FnSched: An Efficient Scheduler for Serverless Functions

Amoghvarsha Suresh, Anshul Gandhi
PACE Lab, Stony Brook University
{amsuresh,anshul}@cs.stonybrook.edu

Abstract

An imminent challenge in the serverless computing landscape is the escalating cost of infrastructure needed to handle the growing traffic at scale. This work presents FnSched, a function-level scheduler designed to minimize provider resource costs while meeting customer performance requirements. FnSched works by carefully regulating the resource usage of colocated functions on each invoker, and autoscaling capacity by concentrating load on few invokers in response to varying traffic. We implement a prototype of FnSched and show that, compared to existing baselines, FnSched significantly improves resource efficiency, by as much as 36%–55%, while providing acceptable application latency.

1 Introduction

Serverless computing is an emerging paradigm for running user-specified functions on provider resources with virtually unlimited scalability [3]. In serverless computing, the user is responsible for writing the code and packaging it, and the cloud provider is responsible for provisioning and maintaining the infrastructure, including host servers, needed to execute the user code/program [10]. Serverless computing can also include frameworks that cater to specific application requirements such as BaaS (Backend as a Service) offerings [3]. The pay-per-use pricing for functions, currently at about 20 cents per million invocations (per AWS Lambda [3]), makes serverless a very lucrative choice for end-users. While the set of applications supported by serverless computing is still evolving, there is consensus in the computing community that several classes of applications, including MapReduce-based frameworks, will eventually run seamlessly on serverless platforms [10].

An imminent challenge that we envision in this computing landscape is the escalating cost of infrastructure needed to handle the growing serverless traffic. While the more generic objective of achieving high utilization in serverless environments has been alluded to by recent studies, including the position paper by RISELab at UC Berkeley [10], the specific problem we consider in this paper is centered around scheduling: *how should user functions be scheduled on bare-metal servers to minimize the provider's expenses at scale while providing acceptable latencies?* Note that addressing this challenge requires both (i) efficient placement of the incoming workload to minimize the provider's capital expenses, and (ii) dynamic autoscaling of the serverless platform to minimize the provider's operating expenses.

Existing solution, such as schedulers designed for VM placement or web load balancers, are not well suited for serverless scheduling. The former requires specification of the resource request (e.g., number of cores), which is not an input that a serverless user needs to provide. The latter assumes that any server can execute an incoming request, whereas in serverless computing, specific servers that are already "warm" (have an active container that can serve the incoming function) will be preferred. While container schedulers, such as Borg, may appear to be well suited for serverless workloads, they are not necessarily designed for short-lived functions, and can have task placement latencies as high as 25s [17]; by contrast, serverless functions typically have latencies ranging from milliseconds to few seconds [10, 13]. Kubernetes [10], another popular solution for scheduling containers, requires resource provisioning policy parameters to be specified by the customer, thus adding more burden on the user.

We present FnSched, a function scheduler designed to minimize provider expenses while providing acceptable request latencies. FnSched takes into account the resource consumption and lifetime patterns of serverless functions by classifying them into different categories. FnSched then scalably determines function placement based on the inferred class of the incoming function. To provide acceptable latencies, FnSched mitigates the resource contention between colocated functions by dynamically regulating their cpu-shares at runtime.

Beyond a single host, FnSched also enables autoscaling by elastically adding and removing hosts as needed, based on the incoming workload demand. FnSched adds an additional host when the performance of in-service functions degrades beyond a certain threshold. To remove hosts, FnSched greedily concentrates load on few hosts, allowing unneeded hosts to idle and eventually be turned off.

We implement a prototype of FnSched on Apache OpenWhisk [20] and evaluate its performance on a 10-VM serverless cluster. FnSched is primarily implemented on the Controller component of OpenWhisk, with less than 1,500 lines of code. Our experimental results show that FnSched can provide acceptable latencies across a diverse set of functions, unlike the existing baseline schedulers in OpenWhisk and Linux. Our multi-host evaluation, using time-varying (trace-driven) traffic, highlights the efficiency of FnSched; compared to existing scheduling policies, FnSched handles the incoming traffic with 36%–55% fewer hosts.

2 Background and Motivation

In this section we provide an overview of serverless computing and highlight the challenges in scheduling serverless applications.

Serverless computing. In a serverless computing platform, the user writes a cloud function in a high-level language and create the trigger (events from the back-end, http end-points) [3] to run the function. The serverless provider is then responsible for the infrastructure that will execute the user function every time the function is triggered. In particular, the provider is responsible for the resource management – instance selection, scaling, deployment, fault tolerance, etc. [10]. Typically the serverless platform uses containers or other sandboxing approaches to host the functions [18]. In

WOSC '19, December 9–13, 2019, Davis, CA, USA
2019. ACM ISBN 978-1-4503-7038-7/19/12…$15.00
https://doi.org/10.1145/3366623.3368136

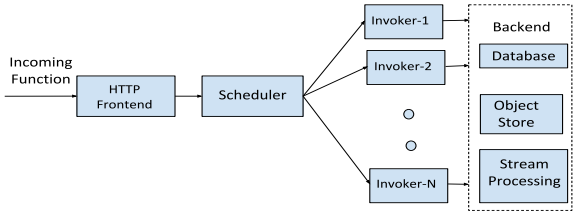


Figure 1: Illustration of a typical serverless platform.

this paper, we assume that the serverless platforms use containers, similar to Google Cloud Run [19] and Apache OpenWhisk [20].

Figure 1 illustrates a typical serverless platform environment. The incoming user function is sent to the scheduler via an HTTP front-end. The scheduler then chooses an invoker on which the function can run. Depending on the application, a function may use additional back-end services, such as a database, object store, etc. In this work, we use the terms invoker, node, and host interchangeably, and all of them refer to the server or VM which hosts the container of a serverless function. Likewise, we use the terms application, function, and workload interchangeably.

Scheduling user functions in serverless environments. In the serverless environment, the provider must decide how to schedule incoming user functions, that is, which invoker should handle the incoming user function. One of the challenges of scheduling in serverless platforms is the diversity in the set of user applications. The advantages of serverless platforms – no explicit provisioning, automatic and virtually unlimited scaled, and being billed on usage – have attracted interest from different application communities, including high-performance computing [16]. This diversity necessitates application-specific scheduling of functions as different applications have vastly different resource consumption patterns, such as short-lived applications, applications with intermittent activity, and embarrassingly parallel services [2, 9, 10].

Another challenge in scheduling serverless functions is the use of containers to host applications. While containers help in packaging the application and ease the function invocation process, they do have an undesirable performance impact. When processing the function of a new application allocated to a host, the following steps are involved (assuming the container of the application is not already present on the host): (i) launching a new container, (ii) setting up the runtime environments, and (iii) application-specific initialization; the latency for these steps is collectively termed as cold start latency [10, 18]. In our experiments, the cold start latency typically ranges from 3 to 6 seconds; by contrast, function execution can take as little as 0.5 seconds. Clearly, cold start of the container is undesirable, as also noted in prior works [7]. While recent advances, such as AWS Firecracker [1], reduce the latency of launching a new container (or microVM), the application and runtime initialization latencies still remain, and can dwarf the former [10].

Scheduling challenges also extend to multi-host scenarios where the incoming functions must be scheduled across multiple invokers. Popular load balancing solutions, such as Round Robin, are designed to spread the load among available hosts, and so naturally tend to use all available hosts (regardless of whether they are all needed or not), resulting in potential under-utilization. Further, in serverless environments, hosts that already have an application's container running on them are preferred over other hosts to avoid the cold start latency; thus, a scheduler must take into account the contents of each invoker while scheduling incoming functions. Finally, a scheduler must ensure that sufficient capacity is available at all times to handle bursts of function requests, given that serverless platforms tout their ability to handle any scale of requests [3].

3 Design of FnSched

To address the challenges outlined above, we begin our investigation with the following two questions: (i) how to efficiently use a single invoker without significantly impacting function latency, and (ii) how to autoscale the number of invokers in response to change in workload demand? In addressing these questions, we consider a serverless environment as illustrated in Figure 1 with a tier of homogeneous invoker nodes. We consider multiple applications, possibly belonging to different users, that issue function requests; thus, functions can be heterogeneous in nature. Each invoker hosts application-specific containers, that in turn serve incoming functions for their application. The scheduler decides which invoker (and container) handles the incoming function, and also decides on the containers to be instantiated on each invoker and the number of invokers to be used.

In serverless platforms, the user expects a certain level of acceptable service from the provider, in terms of the latency of executing the user function. We capture this requirement as a Service Level Objective (SLO) which dictates the allowable latency degradation for a user function, $latencyThd$. Specifically, if the latency of function execution when run in isolation on a dedicated host is $isoLatency$, then the SLO dictates that the latency in the serverless environment should be no more than $(isoLatency \cdot latencyThd)$.

3.1 Scheduling for a Single Invoker

We start with the case of a single invoker where the goal of FnSched is to manage the containers and their cpu-shares to ensure that the SLO is met for all incoming user functions. The key challenge to address in the single invoker case is mitigating resource contention among different application containers in an effort to maximize the number of functions that can be simultaneously served at the invoker without violating the SLOs. In our experiments, the CPU is often under contention, so we focus on mitigating CPU contention. To regulate CPU usage of containers, FnSched leverages *cpu-shares* [4], which specifies the relative share of CPU time available to the container. Note that cpu-shares is a soft limit enforced only when CPU cycles are constrained.

In general, the CPU requirements of a function depend on the underlying application. For example, an embarrassingly parallel application will require many more CPU cycles compared to a short-lived script. Consequently, a long-running application may be able to tolerate a few milliseconds of CPU contention without much impact on latency, unlike a short-lived, cpu-intensive application. Thus, FnSched takes into account the nature of the application when regulating its cpu-shares. We classify serverless applications into two broad categories – Edge Triggered (ET), and Massively Parallel (MP) – based on their runtime and resource usage. ET refers to applications which are typically short-lived and/or triggered based on events, e.g., streaming analytics and IoT back-end [3]. MP applications are resource intensive and are typically embarassingly parallel, e.g., data mining [5], MapReduce [14], etc.

```
numUpdates+=1;
latencyRatio = latency/isoLatency;
if latencyRatio > updateLatencyThd then
    if numUpdates > numUpdatesThd then
        if curShares < perContainerMax then
            toAddShares = cpuSharesStep * numConts;
            if (totShares+toAddShares) < maxCpuShares then
                curShares = curShares + cpuSharesStep ;
                totShares = totShares + toAddShares
            else
                toReduceShares =
                 (toAddShares/numOtherConts);
                rebalanceCpuShares(toReduceShares);
                deltaShares = (maxCpuShares - totShares) /
                 numConts ;
                curShares = curShares + deltaShares ;
                totShares = maxCpuShares
            end
        end
    end
end
```

Algorithm 1: Application-aware cpu-shares regulation.

Regulating cpu-shares. FnSched's application-aware, cpu-shares regulation policy is shown in Algorithm 1. The algorithm is employed at runtime, and updates the allotted cpu-shares of active containers. FnSched monitors the average latency of applications over a moving-window (10 requests, in our case). If the latency for an application starts to approach the SLO ($isoLatnecy \cdot latencyThd$), FnSched increases its cpu-shares. In particular, FnSched increases an application's cpu-shares if its moving-average latency exceeds $isoLatnecy \cdot updateLatencyThd$, where $updateLatencyThd < latencyThd$; the $updateLatencyThd$ acts as an early warning sign to prevent SLO violations. For example, in our experimental evaluation, we set $latencyThd = 1.15$ (meaning no more than 15% degradation) and set $updateLatencyThd = 1.10$.

FnSched increments cpu-shares of all containers of the application in steps of *cpuSharesStep*, to ensure that the cpu allocation of containers is increased gradually. If the increase in cpu-shares will exceed the total cpu-shares of the invoker (1024 cpu-shares per core), FnSched rebalances the shares from the other application containers. After increasing cpu-shares for an application, FnSched waits for *numUpdatesThd* iterations (or requests) before evaluating the application's latency again, to ensure that the newly set cpu-shares value has fully taken effect. We perform a sensitivity analysis of the various algorithm parameters in Section 4.4.

3.2 Multi-Invoker Scheduling

When the load exceeds the capacity of one invoker, FnSched must scale out to add more invokers and maintain acceptable latencies. Likewise, when the load decreases, FnSched must reduce the number of invokers to scale capacity (and operating expenses) with demand, also referred to as autoscaling [8].

To autoscale invoker capacity, FnSched employs a greedy algorithm. The central idea, based on the AutoScale data center power management policy [8], is to index all invokers, say from 1 to n, and try to pack requests as efficiently as possible on lower-numbered invokers; consequently, higher-numbered invokers, if not needed, will be idle and can be turned off after some duration of inactivity. Specifically, an incoming request is sent to the lowest-numbered invoker that can accommodate it.

To check whether an invoker can accommodate an incoming application request, FnSched checks the available memory (if a new container needs to be created), available admission capacity, and the moving-average latency of the application on that invoker, as explained below. The admission capacity is equal to the number of idle cores in the invoker and is used as a proxy for the additional number of requests that can be supported on the invoker. The latency of the application is classified into three operating zones – safe: less than 50% of SLO, warning: between 50–75% of SLO, and unsafe: greater than 75% of SLO. An invoker is considered eligible to run a request in either safe or warning zone. When the latency of an application in a host enters the warning zone, we halve the host's admission capacity. If an invoker enters the unsafe zone, we do not admit any more requests from the application for a brief period of time (2 seconds in our case). After this period, the invoker is moved into the warning zone and the latency moving-window is reset so that the appropriate operating zone can be inferred.

A key advantage of FnSched's greedy algorithm, in the context of serverless scheduling, is that by preferring lower numbered invokers, FnSched tends to reuse previously used invokers, thus avoiding cold starts. To further avoid cold starts and reserve capacity for workload bursts, when we *first* use a given invoker for an application, say invoker with index n, we proactively spawn a container of this application on a new invoker, indexed $(n + 1)$, thereby scaling out capacity. We periodically send a heartbeat request to this proactively spawned container to keep it warm.

4 Evaluation

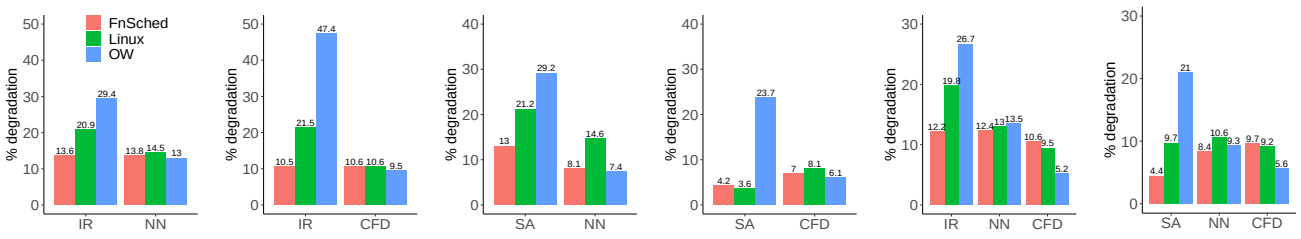
We now present our evaluation of FnSched. We start by detailing our prototype implementation of FnSched, and then describe our experimental setup and evaluation methodology. Finally, we present our experimental evaluation results for FnSched under the single-invoker and multi-invoker settings.

4.1 Implementation of FnSched

We implement FnSched on top of Apache OpenWhisk [20], an open-source serverless cloud platform. OpenWhisk has a REST interface to accept requests and provide a response to them. The Controller component is responsible for processing the HTTP method and executing the function on the invoker, which in turn creates the Docker container. We implement our cpu-shares regulation algorithm (Algorithm 1) at each invoker, and implement our greedy autoscaling algorithm in the Controller. In total, we add about 1,500 lines of code in Scala to implement FnSched.

4.2 Experimental Setup

We host OpenWhisk on a cluster with 10 VMs, each with 4 cores and 8GB of memory. The HTTP front-end runs on a dedicated VM and the remaining OpenWhisk components, including the Controller, run on a different VM. The remaining 8 VMs serve as invokers. The applications hosted on these invokers are deployed on Docker containers, with the inactivity timeout for the containers set to 1 minute. To support the applications we employ (see below), we also



(a) IR colocated with NN. (b) IR colocated with CFD. (c) SA colocated with NN. (d) SA colocated with CFD. (e) IR with NN and CFD. (f) SA with NN and CFD.

Figure 2: Latency degradation when ET application is colocated with MP applications on a single invoker.

use dedicated hosts for back-end services – distributed filesystem (3 servers), database (3 servers), and Apache Kafka (1 server).

Applications. We use two edge triggered (ET) and two massively parallel (MP) applications in our experiments. We implemented all of these applications in Python 3.5.

- *Image Resizing (IR)* accepts an input image, which is read from a distributed file system, resizes it into three different image sizes, and stores them back into the distributed file system.
- *Streaming Analytics (SA)* accepts a batch of messages, processes them, and conditionally writes them to the database.
- *Nearest Neighbor (NN)* is an MP application whose MPI implementation has been ported from Rodinia [5], a GPU benchmark suite. To exchange data, NN uses a distributed file system.
- *Computational Fluid Dynamics (CFD)* is an MP application whose MPI implementation has also been ported from Rodinia [5]. To exchange data, CFD uses a distributed file system.

Both the ET applications are modeled from the Lambda reference architecture examples [2]. The applications use Ceph Distributed filesystem, working as an Object Store (similar to AWS S3), to read and store the data (when applicable). Streaming Analytics uses Apache Kafka to read the streaming data and Redis database to store the data. All applications use 256 MB of reserved memory, expect CFD, which uses 512 MB of memory.

4.3 Evaluation Methodology

Metrics. We use two key metrics to evaluate the performance of different scheduling policies in our experiments.

(1) *Latency degradation*: this is the average percentage latency degradation of an application compared to its standalone latency, *isoLatency*. We set the SLO to be 15% over the *isoLatency*; thus, $latencyThd = 1.15$ (see Section 3). Thus, at most 15% degradation is allowed, over an application's standalone latency.
(2) *Invokers used*: this is the time-average number of invokers used by a scheduler over the duration of the experiment, and acts as a proxy for the provider's expenses. An invoker is considered to be "in use" if it has at least one live container. Since all invokers are homogeneous in our setup, invoker count serves as a proxy for the cost metric. This metric is relevant for the multi-invoker experiments.

All reported metrics are averaged across 3 runs of an experiment. To compute the standalone latency, *isoLatency*, of an application we use a dedicated host and launch a warm container of the application and profile its latency in isolation.

Workload and traces. For the applications we employ, the standalone latency, or *isoLatency*, is as follows: 0.67s for IR, 0.73s for SA, 7.2s for NN, and 19.2s for CFD. Unless otherwise mentioned, we use a request rate of 1 req/s for IR, 0.5 req/s for SA, 0.25 req/s for NN, and 0.17 req/s for CFD. Since NN and CFD are massively parallel applications, we issue requests in batches (of 4).

For single invoker evaluation (Section 4.4), the experiments have a warm up phase wherein we spawn a container per core per application, followed by 15 minutes of constant load, as dictated by the above mentioned request rates. For multi-invoker evaluation, we use synthetic and real-world traffic traces to drive the time-varying load, as detailed in Section 4.5. For real-world traffic traces, we employ network traffic traces from WITS [21], suitably scaled to our testbed capacity, as a proxy for serverless traffic.

4.4 Evaluating Single Invoker Scheduling

We start by evaluating FnSched under a single invoker. Recall from Section 3.1 that the key challenge when scheduling in a single invoker is the resource (cpu) contention among functions. For each experiment, we compare FnSched with the following baselines.

(1) *OpenWhisk default*: OpenWhisk sets cpu-shares for each container proportional to its requested memory capacity [20]. Specifically, if m is the memory requested by a container, and M and C are respectively the total memory and cpu-shares capacity of the invoker, the container is provided $m \cdot C/M$ cpu-shares. This policy will maximize the number of containers that can be spawned in a host, since both memory and cpu are being proportionally allocated.
(2) *Linux default*: Linux's cpu-shares policy allocates every container with one CPU core worth of cpu-shares (i.e., 1024 shares).

Sensitivity analysis. FnSched's cpu-shares regulation policy employs Algorithm 1, which has three parameters – *numUpdatesThd*, *cpuSharesStep*, and *maxCpuShares*. To determine appropriate values for these parameters, we conduct a sensitivity analysis using FnSched by colocating different ET and MP applications together on one core. We experiment with the following ranges for each parameter: *numUpdatesThd* – 1, 3 and 5; *cpuSharesStep* – 64 and 128; *maxCpuShares* – 256, 512, 768, and 1024 (a full CPU core).

The *maxCpuShares* parameter determines the ceiling of the cpu-shares for an application. Based on our sensitivity analysis experiments, we find that a *maxCpuShares* value of 768 for ET and 256 for MP works well to accommodate containers of MP and ET on a single core. Since *maxCpuShares* is high for ET and low for MP, we find that *cpuSharesStep* of 128 and 64 work well for ET and MP, respectively. The *numUpdatesThd* parameter determines the minimum number of iterations required before we consider updating cpu-shares again. For *numUpdatesThd*, we find that values of 5 and 3

work well for ET and MP, respectively. We use the above parameter values when employing FnSched for the rest of our experiments.

Colocating two applications. Figures 2(a) and 2(b) show our results when colocating IR with NN and CFD, respectively. In both cases, FnSched satisfies the 15% latency degradation SLO. By contrast, the SLO is violated for IR under both comparison baselines; under the OpenWhisk default scheduler, the latency degradation can be as high as 47.4%.

Under FnSched, both IR and NN quickly stabilize at their respective maximum cpu-shares. Subsequently, when there is contention, IR gets higher preference (since it has a *maxCpuShares* value of 768 compared to 256 for NN) and so its latency degradation is minimized. Under the Linux default baseline, IR and the MP application are allocated the same cpu-shares value, making the short-lived IR application vulnerable to resource pressure from the longer-running, cpu-intensive MP applications. Under the OpenWhisk baseline, which allocates cpu-shares proportional to the memory requirements of the application, IR gets equal preference when colocated with NN, but gets lower preference when colocated with CFD since CFD uses 512MB of memory compared to the 256MB used by IR. Consequently, IR is unable to avoid the cpu contention with the MP application, resulting in severe latency degradation.

The result of colocating SA with NN and CFD is shown in Figure 2(c) and 2(d). Results are similar, with FnSched complying with the latency SLO in all cases, unlike the other baselines.

Colocating three applications. We also experiment with colocating the ET applications with more than one MP application. Of the 4 cores on the invoker, we colocate the ET application with NN on 2 of them and with CFD on the other 2. Figures 2(e) and 2(f) shows the resulting latency degradation when using IR and SA as the ET applications. FnSched consistently meets the SLO requirements of all applications in all cases. By contrast, the latency degradation of the ET application is much higher under the Linux and OpenWhisk baselines, with the degradation violating the SLO in most cases. As before, by prioritizing the ET application, FnSched protects the performance of the short-lived functions. While the MP applications, which are not prioritized, can have higher latency under FnSched as compared to the other baselines, the latency degradation is still within the SLO requirements.

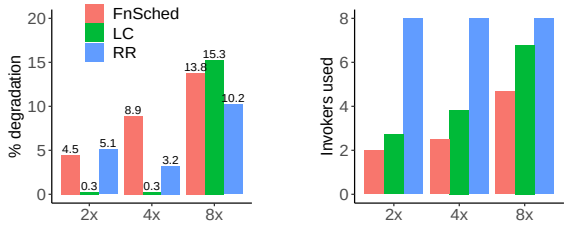
We also experimented with other potential colocation combinations, such as ET with ET, and ET with ET and MP. In these cases, we found that all policies perform similarly.

4.5 Evaluating Multi-Invoker Scheduling

We now evaluate the performance of FnSched in multi-invoker scenarios. As mentioned in Section 3, the objective for multi-invoker scheduling is to autoscale capacity, that is, minimize the average number of invokers employed while ensuring acceptable application latencies. For evaluating FnSched in such scenarios, we compare its performance with the following baseline schedulers:

(1) *Round Robin* spreads the load evenly among invokers by sending successive requests to different invokers in a cyclic manner.
(2) *Least Connections* sends the incoming request to the least loaded (fewest in-flight requests) invoker.

For all baselines, and FnSched, we employ the superior cpu-shares policy identified in Section 4.4, namely, FnSched's Algorithm 1, with the parameters identified by our sensitivity analysis.



(a) Latency degradation (%). (b) Average number of invokers.

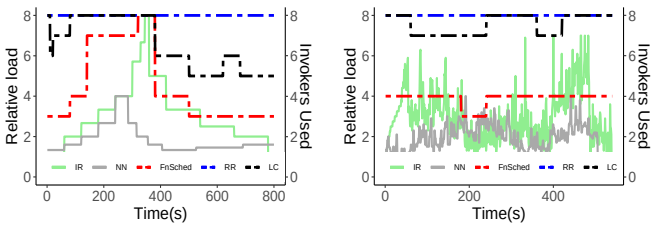
Figure 3: Performance of different multi-invoker schedulers under different load levels of IR.

Since our focus in the multi-invoker evaluation is on autoscaling abilities, we discuss our results under the following experimental conditions. First, we consider the case where there is only one application, IR; then, we consider the case where requests for two different applications, IR and NN, are being received. For both cases, we experiment with time-varying load.

Scaling with a single application. To drive the load for our experiments, we employ three synthetic traces with the load gradually building up from 0 and going up to a maximum of 2×, 4×, and 8× of the single invoker load, respectively. For example, under 8× load, the peak request rate for IR is $8 \times 1 = 8$ req/s (see Section 4.3). The gradual build up happens over 1–3 minutes duration, and then the peak load is sustained for 5 minutes.

Our results for single application scaling are shown in Figure 3, where the x-axis represents the peak load of the synthetic trace (relative to the single invoker load). We see from Figure 3(a) that FnSched meets the SLO in all cases, similar to the other baselines. Figure 3(b) highlights the crucial resource savings enabled by FnSched as it employs fewer invokers, on average, compared to Round Robin (RR) and Least Connection (LC). Since RR cycles incoming requests among invokers, it keeps all invokers busy, that is, each invoker has at least 1 live container on it; thus, RR uses 8 invokers on average. LC tends to use all invokers at peak load, but uses fewer invokers during low load, resulting in slightly less than 8 invokers in use, on average. By contrast, by packing requests on few containers, FnSched significantly reduces the number of invokers used, while still meeting the SLO requirements; we find that the proactively-spawned invoker under FnSched helps to meet the latency SLO when packing requests. Across all three traces, FnSched reduces the average number of invokers employed by about 62% and 31%, respectively, compared to RR and LC.

Scaling with multiple applications. We now consider the case of multi-invoker scheduling with multiple applications. We start with synthetic traces, shown by the solid lines (left y-axis) in Figure 4(a), with requests from two applications – IR and NN. Here, the relative load denotes the multiple of request rate compared to the single invoker experiments. The number of invokers used by different policies is shown on the right y-axis. All policies meet the SLO, but the average number of invokers employed varies significantly. FnSched closely follows the load, resulting in more responsive scaling of invokers. By contrast, RR and LC generously employ invokers, resulting in higher usage. Over the length of the experiment, FnSched reduces the average number of invokers employed by about 46% and 32%, respectively, compared to RR and LC.



(a) Synthetic trace. (b) WITS trace-1.

Figure 4: Load and the number of hosts used in colocation of multiple applications scaling experiments.

We now consider experiments where the load is driven by real-world (WITS [21]) traffic traces. We employ three different sets of traces to drive the load for IR and NN; one of these scenarios is shown in Figure 4(b) with the load for IR and NN illustrated by the solid lines. For all traces, while all policies meet the SLO, the number of invokers employed varies across the policies. As shown in Figure 4(b), FnSched typically uses about 40%–50% fewer invokers compared to RR and LC. On average, across all traces, FnSched employs about 55% and 36% fewer invokers compared to RR and LC, respectively.

5 Prior Work

Serverless platform characterization: A performance study was conducted by Wang et al. [18] on the serverless offerings of AWS, Azure and Google. The study finds that AWS uses bin-packing-like strategy to maximize VM memory utilization and that severe contention between functions can arise in AWS and Azure. A similar characterization was conducted by Lloyd et al. [11], revealing that container initialization burdens serverless computing platforms. The authors found that extra infrastructure is provisioned to compensate for initialization overhead of cold service requests, motivating the need for resource efficiency in serverless environments.

Scheduling: While scheduling approaches have been studied for decades, to the best of our knowledge, we have not found any prior works specific to serverless scheduling for cloud providers. Rausch et al. [15] propose a serverless platform for operating edge AI applications in edge clouds. The authors envision a hybrid edge/cloud platform, and show that the task placement latency of the Kubernetes default scheduler is unacceptable when the cluster has a large number of nodes.

Existing request-level schedulers, such as Sparrow [12], typically consider requests that have similar execution patterns. By contrast, serverless requests can take anywhere from a few milliseconds (ET applications) to multiple seconds or even minutes (MP applications) to complete. As such, FnSched also regulates the resource usage of the requests at a fine-grained level (via cpu-shares) to mitigate contention. For cluster-level schedulers, such as Quasar [6], the focus is typically on scheduling (VM-hosted) applications; by contrast, in our work, the focus is on scheduling individual requests across application containers.

Kubernetes, a technology that simplifies management of serverful computing [10], is a reasonable option to schedule containers. However, while Kubernetes can schedule serverless containers (currently available as Knative service), it requires the scaling policy parameters to be specified by the user. By contrast, FnSched monitors resource usage and scales resources (invokers) automatically to meet SLO requirements, without introducing too much complexity in the scheduling design. We believe that FnSched can be integrated with Knative, which we plan on pursuing as part of future work.

6 Conclusion

This paper presents our ongoing efforts on serverless scheduling. Our experience with FnSched suggests that the design of serverless schedulers should take into account the diversity in resource consumption and lifetime of functions. Further, to achieve efficiency at scale, load should be "unbalanced", contrary to existing load balancing policies. With this paper, we also aim to highlight the scheduling problem, and its challenges, in the context of serverless.

One of the limitations in the current design of FnSched is the assumption that function execution times are not variable, and can thus be estimated via profiling for determining cpu-shares. In public clouds, this assumption may not hold true; we are working on enhancing FnSched to address this limitation. A second limitation that we plan on addressing is the (currently) manual classification of functions into ET and MP categories. Finally, while this work assumes homogeneous invokers, we plan to extend FnSched to heterogeneous invokers, such as those in the public cloud [18], as part of future work.

Acknowledgment

This work was supported by NSF grants 1617046, 1717588, & 1750109.

References

[1] Amazon Web Services 2019. *Firecracker.* Amazon Web Services. https://firecracker-microvm.github.io/
[2] Amazon Web Services 2019 (accessed August 25, 2019). *Reference Architectures.* Amazon Web Services. https://aws.amazon.com/lambda/resources/#Reference_Architectures
[3] Amazon Web Services 2019 (accessed August 25, 2019). *Serverless Computing.* Amazon Web Services. https://aws.amazon.com/lambda/
[4] Docker 2019 (accessed August 25, 2019). *Specify container resource.* Docker. https://docs.docker.com/config/containers/resource_constraints/
[5] Che et al. 2009. Rodinia: A Benchmark Suite for Heterogeneous Computing. In *IISWC '09.*
[6] Delimitrou et al. 2014. Quasar: Resource-efficient and QoS-aware Cluster Management. In *ASPLOS '14.*
[7] Fouladi et al. 2017. Encoding, Fast and Slow: Low-Latency Video Processing Using Thousands of Tiny Threads. In *NSDI 17.*
[8] Gandhi et al. 2012. AutoScale: Dynamic, Robust Capacity Management for Multi-Tier Data Centers. *ACM Trans. Comput. Syst.* (2012).
[9] Gan et al. 2019. An Open-Source Benchmark Suite for Microservices and Their Hardware-Software Implications for Cloud & Edge Systems. In *ASPLOS '19.*
[10] Jonas et al. 2019. *Cloud Programming Simplified: A Berkeley View on Serverless Computing.* Technical Report. EECS Department, UC Berkeley.
[11] Lloyd et al. 2018. Serverless Computing: An Investigation of Factors Influencing Microservice Performance. In *IC2E 2018.*
[12] Ousterhout et al. 2013. Sparrow:Distributed,Low Latency Scheduling. In *SOSP'13.*
[13] Oakes et al. 2018. SOCK: Rapid Task Provisioning with Serverless-Optimized Containers. In *ATC'18.*
[14] Pu et al. 2019. Shuffling, Fast and Slow: Scalable Analytics on Serverless Infrastructure. In *NSDI'19.*
[15] Rausch et al. 2019. Towards a Serverless Platform for Edge AI. In *2nd USENIX Workshop on Hot Topics in Edge Computing (HotEdge 19).*
[16] Shankar et al. 2018. numpywren: serverless linear algebra. *CoRR* (2018).
[17] Verma et al. 2015. Large-scale Cluster Management at Google with Borg. In *EuroSys '15.*
[18] Wang et al. 2018. Peeking Behind the Curtains of Serverless Platforms. In *2018 USENIX Annual Technical Conference (USENIX ATC 18).*
[19] Google Cloud Platform. 2019. Cloud Run. (2019). https://cloud.google.com/run/
[20] The Apache Software Foundation 2019. *apache/openWhisk.* The Apache Software Foundation. https://github.com/apache/openwhisk
[21] WAND Network Research Group. 2019. WITS: Waikato Internet Traffic Storage. http://www.wand.net.nz/wits/index.php. (2019).