work well under full interference, how to extend their approaches by considering complex function call structures as well as partial interference in a serverless system is still an unresolved problem.

8 Conclusions

In serverless computing, workloads continue to become increasingly complex due to small-sized functions. This leads to a variety of partial interference types that have not been studied extensively by existing solutions. This work first characterizes the features of partial interference and presents *Gsight*, which takes an incremental learning model to predict workload performance under partial interference. The experimental results show that *Gsight* employing the *IRFR* machine learning model yields very high accuracy. By applying *Gsight* in the context of a scheduling use case, we also demonstrate that the proposed approach can significantly improve resource efficiency.

9 Acknowledgments

Our sincere thanks and appreciation goes to the SC reviewers, for their extremely insightful and helpful comments and suggestions that have significantly improved the quality of this paper. We thank Dr. Tao Li for his significant contribution to this work. This work is supported by the National Key Research and Development Program of China No. 2016YFB1000205; the National Natural Science Foundation of China under grant 61872265; the new Generation of Artificial Intelligence Science and Technology Major Project of Tianjin under grant 19ZXZNGX00010; State Key Lab. of Computer Architecture (ICT, CAS)-CARCHA202009; NUDT-PDL-6142110200405; State Key Lab. of Communication Content Cognition fund No.A32001.

References

[1] 2020. AWS Lambda. https://aws.amazon.com/lambda/.

[2] Y. Amannejad, D. Krishnamurthy, and B. Far. 2016. Predicting Web service response time percentiles. In *2016 12th International Conference on Network and Service Management (CNSM)*. 73–81.

[3] ApacheSolr. 2020. Solr is the popular, blazing-fast, open source enterprise search platform built on Apache Lucene. https://lucene.apache.org/solr/.

[4] Quan Chen, Zhenning Wang, Jingwen Leng, Chao Li, Wenli Zheng, and Minyi Guo. 2019. Avalon: Towards QoS Awareness and Improved Utilization Through Multi-resource Management in Datacenters. In *Proceedings of the ACM International Conference on Supercomputing* (Phoenix, Arizona) *(ICS '19)*. ACM, New York, NY, USA, 272–283.

[5] Quan Chen, Hailong Yang, Minyi Guo, Ram Srivatsa Kannan, Jason Mars, and Lingjia Tang. 2017. Prophet: Precise QoS Prediction on Non-Preemptive Accelerators to Improve Utilization in Warehouse-Scale Computers. *SIGOPS Oper. Syst. Rev.* 51, 2 (April 2017), 17–32.

[6] Shuang Chen, Christina Delimitrou, and José F. Martínez. 2019. 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* (Providence, RI, USA) *(ASPLOS '19)*. ACM, New York, NY, USA, 107–120.

[7] Raphael Couronne, Philipp Probst, and Anne-Laure Boulesteix. 2018. Random forest versus logistic regression: a large-scale benchmark experiment. *BMC Bioinformatics* 19 (2018), 270:1–270:14. Issue 270.

[8] T. Cover and P. Hart. 2006. Nearest Neighbor Pattern Classification. *IEEE Trans. Inf. Theor.* 13, 1 (Sept. 2006), 21–27.

[9] Matt Crane and Jimmy Lin. 2017. An Exploration of Serverless Architectures for Information Retrieval. In *Proceedings of the ACM SIGIR International Conference on Theory of Information Retrieval* (Amsterdam, The Netherlands) *(ICTIR '17)*. Association for Computing Machinery, New York, NY, USA, 241–244.

[10] Yan Cui. 2019. How to build a social network on serverless. https://www.apidays.co/barcelona2019/.

[11] Christina Delimitrou and Christos Kozyrakis. 2013. QoS-Aware Scheduling in Heterogeneous Datacenters with Paragon. *ACM Trans. Comput. Syst.* 31, 4, Article 12 (Dec. 2013), 34 pages.

[12] Christina Delimitrou and Christos Kozyrakis. 2014. Quasar: Resource-efficient and QoS-aware Cluster Management. In *Proceedings of the 19th International Conference on Architectural Support for Programming Languages and Operating Systems* (Salt Lake City, Utah, USA) *(ASPLOS '14)*. ACM, New York, NY, USA, 127–144.

[13] Dong Du, Tianyi Yu, Yubin Xia, Binyu Zang, Guanglu Yan, Chenggang Qin, Qixuan Wu, and Haibo Chen. 2020. Catalyzer: Sub-Millisecond Startup for Serverless Computing with Initialization-Less Booting. In *Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems* (Lausanne, Switzerland) *(ASPLOS'20)*. Association for Computing Machinery, New York, NY, USA, 467–481.

[14] Simon Eismann, Joel Scheuner, Erwin van Eyk, Maximilian Schwinger, Johannes Grohmann, Nikolas Herbst, Cristina L. Abad, and Alexandru Iosup. 2020. A Review of Serverless Use Cases and their Characteristics. *arXiv:2008.11110* (2020).

[15] Andrew D. Ferguson, Peter Bodik, Srikanth Kandula, Eric Boutin, and Rodrigo Fonseca. 2012. Jockey: Guaranteed Job Latency in Data Parallel Clusters. In *Proceedings of the 7th ACM European Conference on Computer Systems* (Bern, Switzerland) *(EuroSys'12)*. Association for Computing Machinery, New York, NY, USA, 99–112.

[16] Yu Gan, Yanqi Zhang, Dailun Cheng, Ankitha Shetty, Priyal Rathi, Nayan Katarki, Ariana Bruno, Justin Hu, Brian Ritchken, Brendon Jackson, Kelvin Hu, Meghna Pancholi, Yuan He, Brett Clancy, Chris Colen, Fukang Wen, Catherine Leung, Siyuan Wang, Leon Zaruvinsky, Mateo Espinosa, Rick Lin, Zhongling Liu, Jake Padilla, and Christina Delimitrou. 2019. An Open-Source Benchmark Suite for Microservices and Their Hardware-Software Implications for Cloud & Edge Systems. In *Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems* (Providence, RI, USA) *(ASPLOS '19)*. ACM, New York, NY, USA, 3–18.

[17] Yu Gan, Yanqi Zhang, Kelvin Hu, Dailun Cheng, Yuan He, Meghna Pancholi, and Christina Delimitrou. 2019. 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*. ACM, 19–33.

[18] Sriram Govindan, Jie Liu, Aman Kansal, and Anand Sivasubramaniam. 2011. Cuanta: Quantifying Effects of Shared On-chip Resource Interference for Consolidated Virtual Machines. In *Proceedings of the 2Nd ACM Symposium on Cloud Computing* (Cascais, Portugal) *(SOCC '11)*. ACM, New York, NY, USA, Article 22, 14 pages.

[19] K. Hornik, M. Stinchcombe, and H. White. 1989. Multilayer Feedforward Networks Are Universal Approximators. *Neural Netw.* 2, 5 (July 1989), 359–366.

[20] Calin Iorgulescu, Reza Azimi, Youngjin Kwon, Sameh Elnikety, Manoj Syamala, Vivek Narasayya, Herodotos Herodotou, Paulo Tomita, Alex Chen, Jack Zhang, and Junhua Wang. 2018. PerfIso: Performance Isolation for Commercial Latency-Sensitive Services. In *2018 USENIX Annual Technical Conference (USENIX ATC 18)*. USENIX Association, Boston, MA, 519–532.

[21] Zhipeng Jia and Emmett Witchel. 2021. Nightcore: Efficient and Scalable Serverless Computing for Latency-Sensitive, Interactive Microservices. In *Proceedings of the 26th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS '21)*. ACM, New York, NY, USA, 127–144.

[22] Eric Jonas, Qifan Pu, Shivaram Venkataraman, Ion Stoica, and Benjamin Recht. 2017. Occupy the Cloud: Distributed Computing for the 99%. In *Proceedings of the 2017 Symposium on Cloud Computing*. Association for Computing Machinery, New York, NY, USA, 445–451.

[23] Kostis Kaffes, Neeraja J. Yadwadkar, and Christos Kozyrakis. 2019. Centralized Core-Granular Scheduling for Serverless Functions. In *Proceedings of the ACM Symposium on Cloud Computing* (Santa Cruz, CA, USA) *(SoCC '19)*. Association for Computing Machinery, New York, NY, USA, 158–164.

[24] Ram Srivatsa Kannan, Michael Laurenzano, Jeongseob Ahn, Jason Mars, and Lingjia Tang. 2019. Caliper: Interference Estimator for Multi-tenant Environments Sharing Architectural Resources. *ACM Trans. Archit. Code Optim.* 16, 3, Article 22 (June 2019), 25 pages.

[25] J. Kim and K. Lee. 2019. FunctionBench: A Suite of Workloads for Serverless Cloud Function Service. In *2019 IEEE 12th International Conference on Cloud Computing (CLOUD)*. 502–504.

[26] Y. Kim and J. Lin. 2018. Serverless Data Analytics with Flint. In *2018 IEEE 11th International Conference on Cloud Computing (CLOUD)*.

[27] Kubernetes. 2020. Kubernetes: Production-Grade Container Orchestration. https://kubernetes.io.

[28] Kubernetes. 2021. https://kubernetes.io/docs/reference/scheduling/policies/.

[29] Sun-In Lee, Honglak Lee, Pieter Abbeel, and Andrew Y. Ng. 2006. EfficientL1Regularized Logistic Regression. In *Proceedings of the 21st National Conference on Artificial Intelligence - Volume 1* (Boston, Massachusetts) *(AAAI'06)*. AAAI Press, 401–408.

[30] Min Li, Jian Tan, Yandong Wang, Li Zhang, and Valentina Salapura. 2015. SparkBench: A Comprehensive Benchmarking Suite for in Memory Data Analytic Platform Spark. In *Proceedings of the 12th ACM International Conference on Computing Frontiers* (Ischia, Italy) *(CF '15)*. ACM, New York, NY, USA, Article 53, 8 pages.

[31] Yusen Li, Chuxu Shan, Ruobing Chen, Xueyan Tang, Wentong Cai, Shanjiang Tang, Xiaoguang Liu, Gang Wang, Xiaoli Gong, and Ying Zhang. 2019. GAugur: Quantifying Performance Interference of Colocated Games for Improving Resource Utilization in Cloud Gaming. In *Proceedings of the 28th International Symposium on High-Performance Parallel and Distributed Computing* (Phoenix, AZ, USA) *(HPDC '19)*. ACM, New York, NY, USA, 231–242.

[32] Andy Liaw and Matthew Wiener. 2002. Classification and Regression by randomForest. *R News* 2, 3 (2002), 18–22.

[33] David Lo, Liqun Cheng, Rama Govindaraju, Parthasarathy Ranganathan, and Christos Kozyrakis. 2016. Improving Resource Efficiency at Scale with Heracles. *ACM Trans. Comput. Syst.* 34, 2, Article 6 (May 2016), 33 pages.

[34] A. K. Maji, S. Mitra, and S. Bagchi. 2015. ICE: An Integrated Configuration Engine for Interference Mitigation in Cloud Services. In *2015 IEEE International Conference on Autonomic Computing*. 91–100.

[35] Jason Mars, Lingjia Tang, Robert Hundt, Kevin Skadron, and Mary Lou Soffa. 2011. Bubble-Up: Increasing Utilization in Modern Warehouse Scale Computers via Sensible Co-locations. In *Proceedings of the 44th Annual IEEE/ACM International Symposium on Microarchitecture* (Porto Alegre, Brazil) *(MICRO-44)*. ACM, New York, NY, USA, 248–259.

[36] D. A. Menasce. 2002. TPC-W: A Benchmark for E-Commerce. *IEEE Internet Computing* 6 (05 2002), 83–87.

[37] N. Mishra, J. D. Lafferty, and H. Hoffmann. 2017. ESP: A Machine Learning Approach to Predicting Application Interference. In *2017 IEEE International Conference on Autonomic Computing (ICAC)*. 125–134.

[38] MongoDB. 2020. MongoDB: The database for modern applications. https://www.mongodb.com/.

[39] Nicolas Moutschen, Tom McCarthy, Jérôme Van Der Linden, and Dan Smith. 2020. AWS Serverless Ecommerce Platform. https://github.com/aws-samples/aws-serverless-ecommerce-platform.

[40] Ingo Müller, Renato Marroquín, and Gustavo Alonso. 2020. Lambada: Interactive Data Analytics on Cold Data Using Serverless Cloud Infrastructure. In *Proceedings of the 2020 ACM SIGMOD International Conference on Management of Data* (Portland, OR, USA) *(SIGMOD '20)*.

Association for Computing Machinery, New York, NY, USA, 115–130.

[41] Ripal Nathuji, Aman Kansal, and Alireza Ghaffarkhah. 2010. Q-clouds: Managing Performance Interference Effects for QoS-aware Clouds. In *Proceedings of the 5th European Conference on Computer Systems* (Paris, France) *(EuroSys '10)*. ACM, New York, NY, USA, 237–250.

[42] OpenFaaS. 2020. https://www.openfaas.com/.

[43] OpenWhisk. 2020. https://openwhisk.apache.org/.

[44] Jinsu Park, Seongbeom Park, and Woongki Baek. 2019. CoPart: Coordinated Partitioning of Last-Level Cache and Memory Bandwidth for Fairness-Aware Workload Consolidation on Commodity Servers. In *Proceedings of the Fourteenth EuroSys Conference 2019, Dresden, Germany, March 25-28, 2019*. 10:1–10:16.

[45] Tirthak Patel and Devesh Tiwari. 2020. CLITE : Efficient and QoS-Aware Co-location of Multiple Latency-Critical Jobs for Warehouse Scale Computers. In *2020 IEEE International Symposium on High Performance Computer Architecture (HPCA)*. IEEE, 193–206.

[46] Redis. 2020. Redis: an open source, in-memory data structure store. https://redis.io.

[47] Alice M. Richardson. 2015. Nonparametric Statistics: A Step-by-Step Approach. *International Statistical Review* 83, 1 (2015), 163–164.

[48] Jim Scott. 2020. Applying the Lambda Architecture with Spark. https://databricks.com/session/applying-the-lambda-architecture-with-spark.

[49] Mohammad Shahrad, Rodrigo Fonseca, Inigo Goiri, Gohar Chaudhry, Paul Batum, Jason Cooke, Eduardo Laureano, Colby Tresness, Mark Russinovich, and Ricardo Bianchini. 2020. Serverless in the Wild: Characterizing and Optimizing the Serverless Workload at a Large Cloud Provider. In *2020 USENIX Annual Technical Conference (USENIX ATC 20)*. USENIX Association, 205–218.

[50] Vaishaal Shankar, Karl Krauth, Kailas Vodrahalli, Qifan Pu, Benjamin Recht, Ion Stoica, Jonathan Ragan-Kelley, Eric Jonas, and Shivaram Venkataraman. 2020. Serverless Linear Algebra. In *Proceedings of the 11th ACM Symposium on Cloud Computing* (Virtual Event, USA) *(SoCC '20)*. Association for Computing Machinery, New York, NY, USA, 281–295.

[51] Alex J. Smola and Bernhard Schölkopf. 2004. A Tutorial on Support Vector Regression. *Statistics and Computing* 14, 3 (Aug. 2004), 199–222.

[52] Lavanya Subramanian, Vivek Seshadri, Arnab Ghosh, Samira Khan, and Onur Mutlu. 2015. The Application Slowdown Model: Quantifying and Controlling the Impact of Inter-application Interference at Shared Caches and Main Memory. In *Proceedings of the 48th International Symposium on Microarchitecture* (Waikiki, Hawaii) *(MICRO-48)*. ACM, New York, NY, USA, 62–75.

[53] Shivaram Venkataraman, Zongheng Yang, Michael Franklin, Benjamin Recht, and Ion Stoica. 2016. Ernest: Efficient Performance Prediction for Large-Scale Advanced Analytics. In *13th USENIX Symposium on Networked Systems Design and Implementation (NSDI 16)*. USENIX Association, Santa Clara, CA, 363–378.

[54] L. Wang, J. Zhan, C. Luo, Y. Zhu, Q. Yang, Y. He, W. Gao, Z. Jia, Y. Shi, S. Zhang, C. Zheng, G. Lu, K. Zhan, X. Li, and B. Qiu. 2014. BigDataBench: A big data benchmark suite from internet services. In *2014 IEEE 20th International Symposium on High Performance Computer Architecture (HPCA)*. 488–499.

[55] Ran Xu, Subrata Mitra, Jason Rahman, Peter Bai, Bowen Zhou, Greg Bronevetsky, and Saurabh Bagchi. 2018. Pythia: Improving Datacenter Utilization via Precise Contention Prediction for Multiple Co-located Workloads. In *Proceedings of the 19th International Middleware Conference* (Rennes, France) *(Middleware '18)*. ACM, New York, NY, USA, 146–160.

[56] Neeraja J. Yadwadkar, Ganesh Ananthanarayanan, and Randy Katz. 2014. Wrangler: Predictable and Faster Jobs Using Fewer Resources. In *Proceedings of the ACM Symposium on Cloud Computing* (Seattle, WA, USA) *(SOCC '14)*. ACM, New York, NY, USA, Article 26, 14 pages.

[57] Hailong Yang, Alex Breslow, Jason Mars, and Lingjia Tang. 2013. Bubble-flux: Precise Online QoS Management for Increased Utilization in Warehouse Scale Computers. In *Proceedings of the 40th Annual International Symposium on Computer Architecture* (Tel-Aviv, Israel) *(ISCA '13)*. ACM, New York, NY, USA, 607–618.

[58] Tianyi Yu, Qingyuan Liu, Dong Du, Yubin Xia, Binyu Zang, Ziqian Lu, Pingchao Yang, Chenggang Qin, and Haibo Chen. 2020. Characterizing Serverless Platforms with Serverlessbench. In *Proceedings of the 11th ACM Symposium on Cloud Computing* (Virtual Event, USA) *(SoCC '20)*. Association for Computing Machinery, New York, NY, USA, 30–44.

[59] Zhibin Yu, Zhendong Bei, and Xuehai Qian. 2018. Datasize-Aware High Dimensional Configurations Auto-Tuning of In-Memory Cluster Computing. In *Proceedings of the Twenty-Third International Conference on Architectural Support for Programming Languages and Operating Systems* (Williamsburg, VA, USA) *(ASPLOS'18)*. Association for Computing Machinery, New York, NY, USA, 564–577.

[60] F. V. Zacarias, V. Petrucci, R. Nishtala, P. Carpenter, and D. Mossé. 2019. Intelligent Colocation of Workloads for Enhanced Server Efficiency. In *The 31st International Symposium on Computer Architecture and High Performance Computing (SBAC-PAD)*. 120–127.

[61] Haoran Zhang, Adney Cardoza, Peter Baile Chen, Sebastian Angel, and Vincent Liu. 2020. Fault-tolerant and transactional stateful serverless workflows. In *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*. USENIX Association, 1187–1204.

[62] Xiao Zhang, Eric Tune, Robert Hagmann, Rohit Jnagal, Vrigo Gokhale, and John Wilkes. 2013. CPI2: CPU Performance Isolation for Shared Compute Clusters. In *Proceedings of the 8th ACM European Conference on Computer Systems* (Prague, Czech Republic) *(EuroSys '13)*. ACM, New York, NY, USA, 379–391.

[63] Yunqi Zhang, Michael A. Laurenzano, Jason Mars, and Lingjia Tang. 2014. SMiTe: Precise QoS Prediction on Real-System SMT Processors to Improve Utilization in Warehouse Scale Computers. In *Proceedings of the 47th Annual IEEE/ACM International Symposium on Microarchitecture* (Cambridge, United Kingdom) *(MICRO-47)*. IEEE Computer Society, Washington, DC, USA, 406–418.

[64] Jiacheng Zhao, Huimin Cui, Jingling Xue, and Xiaobing Feng. 2016. Predicting Cross-Core Performance Interference on Multicore Processors with Regression Analysis. *IEEE Trans. Parallel Distrib. Syst.* 27, 5 (May 2016), 1443–1456.

[65] Laiping Zhao, Yanan Yang, Kaixuan Zhang, Xiaobo Zhou, Tie Qiu, Keqiu Li, and Yungang Bao. 2020. Rhythm: Component-Distinguishable Workload Deployment in Datacenters. In *Proceedings of the Fifteenth European Conference on Computer Systems* (Heraklion, Greece) *(EuroSys'20)*. Association for Computing Machinery, New York, NY, USA, Article 19, 17 pages.

Appendix: Artifact Description/Artifact Evaluation

SUMMARY OF THE EXPERIMENTS REPORTED

We implement the GSight controller module in Openfaas v0.18.13, and ran it on a Kubernetes v1.18.3 cluster. It periodically queries the functions' deployment status and monitors the function scaling activities. When a new function instance is invoked, the GSight controller calls the scheduling algorithm and performance predictor, and find a placement for it (see the Github of gray/Algorithm). The load generator generates requests towards benchmarks in ServerlessBench, FunctionBench (see gray/LoadGen and gray/LoadGenSimClient). The metric collector is used to collect applications' microarchitecture metrics under different interference scenaios for training the machine learning model (see gray/gsightCollector, gray/models and gray/benchmarks). More details are listed as below.

Benchmark Installation

1. Deployment guide of OpenFaaS for Kubernetes is available here: *https://docs.openfaas.com/deployment/kubernetes/*.

2. Run benchmarks */install.sh* to build & deploy all functions.

Notice: this operation takes some time.

We can also build & deploy functions according to the description below: *https://docs.openfaas.com/cli/build/*

Workload Generator Installation

Guidance is available here: *loadGen/README.md*

Notice: Be sure that OpenFaaS functions and workload generator work well before running thefollowing script. The project LoadGenSimClient cannot work if the LoadGen has not yet been deployed.

Metrics Collector

Compile *gray/gsightCollector/* to generate a runnable jar package. Then, run *java -jar collector.jar func_name interval result_dir_name* to collect metrics of corresponding functions under their solo-run.

Run *models/collector/start.sh* , and it will create a csv file that stores the metrics under co-locating. You can edit *start.sh* to set the QPS of LS workloads, edit *models/collector/get_ml_data.py* to set the amount of data to be collected, and edit *models/collector/runBEPara.py* to configure tasks that co-locate with the LS of social network.

Model Training

The initial training dataset is in *models/algorithm/data/*. Run *models/algorithm/RFR_model_training.py* , and it will create a file named "RFR" to store the RM model, and csv file to store importance of the metrics.

Run *models/algorithm/plots_imports.py* to review the impurity based importance of the 16 metrics, and run *models/algorithm/plots_imports_all.py* to show the importance of 16 metrics under all combinations of workload and server.

Scheduler

It's quite hard to show the whole system, so we provide an example of binary-search scheduling algorithm instead. The example uses randomly generated states of servers and workloads. And it invokes the actual RFR model trained above for checking SLA violation.

Compile and run *scheduler/src/util/test/GsightScheduler.java* to review the example process of scheduling.

Plotting Results

(1) Figure 3: *Figure/matlab/motivation/sn/sn*$_l ci.m$
(2) Figure 4: *Figure/matlab/motivation/timediff/TimeDiffdelay.m*
(3) Figure 5(a): *Figure/matlab/motivation/sn/sn_il_1_new.m*
(4) Figure 5(b): *Figure/matlab/motivation/sn/sn_il_6_new.m*
(5) Figure 6(a): *Figure/python/violin*$_c ode/IPC/violin.py$
(6) Figure 6(b): *Figure/python/violin*$_c ode/Lat/violin.py$
(7) Figure 6(c): *Figure/matlab/motivation/GsightLatency.m*
(8) Figure 8(a): *Figure/matlab/evaluation/IPCLatency/IPCLatency.m*
(9) Figure 8(b): *Figure/matlab/evaluation/IPCLatency/IPCLatencyEC.m*
(10) Figure 9: *Figure/matlab/add/evaluation/FeatureImportance.m*
(11) Figure 10(a): *Figure/matlab/evaluation/xiangti2_before/data90.m*
(12) Figure 10(b): *Figure/matlab/evaluation/xiangti2/data90LatencyIRFRCompare.m*
(13) Figure 11(a): *Figure/matlab/errorProcess/Serverful.m*
(14) Figure 11(b): *Figure/matlab/errorProcess/Serverless.m*
(15) Figure 11(c): *Figure/matlab/evaluation/xiangti2/data90multi.m*
(16) Figure 12(a): *Figure/matlab/evaluation/cdf/DensityCDF.m*
(17) Figure 12(b): *Figure/matlab/evaluation/cdf/CPUCDF.m*
(18) Figure 12(c): *Figure/matlab/evaluation/cdf/MemCDF.m*
(19) Figure 13(a): *Figure/matlab/evaluation/inferenceSLA/SNSLA.m*
(20) Figure 13(b): *Figure/matlab/evaluation/inferenceSLA/TPCWSLA.m*
(21) Figure 14(a): *Figure/matlab/Revenue/PlotPie.m*
(22) Figure 14(b): *Figure/matlab/Revenue/PlotLine.m*
(23) Figure 15(a): *Figure/matlab/add/evaluation/PlotPie.m*
(24) Figure 15(b): *Figure/matlab/add/evaluation/TimeDiffdelay.m*

Author-Created or Modified Artifacts:

```
Persistent ID:
↪  https://zenodo.org/badge/latestdoi/355110307
Artifact name: Source Code for Gray Interference
```

BASELINE EXPERIMENTAL SETUP, AND MODIFICATIONS MADE FOR THE PAPER

Relevant hardware details: 8 nodes of cluster, each server has 256 GB RAM, 40 cores Intel Xeon E7-4820v4 CPU and 960GB SSD capacity

Operating systems and versions: Ubuntu 16.04 running Linux kernel 4.13.0

Compilers and versions: JDK1.8.0, Python3.7, Go1.10.3

Applications and versions: OpenFaaS v0.18.13, Kubernetes v1.18.3

Key algorithms: Random Forest