Inverse Response Time Ratio Scheduler: Optimizing Throughput and Response Time for Serverless Computing

Mina Morcos
Computer Science
Boston University
minawm@bu.edu

Ibrahim Matta
Computer Science
Boston University
matta@bu.edu

***Abstract*—We explore the problem of scheduling in a distributed multi-cloud serverless scenario, in the case where requests to function instances contend over the same resources. For this case, we present an efficient scheduling algorithm that leverages function profiling to detect resource contention, and improves the response time of requests, as well as the overall completion time of all requests. We compare our work to other schedulers such as a simple random scheduler, a simple round-robin scheduler, and schedulers that either load balance requests for each function across clouds, choose the cloud with the best profile, or select the cloud with the most available resources. Besides simulations, we have created a simple experiment of our scheduler running against two OpenWhisk serverless instances over the FABRIC testbed. We show that our inverse response time ratio scheduling algorithm can yield an improvement in average response time of around 32% over the best of the other schedulers when the execution time of a function on a given cloud is twice as much as on another. We also show that the improvement increases as the dispersion of the execution time across the distributed environment increases.**

***Index Terms*—Scheduling; Serverless; Cloud Computing; Performance Analysis; Testbed Evaluation**

I. Introduction

Cloud computing offers developers many types of services to deploy their software applications. In Infrastructure-as-a-Service (IaaS), the developer typically reserves virtual machines (VMs) in the cloud. The cloud computing platform would be running an operating system, typically called a hypervisor, that manages the resources. The developer asks for an allocation of resources from the hypervisor, and gets a virtual machine that leverages them to run. However, the developer would then need to spend time to configure the VMs and deploy the application.

Serverless computing abstracts part of this process away. In serverless, the developer is asked to submit code in the form of a function, or an interconnected set of functions. The cloud provider manages resource provisioning and configuration, which is typically done through allocating containers or lightweight VMs. The function code can then be invoked multiple times, sometimes with different input parameters.

The resources of a serverless cloud provider may be internally distributed across multiple locations (regions). When invoking a function, the provider would need to decide which location to pick in order to run the request. Moreover, requests to a certain function type may perform better in one location over another location. This may be caused by multiple factors, such as sharing a networking path with other function requests, or needing a graphics processing unit (GPU) for faster graphical processing, or needing a non-volatile memory express (NVMe) for faster disk I/O (Input/Output) operations.

We have explored state-of-the-art scheduling policies for distributing serverless requests over a distributed multi-location (or multi-cloud or multi-region) system. Some of these policies include *fair sharing*, in which the load is distributed equally across all locations; *most available resources* policy, which targets the location with the most free resources; and *best profile* policy, which picks the location with the function's best performance.

In this work, we present a scheduling policy that outperforms the aforementioned scheduling policies, as well as some other scheduling policies we explored. We have verified the results theoretically and experimentally. Our policy leverages performance profiling to determine how instances of functions perform across the various locations. It favors locations where the function performs well, and discourages locations where the function performs poorly by *a certain ratio* that optimizes the throughput of the overall system. Our results show that our inverse response time ratio scheduler outperforms other schedulers when requests to the same function contend over the same resources while requests to different function types are isolated from each other. For example, requests of one function type may contend over a specific network link to access a certain server, or over a specific group of files on disk, or over the GPU, etc.

Code for work presented in this paper is available online on Github [1].

The paper is organized as follows: Section II reviews related work. Section III motivates our proposed scheduler, which is illustrated in Section III-G. Section IV presents an analysis of how our scheduler optimizes system throughput, and Section V describes the operation of our scheduling algorithm. We present results from simulation in Section VI-A and from

experiments on the FABRIC testbed in Section VI-B. We conclude with future work in Section VII.

II. Related Work

In this section, we describe work on scheduling algorithms for serverless functions, especially in distributed scenarios. We also present other schedulers studied in non-serverless settings and adapt their techniques to our serverless setting.

Recently, Zhang et al. [2] (2021), Rausch et al. [3] (2021), Baresi et al. [4] (2022), Baarzi et al. [5] (2021), and Gadepalli et al. [6] (2019) proposed optimizing system throughput by directing incoming requests to the location where the function has experienced the least execution time or response time.[1] We show that this greedy best-location approach may seem intuitive, but it can experience contention and oscillation issues.

Moreover, fairly recently, Kaffes et al. [7] (2019) and Stein [8] (2018) take the opposite approach to the previous approach. They work on fair sharing or load balancing, which means distributing the load over available locations equally. They aim to maximize throughput, as well as minimize queuing delay.

Mampage et al. [9] (2021) present a policy that schedules requests to functions to the location that has the most amount of free resources. This approach does not predict where a function performs best, and can perform worse than the aforementioned best-location scheduling policies. On the other hand, our proposed scheduler identifies (predicts) the best location by directly measuring function's response time.

Sun et al. (HPSO) [10] (2014), Soltani et al. [11] (2018), and Zhao et al. [12] (2021) all measure average function's response time, and aim to optimize performance through predictive scheduling policies, migration and detecting function interference. Their prediction policy (HPSO) basically predictively schedules requests to the location where free resources will become available, similar to Mampage et al. [9]. It does not predict where the function will perform best. Also, their approach is computationally expensive. We have experimentally adapted their work in our simulations, and our results show that our approach outperforms HPSO. And as for migration, the process is not trivial, since the function needs to be halted, moved over the network, and resumed, all while making sure its state and functionality remain intact. This is also time consuming.

Liang et al. [13] (2020) and Delimitrou et al. [14] (2013) attempt to optimize the response time by tracking parameters such as average response time, available resources, hardware configuration, and function profiles. Liang et al. use two deep convolutional neural networks (Advantage Actor Critic), and Delimitrou et al. employ performance profiling. Both studies optimize performance through individual nodes in the distributed environment, as opposed to working on the aggregate distributed environment.

[1]Response (service) time of a function request consists of its time waiting to start its execution (queuing time) plus the execution time (run time) of the request.

Edgenet [15] (2021), Delay Scheduling [16] (2010), and Quincy [17] (2009) aim at optimizing aspects of the system such as response time, throughput, round-trip time, and fairness through improving data locality. They move function requests closer to the data needed. This is achieved via methods such as IP geolocation via labeling, or imposing delay until good data locality is achieved. IP geolocation requires parsing the function code, labels, and an accurate IP-to-location database. Imposing delay would in some cases introduce improved data locality, and therefore performs similar to aforementioned policies that schedule requests to the best location.

Soltani et al. [18] (2018) and Quenum et al. [19] (2021) simply mention the use of one broker serving multiple serverless cloud providers, without providing a scheduler for the system, which we provide in this paper.

De Palma et al. [20] (2022) and Samea et al. [21] (2019) present languages for defining scheduling policies, with simple schedulers such as random, which we trivially test against.

Table I classifies and compares our response time scheduler to related work along several dimensions.

	Input Parameters	Distributed Environment	Scheduling Policies	Objective Function(s)	Is Work Conserving
Our Approach	Average Response Time	✓	Leveling Profiled Average Function Response Times	System Throughput	✓
[2], [3], [4]	Average Response Time [2] Locality & Hardware [3] & [4]	✓	Best Location	System Throughput	✗
[5], [6]	SLO Violations	✓	Least SLO Violations	Latency (Run Time) & Predictability [6]	✗
[7], [8]	Resources [7] Arrival Rate & Average Response Time [8]	✓	Load Balancing	Throughput [7] Queuing Delay [8]	✗
[9]	Amount of Free Resources	✓	Location With Most Amount of Free Resources	Deadline (Run Time) & Resource Consumption Cost	✓
[10], [11], [12]	Average Response Time	✓	Prediction [10] Migration [11] Interference [12]	Run Time	([10] ✓) ([11], [12] ✗)
[13], [14]	Average Response Time & Resources [13] Hardware Configuration & Function Profiles [14]	✗	CNN [13] Profiling Based [14]	Run Time	([13] ?) ([14] ✗)
[15], [16], [17]	Data Locality	✓	IP Geolocation & Labels [15] Waiting [15] Graph Solving [17]	Throughput & RTT [15] Run time [16] Fairness [17]	✗

TABLE I
Classification of related work.

III. Background and Motivation

We present a scheduling algorithm for distributed serverless computing scenarios. We leverage dispersion of function performance across locations, as well as resource contention among requests of instances of the same function. Figure 1 compares our inverse response time ratio scheduler with two other scheduling policies, which can be thought of as two ends of a spectrum.

On one end of the spectrum, one can distribute the load in a fair manner across the available nodes. On the other end of the spectrum, one could send the load to the fastest node(s). We elaborate that the work we present outperforms both ends of the spectrum.

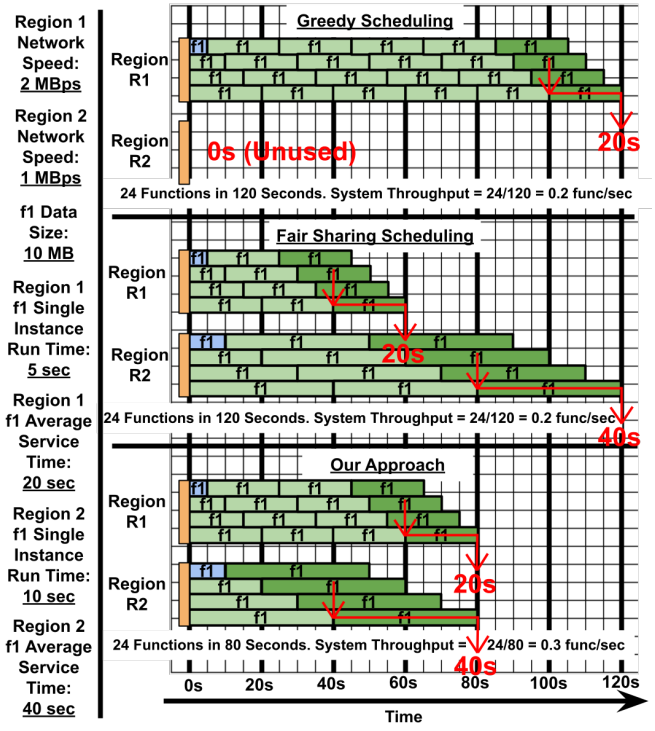

Fig. 1. This example compares our inverse response time ratio scheduler with two other scheduling policies, which can be thought of as two ends of a spectrum. We show how our approach optimizes throughput by scheduling requests with ratios that represent the inverse of the average response (service) time at each location. When profiling, we can keep the measurements of only recent function requests. In this example, we highlight the last four function requests. We assume that the vertical bar in 'orange' next to each location represents the bottleneck resource, and it allows only four function instances (requests) to run concurrently at the location. Assume the bottleneck resource is memory. Also, typically, different functions would be isolated from each other – they might each run in a separate container. We thus assume that requests of different function types compete on separate bottleneck resources, without contending with each other. We measure the average service (response) time by recording the difference between the start and end times of function requests.

We examine two performance metrics: (1) *Average response (service) time* of each function type, which is defined as the average time elapsed to serve the requests of that function type, from the time they enter the scheduler until they are served and depart the system; (2) *Completion time of the experiment*, which is calculated as the time when the experiment finishes (i.e., all requests are served) minus the time the experiment started.

As an example, consider two locations, loc_1 and loc_2. Assume we have one type of function f_1. This function f_1, when run on loc_1, takes 1 second to finish. But when run on loc_2, it takes 2 seconds to finish. Also, assume each of the two locations can only run one instance (request) at a time. Let's say there are six f_1 function invocation requests in the queue.

A. Random

A naïve random scheduler would just pick any location at random for each request.

B. Round Robin

A naïve round-robin scheduler would distribute the requests equally over the two locations. Thus, since there is contention of resources by f_1 requests, loc_1 would finish after 3 seconds as each one of the three requests at loc_1 takes 1 second. But loc_2 would finish serving its three requests after 6 seconds, since each f_1 request takes 2 seconds at loc_2. Thus, the completion time would be 6 seconds.

To calculate the average response time, for each function instance, we need the arrival time, and the finish time. For the arrival times, all requests arrive at time zero in this simple example. For the completion times, at location 1, they finish after 1 second, 2 seconds, and 3 seconds, and at location 2, they terminate after 2 seconds, 4 seconds, and 6 seconds. So, the average response time would be $((1-0)+(2-0)+(3-0)+(2-0)+(4-0)+(6-0))/6$, which is 3 seconds.

C. Load Balance Each Function

Load balancing would do the same and place an equal number of function invocation requests at each location. This is the same behavior as that of round robin, so we get the same completion time (6 seconds) and average response time (3 seconds).

D. HPSO Adapted Scheduler [10] (2014)

This is a scheduler adapted from the High Performance Scheduling Optimizer (HPSO) scheduler. We have put emphasis on the predictive part of the scheduler. The adapted scheduler, for the most part, predicts the earliest time where there will be free resources for the function instance to be placed at each location, and places the instance at the location where there would be free resources at the earliest time. The method should be quite challenging to implement in comparison to the rest. For this simple case, HPSO, based on a brief tracing analysis, is not deterministic due to different ways to breaking ties, and it can place an equal amount of requests at each location in the worst case, so we get the same result, i.e. completion time of 6 seconds, and average response time of 3 seconds. Our method mostly relies on collecting profiles of function service times, as well as collecting the count of function requests running at each location, which should be easier to obtain. Our method should be more efficient. Moreover, HPSO considers the available resources of the system, and not the function profiles.

E. Best Profile

A greedy profiling only scheduler would schedule all the functions on loc_1, and leave loc_2 not utilized, since the execution time is lower on loc_1. The completion time would be 6 seconds, given each one of the six requests takes 1 second on loc_1. The average response time would be $((1-0)+(2-0)+(3-0)+(4-0)+(5-0)+(6-0))/6$ = 3.5 seconds.

F. Most Free Resources

This scheduler, also referred to as Least Busy Resources First (LBRF), would behave the same way as Round Robin. It would work on balancing the load. All requests of f_1 require the same amount of resources. So, we get a completion time of 6 seconds, and an average response time of 3 seconds. This scheduler does not use request-specific metrics to potentially improve performance.

G. Our Inverse Response Time Ratio Algorithm

Our inverse response time ratio algorithm uses the profile of the function that it has collected. It optimizes the completion time, and the average response time of the overall system. We assume that the scheduler has collected a profile of the function, which contains metrics such as the average response time of the function at each location, both recent and non-recent. Consider the following example. Assume the average response time of function f_1 on loc_1 is half that on loc_2. For example, a request for f_1 on loc_1 takes 1 second, and on loc_2 it takes 2 seconds. Our inverse response time ratio scheduler would then schedule the requests according to the inverse ratio of the profiles. Thus, since the ratio is 1:2 on loc_1:loc_2, it would schedule the function requests with a ratio of 2:1 on loc_1:loc_2. Assuming six requests, that would be 4 requests on loc_1 and 2 requests on loc_2. Four functions on loc_1 take 4 seconds to complete, whilst two functions on loc_2 also take 4 seconds to complete. Location 1 finishes after $1+1+1+1$ = 4 seconds, and location 2 finishes after $2+2$ = 4 seconds. Thus, the completion time would be 4 seconds. For the average response time, it would be $((1-0)+(2-0)+(3-0)+(4-0)+(2-0)+(4-0))/6 \approx 2.67$ seconds. Thus, we can see that our inverse response time ratio scheduler algorithm optimizes the completion time, the throughput, and the average response time of the overall system.

IV. Analytical Model

Optimally, completion times of all locations of the distributed system would be equal. Therefore, we start by equating completion times, since that would mean that no location finishes later than the rest.

Denote by C_i the completion time of all requests at location i, n the total number of requests in the queue, $\overline{R}_i$ the average response time at location i, and α_i the ratio of requests directed to location i.

We want:

$$C_1 = C_2 = C_3 = \cdots$$

$$\alpha_1 \times n \times \overline{R}_1 = \alpha_2 \times n \times \overline{R}_2 = \alpha_3 \times n \times \overline{R}_3 = \cdots$$

$$\alpha_1 \times \overline{R}_1 = \alpha_2 \times \overline{R}_2 = \alpha_3 \times \overline{R}_3 = \cdots$$

Solving for the values of α_i, the values of $1/\overline{R}_i$ satisfy the above equality, i.e., all terms will equate to 1.

$$\tfrac{1}{\overline{R}_1} \times \overline{R}_1 = \tfrac{1}{\overline{R}_2} \times \overline{R}_2 = \tfrac{1}{\overline{R}_3} \times \overline{R}_3 = \cdots$$

Thus, our scheduler would direct requests according to the ratios of the inverse average response times at the different locations:

$$\tfrac{1}{\overline{R}_1} : \tfrac{1}{\overline{R}_2} : \tfrac{1}{\overline{R}_3} : \cdots$$

V. Our Inverse Response Time Ratio Scheduler

Algorithm 1: Inverse Response Time Ratio Algorithm

```
1  new_functions = get_new_functions_to_schedule()
2  function_index = 0
3  while function_index < len(new_functions) do
4      function = new_functions[function_index]
5      current_counts = {}
6      current_total = 0
7      algorithm_ratios = {}
8      algorithm_total = 0
9      for location in locations do
10         current_counts[location] =
             get_count_functions_at_location(location,
             function.type)
11         algorithm_ratios[location] = 1 /
             get_run_time_at_location(location,
             function.type)
12         current_total += current_counts[location]
13         algorithm_total += algorithm_ratios[location]
14     differences = []
15     for location in locations do
16         difference = current_counts[location] −
             (algorithm_ratios[location] / algorithm_total × current_total)
17         differences.append({"location":location,
             "difference":difference})
18     differences.sort_ascending(key="difference")
19     schedule_on_location(differences[0]["location"]
       /* If location is not available,
          schedule on the first available
          location in the order of the
          differences array                    */
20     function_index += 1
```

Algorithm 1 gives a pseudocode snippet of our Inverse Response Time Ratio Scheduler.

The algorithm runs when there is a function invocation request waiting to be scheduled. The algorithm processes requests on a first-come-first-serve basis. It works by placing the request at a location so as to improve the throughput, as well as the completion time, of the overall system.

The algorithm relies on function profiles, i.e., for each function type, it builds up some ideal ratio to schedule requests to run on function instances at different locations. For example, if there are two locations loc_1 and loc_2, and the determined ideal ratio is 1:2, and there are 300 function requests in the queue, 100 requests are scheduled to loc_1, and 200 requests are scheduled to loc_2.

The algorithm keeps two dictionaries. The first dictionary (`algorithm_ratios[]` in Algorithm 1) contains the algorithm ratios for each function type. These are *desired* ratios that the scheduler always aims to achieve when distributing

requests over the different locations. As described earlier, these algorithm ratios are the inverses of average response times at the different locations. Denote the desired ratio as α_i for location i. The second dictionary (`current_counts[]` in Algorithm 1) keeps track of the number of function requests that are currently running for each function type, denoted as n_i, for location i.

Then, for each function type, for each location, we append to an array a tuple consisting of the location identifier and a value (lines 15-17 in Algorithm 1). That value is given by:

$$n_i - \alpha_i \times n$$

where $n = \sum n_i$ over all locations. The array is then sorted in ascending order, and then the function is scheduled on the first available corresponding location where there are free resources to schedule the request (lines 18-19 in Algorithm 1). That way, our algorithm attempts to increase the actual load n_i to reach the desired load $\alpha_i \times n$ at location i whose actual load is further behind its desired load value.

VI. Results

A. Simulation

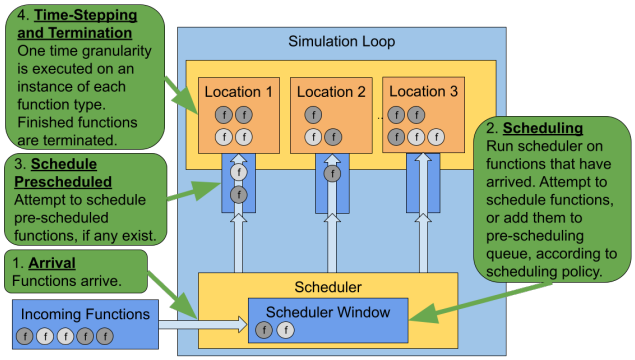

Fig. 2. Simulation model.

We have utilized our own simulation model, which is shown in Figure 2.[2] The simulation environment takes a scheduling algorithm, in the form of a Python function, as an input parameter. Scheduling algorithms would then be run separately and compared. The simulation environment also takes as input a numeric parameter that specifies the rate at which the function invocations / requests in the system queue should be dequeued and processed, which can also be thought of as a metric similar to the arrival rate of the function requests to the environment. The simulation environment also takes as input some configurations specifying information about the function types, the number of locations / clouds, and how each function performs at each location, which includes for example its execution time. In this work, we refer to the function response time to be the function execution time plus queuing time. The function response time is calculated as follows. First,

[2]Code and associated scripts for our simulations and FABRIC experiments will be publicly released upon paper acceptance.

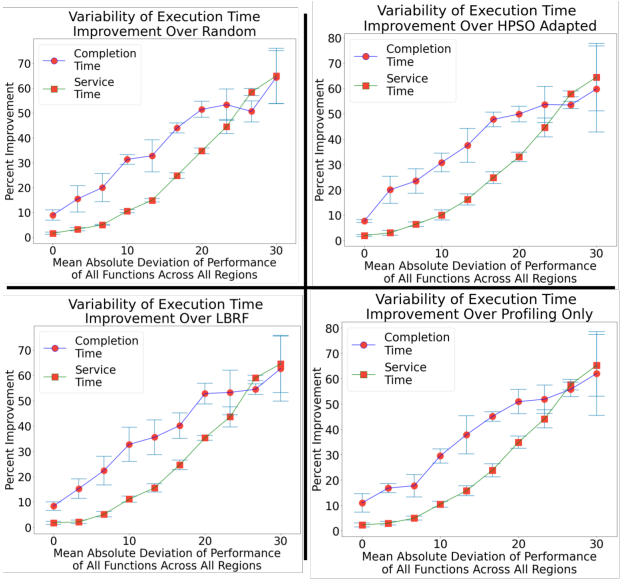

Fig. 3. Varying execution time.

the time when the function is sent to the location is recorded as timestamp A. Then, we compute timestamp B when the function sends back the response. Finally, we calculate the response time by subtracting timestamp A from timestamp B.

Now, we discuss the main result. We vary the statistical dispersion of the values of execution time for the functions, which is a measure of how scattered the values are. We use the "mean absolute deviation" to measure the statistical dispersion. It is defined as the average distance between "each value in the data set" and "the data set mean". The simulation results in Figure 3 show that as the dispersion increases, the improvement in our scheduler's function average response time and overall simulation completion time over the other schedulers increases – we observe improvement of up to 60% over a random scheduler, an HPSO-based scheduler, a most-free-resources (LBRF) scheduler, and a best profiling-based scheduler.

We have also varied the following parameters in Figures 4, 5, 6, 7 and 8:

- *statistical dispersion of resources* needed by each function type;
- *synthetic noise* added to the execution times of the function instances being serviced;
- *function types count* for a given set of parameter values;
- *ratio of the count of function instances* to run, arranged by instance type; and
- *location count* for a given set of parameter values, once with fixed total resources, and once with additional resources for each location addition.

The results in the figures show that the improvement in completion time and average response time is not correlated with the dispersion of resources across clouds (Figure 4), the number of function types (Figure 6), nor the number of cloud provider locations (Figure 8).

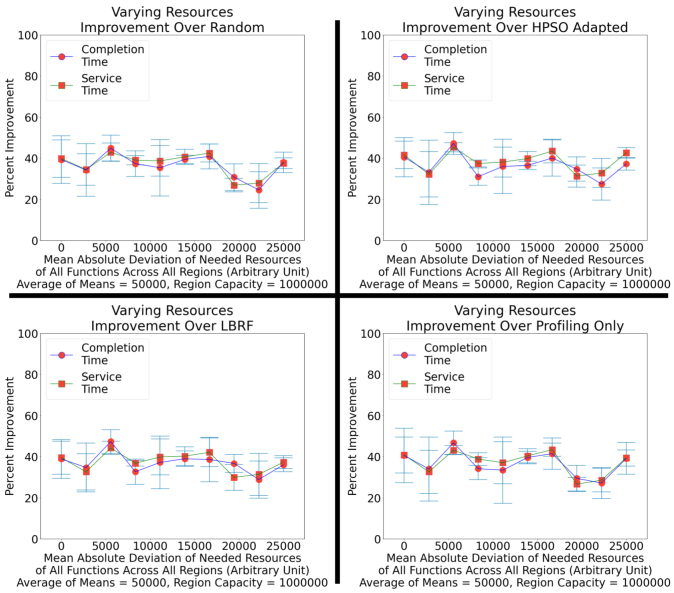

Fig. 4. Varying resources.

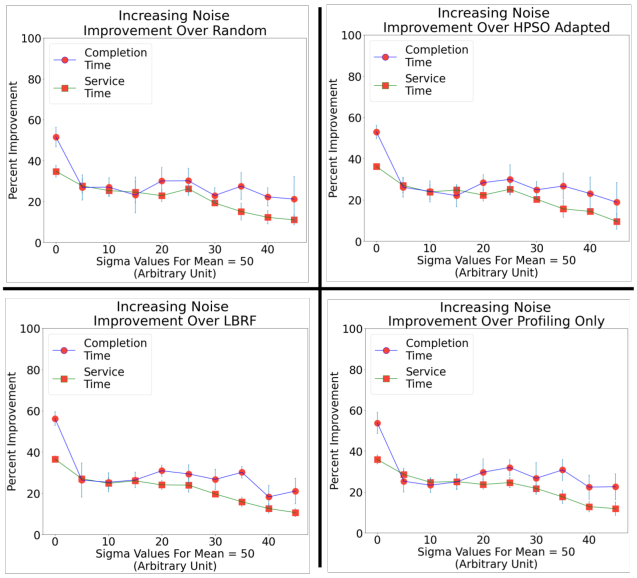

Fig. 5. Increasing noise.

For the noise (Figure 5), more noise decreases our improvement, but our inverse response time ratio scheduler still outperforms other schedulers. This is because adding noise gradually takes away the advantage of any profiling our algorithm captures and uses in function placement (request routing). The system would gradually become more random, and therefore, any scheduling technique would gradually no longer yield any advantage.

Moreover, only via empirical results, we get the best performance when the types of function instances in the queue are balanced – Figure 7 shows that our scheduler yields improvement of around 28% in average response time and

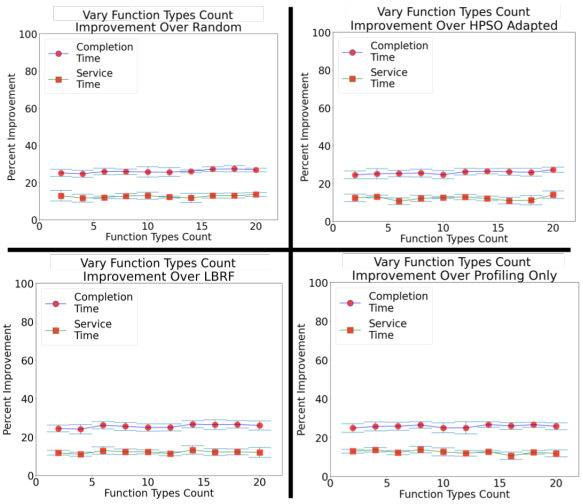

Fig. 6. Varying function types count.

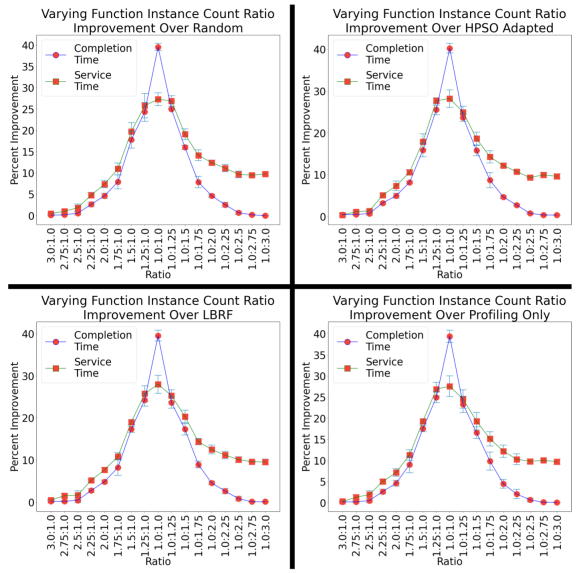

Fig. 7. Varying ratio of the count of running function instances, arranged by instance type.

40% in completion time when the load across function types is balanced.

B. *FABRIC [22] Experiment*

We used the setup shown in Figure 9, with details described in the caption. To emulate the serverless providers, we used the OpenWhisk open-source serverless cloud provider software [23]. The scheduler runs in Python. We have also used flask [24], the web application framework, to translate http requests to OpenWhisk invocations. We used SQlite [25] for hosting the database servers, and, for the database query, we issue a simple select * query.

We use the throttling of the network connections to set the connection speeds. We can set the bit rate of the horizontal links to be approximately double the speed of the cross links.

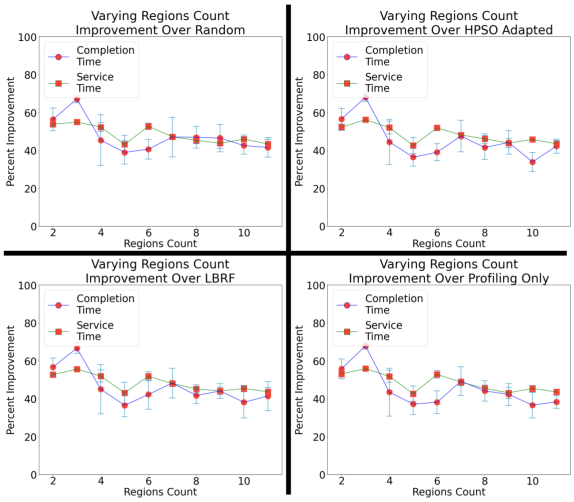

Fig. 8. Varying locations count with fixed total resources.

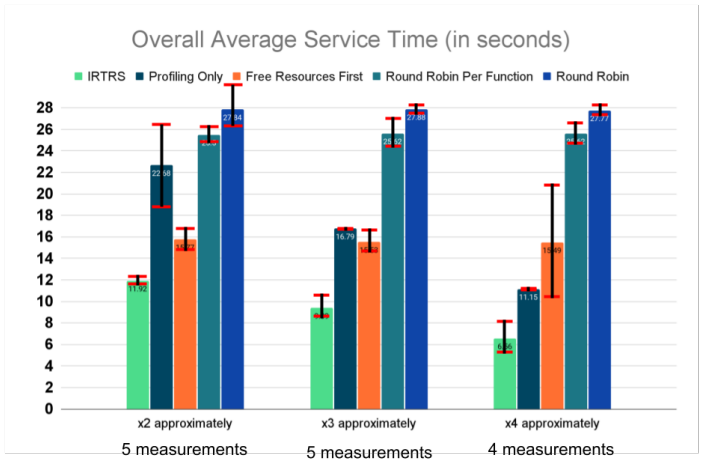

Fig. 10. Experimental results on FABRIC.

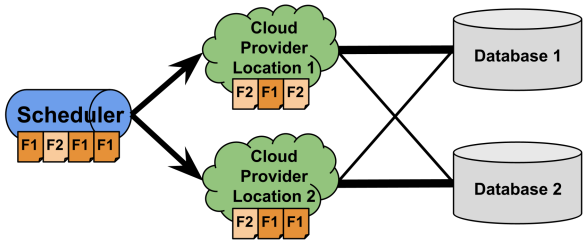

Fig. 9. Topology of the experiment on the FABRIC testbed. We have reserved 5 nodes with network cards connected to each other as shown. The role of each component is shown. The blue component is the scheduler, connected to two nodes simulating a distributed multi-location / multi-cloud / multi-region system. The two locations are connected to two nodes running as database nodes. The line width represents the bandwidth of the network connections between the cloud provider locations and the database locations. Functions F1 and F2 need to make database queries to database 1 and database 2, respectively, over those network links. The execution time is in theory calculated as (the amount of data to be transferred) divided by (the connection speed) plus additional processing time and other delays.

We set the bit rate of the links to 1Mbits and 512Kbits, respectively. Then, we emulate triple the speed for the horizontal links, and so on. We used `tc` to set the bit rate of the links – `tc` is a Linux tool to show and change network traffic settings [26]. Thus, we can vary the dispersion of function execution time, which is the main parameter of our study. Function execution time is calculated as the amount of data to be transferred, divided by the bit rate. The amount of data to be transferred is the same in all functions. There are other factors that might add to the execution time of the functions, such as function start time, processing time and disk I/O. Apart from function start time, the other factors should not be very significant in our case.

We evaluate our inverse response time ratio scheduling (IRTRS) algorithm on the emulation setup described above. Figure 10 shows that, for two functions, where the execution time of one is twice the other, we get around 32% improvement in average response time over the best of the other schedulers. We also show that the improvement increases as the dispersion increases. For around x3 dispersion, the improvement is around 65%, and for around x4 dispersion, the improvement is around 70%.

VII. Conclusion

We have presented an efficient scheduling algorithm for distributed serverless scenarios, for the case where there is contention over resources among function requests. Our scheduling algorithm improves the overall response times of function invocation requests, as well as the completion time of the queue of requests being served. We implemented our algorithm over a python simulation and over a testbed emulation. We compared our algorithm with multiple other algorithms and we highlighted how our algorithm would outperform them. One of our key findings is that the improvement of our algorithm over the rest increases as the dispersion of function performance across locations increases.

There are several extensions that we plan to investigate in the future. We will study how instances of different function types interact at a location. Some different function types might be competing on the same resource, and that therefore would affect the completion time and average service time. We will also investigate larger applications, each with an interconnected set of serverless functions.

Acknowledgment

This work has been supported by National Science Foundation Award CNS-1908677.

References

[1] "Inverse Response Time Ratio Scheduler Github Repository." [Online]. Available: https://github.com/842Mono/Inverse_Response_Time_Ratio_Scheduler

[2] M. Zhang, C. Krintz, and R. Wolski, "Edge-adaptable serverless acceleration for machine learning internet of things applications," *Software: Practice and Experience*, vol. 51, no. 9, pp. 1852–1867, 2021.

[3] T. Rausch, A. Rashed, and S. Dustdar, "Optimized container scheduling for data-intensive serverless edge computing," *Future Generation Computer Systems*, vol. 114, pp. 259–271, 2021.

[4] L. Baresi, D. Y. X. Hu, G. Quattrocchi, and L. Terracciano, "Neptune: Network-and gpu-aware management of serverless functions at the edge," *arXiv preprint arXiv:2205.04320*, 2022.

[5] A. F. Baarzi, G. Kesidis, C. Joe-Wong, and M. Shahrad, *On Merits and Viability of Multi-Cloud Serverless*. New York, NY, USA: Association for Computing Machinery, 2021, p. 600–608. [Online]. Available: https://doi.org/10.1145/3472883.3487002

[6] P. K. Gadepalli, G. Peach, L. Cherkasova, R. Aitken, and G. Parmer, "Challenges and opportunities for efficient serverless computing at the edge," in *2019 38th Symposium on Reliable Distributed Systems (SRDS)*, 2019, pp. 261–2615.

[7] K. Kaffes, N. J. Yadwadkar, and C. Kozyrakis, "Centralized core-granular scheduling for serverless functions," in *Proceedings of the ACM symposium on cloud computing*, 2019, pp. 158–164.

[8] M. Stein, "The serverless scheduling problem and noah," *arXiv preprint arXiv:1809.06100*, 2018.

[9] A. Mampage, S. Karunasekera, and R. Buyya, "Deadline-aware dynamic resource management in serverless computing environments," in *2021 IEEE/ACM 21st International Symposium on Cluster, Cloud and Internet Computing (CCGrid)*. IEEE, 2021, pp. 483–492.

[10] M. Sun, H. Zhuang, X. Zhou, K. Lu, and C. Li, "Hpso: Prefetching based scheduling to improve data locality for mapreduce clusters," in *Algorithms and Architectures for Parallel Processing*, X.-h. Sun, W. Qu, I. Stojmenovic, W. Zhou, Z. Li, H. Guo, G. Min, T. Yang, Y. Wu, and L. Liu, Eds. Cham: Springer International Publishing, 2014, pp. 82–95.

[11] B. Soltani, A. Ghenai, and N. Zeghib, "A migration-based approach to execute long-duration multi-cloud serverless functions." in *ICAASE*, 2018, pp. 42–50.

[12] L. Zhao, Y. Yang, Y. Li, X. Zhou, and K. Li, "Understanding, predicting and scheduling serverless workloads under partial interference," in *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, 2021, pp. 1–15.

[13] S. Liang, Z. Yang, F. Jin, and Y. Chen, "Data centers job scheduling with deep reinforcement learning," 2020.

[14] C. Delimitrou and C. Kozyrakis, "Qos-aware scheduling in heterogeneous datacenters with paragon," *ACM Trans. Comput. Syst.*, vol. 31, no. 4, dec 2013. [Online]. Available: https://doi.org/10.1145/2556583

[15] "Edgenet: A multi-tenant and multi-provider edge cloud," in *EdgeSys 2021 - Proceedings of the 4th International Workshop on Edge Systems, Analytics and Networking, Part of EuroSys 2021*. Association for Computing Machinery, Inc, Apr. 2021, pp. 49–54.

[16] M. Zaharia, D. Borthakur, J. Sen Sarma, K. Elmeleegy, S. Shenker, and I. Stoica, "Delay scheduling: A simple technique for achieving locality and fairness in cluster scheduling," in *Proceedings of the 5th European Conference on Computer Systems*, ser. EuroSys '10. New York, NY, USA: Association for Computing Machinery, 2010, p. 265–278. [Online]. Available: https://doi.org/10.1145/1755913.1755940

[17] M. Isard, V. Prabhakaran, J. Currey, U. Wieder, K. Talwar, and A. Goldberg, "Quincy: Fair scheduling for distributed computing clusters," in *Proceedings of the ACM SIGOPS 22nd Symposium on Operating Systems Principles*, ser. SOSP '09. New York, NY, USA: Association for Computing Machinery, 2009, p. 261–276. [Online]. Available: https://doi.org/10.1145/1629575.1629601

[18] B. Soltani, A. Ghenai, and N. Zeghib, "Towards distributed containerized serverless architecture in multi cloud environment," *Procedia computer science*, vol. 134, pp. 121–128, 2018.

[19] J. G. Quenum and J. Josua, "Multi-cloud serverless function composition," in *Proceedings of the 14th IEEE/ACM International Conference on Utility and Cloud Computing*, 2021, pp. 1–10.

[20] G. De Palma, S. Giallorenzo, J. Mauro, M. Trentin, and G. Zavattaro, "Topology-aware serverless function-execution scheduling," *arXiv preprint arXiv:2205.10176*, 2022.

[21] F. Samea, F. Azam, M. W. Anwar, M. Khan, and M. Rashid, "A uml profile for multi-cloud service configuration (umlpmsc) in event-driven serverless applications," in *Proceedings of the 2019 8th International Conference on Software and Computer Applications*, 2019, pp. 431–435.

[22] "Fabric testbed: Adaptive programmable research infrastructure for computer science and science applications." [Online]. Available: https://fabric-testbed.net/

[23] "Open whisk: Serverless cloud provider open source software." [Online]. Available: https://openwhisk.apache.org/

[24] "Flask: The web application framework." [Online]. Available: https://flask.palletsprojects.com/

[25] "SQLite: The SQL database engine library." [Online]. Available: https://www.sqlite.org/index.html

[26] "TC: The Linux tool to show and change network traffic settings." [Online]. Available: https://manpages.ubuntu.com/manpages/xenial/man8/tc.8.html