A Greedy Load Balancing Algorithm for FaaS Platforms

Youngsoo Lee and Sunghee Choi

Abstract— **In this paper, we propose a new load balancing algorithm for function-as-a-service (FaaS) platforms. We argue that the load balancing algorithm greatly affects the performance of FaaS platforms because heavy operations, such as virtualization and initialization, can be reduced using caching techniques. We demonstrate that a load balancing algorithm that provides higher locality could accelerate the FaaS platforms by increasing the cache-hit ratio, and propose a greedy load balancing algorithm optimized for FaaS. To generalize the experimental results, we conducted the experiment under three different caching policies that could be adopted in various FaaS platforms. Our evaluation reveals that load balancing algorithms with higher locality and a uniform load balance exhibit better results in all three caching policies, and our proposed algorithm achieves better performance compared to the state-of-the-art algorithms.**

Index Terms— **function-as-a-service, load balancing, locality, serverless computing.**

I. Introduction

Function-as-a-service (FaaS) has become a new paradigm in the cloud, where developers no longer have to manage servers and the platform executes codes on their behalf as needed on any scale [1]. As FaaS platforms handle everything required for server management burden, including instance selection, scaling, deployment, fault tolerance, monitoring, logging, and security patches, it has become the next step in the evolution of cloud computing architectures [2], [3]. Currently, FaaS is primarily provided by cloud companies such as Amazon, Microsoft, and Google, though it has also become a considerable option for building private FaaS platforms to distribute private computing resources or internal hardware in various companies and organizations [4].

However, known information about building a private FaaS platform is extremely limited because major FaaS providers have not opened their codes and policies, creating challenges for private companies to build FaaS platforms internally. Among them, we focus on the challenge of assigning numerous applications to limited resources in this paper. We point out that many of the traditional load balancing algorithms cannot be efficiently used in FaaS and demonstrate that the locality of the load balancing algorithm greatly influences FaaS performance.

Along with the increased interest in FaaS, many recent studies [1], [7], [11]–[13], [19] have proposed methods to increase FaaS performance. However, only a few papers [1], [13] have revealed the importance of a load balancer in FaaS. In particular, most of the methods to increase FaaS performance operate by using a cache to reduce the overhead from virtualization and initialization, which is greatly influenced by the load balancing algorithms. We further argue that caching techniques have already been applied on public FaaS platforms because a phenomenon called cold start, caused by the caching techniques, has been observed in major FaaS providers [8]–[10].

We hypothesize that locality in the load balancing algorithm is primarily related to the cache-hit ratio, which is closely related to performance on a FaaS platform. Towards higher FaaS performance, we propose a new load balancing algorithm called GRAF (GReedy Algorithm for Faas platforms), which maximizes the locality while preserving the load balance. We argue that increasing the locality would improve the performance of FaaS platforms by increasing the cache-hit ratio and reducing redundant procedures. To evaluate the effect of locality in FaaS load balancers, we simulate realistic workloads and assess the performance of a FaaS framework with various load balancing algorithms. Moreover, we conduct experiments under three different caching configurations because public and private FaaS platforms could adopt various caching techniques to enhance their performance. Our evaluation demonstrates that load balancing algorithms with a higher locality and load balance achieve higher performance in all three caching policies, and GRAF performs better than assessed state-of-the-art algorithms.

II. Related Work

A. Locality on FaaS Load Balancers

Although recent studies [1], [5], [7] have argued that load balancers should consider locality, the existing load balancers in open-source FaaS platforms, such as OpenWhisk [20], Fission [21], and OpenLambda [22], use simple algorithms that only deal with the load imbalance. A load balancing algorithm that considers locality in FaaS platforms was recently proposed by Aumala *et al.* [13]; however, their work has a limitation in that they only consider the cache-hit ratio of the downloaded packages of the applications (i.e., Pypi packages or npm packages). While they demonstrate the importance of load balancing algorithms in only a limited package reuse scenario, we expand the caching scenario to three to generalize the relationship between the load balancing algorithm and FaaS performance.

This work was supported by Institute of Information & communications Technology Planning & Evaluation (IITP) grant funded by the Korea government(MSIT) (No.2019-0-01158, Development of a Framework for 3D Geometric Model Processing).

Youngsoo Lee and Sunghee Choi are with the School of Computing, Korea Advanced Institute of Science and Technology (e-mail: youngsoo.lee@ kaist.ac.kr, sunghee@kaist.edu).

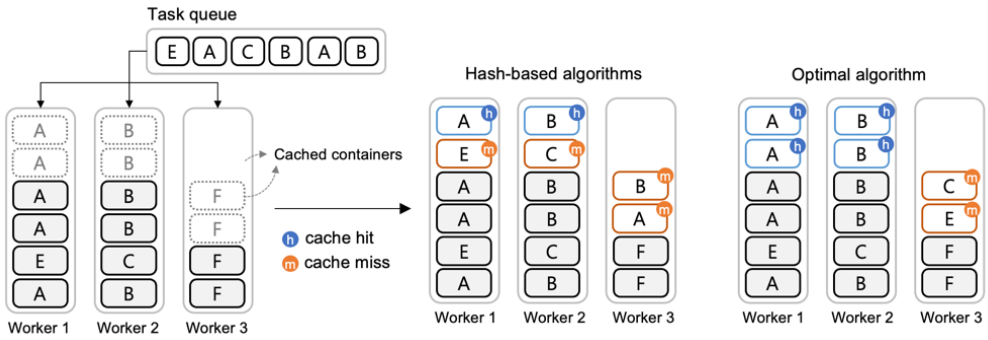

Fig. 1. Scenario in which many cache-misses occur when using hash-based load balancing algorithms.

B. *Load Balancing Algorithms on FaaS*

Round robin and least loaded are the common algorithms adopted in various load balancers, where round robin distributes the requests uniformly between the worker nodes, and least loaded assigns a task to the worker node with the least number of active connections. Although both algorithms are intuitive and work well in terms of load balance, they are not designed with the characteristics of FaaS because they do not consider locality.

A simple strategy to improve locality in the distributed system is to partition the request namespaces and assign the request to a particular node. For instance, a hash function can be used to perform the partitioning. Consistent hashing [14] is the most widely used algorithm in this case, however, this approach suffers from load imbalance under skewed workloads. Algorithms that only pursue locality cannot be used directly in FaaS, but some extended algorithms that also embrace load balance should be considered in FaaS.

One interesting algorithm is consistent hashing with bounded load (henceforth consistent hashing w/ BL) [15], a variation of the consistent hashing algorithm that limits the maximum active connections per each node. This algorithm is designed to maximize locality while minimizing load imbalance, and it seems to have a goal similar to the FaaS load balancing system. Furthermore, Aumala *et al.* [15] propose the PASch algorithm as a load balancing algorithm for FaaS platforms, by leveraging consistent hashing w/ BL. Although they show that they could improve FaaS performance with their algorithm, their algorithm sacrifices load balance and even produces relatively lower locality. In this paper, we analyze the limitations of the off-the-shelf algorithms and propose an algorithm that achieves higher locality and better load balance.

III. Limitations of Hash-based Algorithms

State-of-the-art FaaS load balancing algorithms are written based on the hash algorithm. However, considering the mechanism of the hash-based algorithms, cache-hit ratio commonly drops when the requests reach the maximum capacity. Fig. 1 illustrates the simplified scenario revealing the limitations of the hash-based algorithms when a FaaS platform uses caches to enhance its performance. Assume that each worker node can run up to six tasks concurrently and that the worker creates cached containers with the LRU (Least Recently Used) or LFU (Least Frequently Used) policy. In this case, the optimal algorithm can minimize the cache-miss to two times, but the hash-based algorithms incur four cache-misses out of six upcoming tasks. Unfortunately, we cannot solve this problem by creating more cached containers due to the computing resource restrictions. Configuring the load balancer to check the cached container status and distributing tasks according to the results may be a solution, but it is difficult to implement and the cost of internal communication increases (cache statuses may change rapidly).

As we cannot completely predict new incoming tasks, we cannot always optimally distribute the tasks. However, we can perform the load balancing in a smarter way. For example, we can recognize that *Tasks A* and *B* have been requested more than the other tasks, and we can preserve the space of *Workers 1* and *2* for the future requests of *Tasks A* and *B*, respectively. We cannot implement such a better optimized method only by leveraging the hash algorithm. For a higher locality in FaaS, we have to rethink the load balancing algorithm and design the scheme in a new way.

IV. Proposed Design

To overcome the limitations of the hash-based algorithms, we propose GRAF (GReedy Algorithm for Faas platforms), a completely new algorithm. The GRAF algorithm employs a tabular data structure and works greedily to maximize the locality and load balance. Moreover, it is designed to be optimized according to the caching policy, achieving better performance in FaaS platforms.

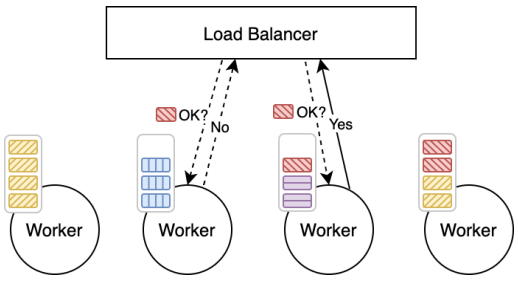

Fig. 2. Simplified concept of GRAF, where worker nodes can reject the tasks in order to maximize the locality.

A. *Tabular Data Structure*

The GRAF algorithm employs a tabular data structure for load balancing. While pure hash algorithms or random-based algorithms do not require a data structure and allow a de-centralized architecture, GRAF necessitates a centralized architecture as it uses a tabular data structure. Generally, a centralized architecture can cause bottlenecks, but FaaS platforms are relatively free from the bottlenecks because the

application execution time overwhelms the overhead from the centralized architecture. For this reason, the existing load balancing algorithms in FaaS including least loaded, consistent hashing w/ BL, and PASch have used a centralized data structure (for keeping the number of active connections).

While existing algorithms only record the number of active connections, GRAF has a table called *AssignedTable*, that contains information about which nodes are responsible for the application. Based on the *AssignedTable*, the load balancer maps the same application to the same nodes as much as possible. Algorithm 1 presents our basic node selection process by using the *AssignedTable*. If nodes are assigned to the application x and any of the assigned nodes are available, the node with the least loads is selected. If no node is assigned, the load balancer picks the least loaded node and registers it to the table.

Algorithm 1 Worker Node Selection

```
Data: List of worker nodes W,
      Table consists of assigned workers T
Input: Application ID x
W' := T[x] /* Assigned worker node set */
W* := available(W', x) /* Available nodes in W' */
if W* ≠ ∅ then
  return leastLoaded(W*)
else
  w := leastLoaded(available(W, x))
  if w is null then
    w := leastLoaded(W)
  end
  T[x] := T[x] ∪ {w}
  return w
end
```

Assume that the *available* function returns true if the worker node has enough CPU capacity to run a new task. In this case, the algorithm still suffers from the scenario in Fig 1. Furthermore, Algorithm 1 does not include a process for deleting entries in the *AssignedTable*. Periodical cleanup of the table would be one of the solutions, but it generates the challenge of deleting entries in dynamic FaaS platforms. In GRAF, we redefine the *available* function to solve both problems. Instead of writing the *available* function from the load balancer's viewpoint, we define it from the worker's viewpoint. The *available* function works for the benefit of the worker node, for example, it might not return true even if the worker has enough capacity. With this greedy approach, we can easily optimize the *AssignedTable* and maximize the locality.

B. Greedy Approach to Optimize the Table

In GRAF, worker nodes are responsible for optimizing the table and increasing the locality in GRAF. Specifically, we grant permissions to the worker nodes to deny some tasks and to delete entries in *AssignedTable* under certain conditions. Fig. 2 illustrates a brief overview of our greedy algorithm design. Note that the load balancer does not physically ask the actual worker nodes. All decisions are made locally in a load balancer, yet the algorithm is written based on the viewpoint of the worker nodes.

Before describing how we employ greedy approach to the load balancing algorithm, we introduce three states of the worker: *full*, *busy*, and *free*. Each worker node belongs to the one of the three states depending on the number of the running tasks. The *full* state indicates too many running tasks, so a new task cannot be run on the node. The *busy* state is an intermediate state where the worker is not full, although many tasks are running. Finally, in the *free* state, the worker can afford to accept a new task. When the worker node receives a new task, it makes different decisions based on these states. In the *full* state, the request is always rejected. In the *free* state, workers always accept the task. In contrast, in the *busy* state, workers selectively accept the task through their own decisions.

From the worker's viewpoint, the best way to maximize the locality is to receive the same types of applications as much as possible. In the *busy* state, worker nodes have permission to decide whether to accept or reject a task to achieve their goal. In detail, busy workers accept the task only if it is one of the *major applications*, which is a set of most running applications in the worker. Let the number of the running tasks of application x_i in the worker w be $N_w[x_i]$. If four applications are assigned to worker w, $N_w[x_i] = \{x_1: 2, x_2: 5, x_3: 3, x_4: 1\}$, and $T_{CACHESIZE} = 2$, then the major applications are x_2 and x_3. In the *busy* state, worker receives the task only if it belongs to the application x_2 or x_3. In GRAF, the number of major applications is decided by the threshold variable $T_{CACHESIZE}$, which is closely related to the cache size and considerably optimizes the algorithm for FaaS.

Algorithm 2 Available Function

```
Data: Number of running tasks whose app ID is x_i in
      worker w: N_w[x_i]
Input: Worker node w, Request application ID x
n_total := Σ_i N_w[x_i]
if n_total ≥ T_FULL then
  /* Full state: Always reject */
  return false
else if n_total ≥ T_BUSY then
  /* Busy state: Only accept major applications*/
  majorApps := popMaxItems(N_w, T_CACHESIZE)
  if x ∈ majorApps then
    return true
  else
    T[x] := T[x] - {w}
    return false
  end
else
  /* Free state: Always accept*/
  return true
end
```

Algorithm 2 presents the redefined *available* function. If the worker node rejects the application in the *busy* state, it is also removed from the *AssignedTable*. Consequently, the entries in the table will no longer grow indefinitely and the locality could be further improved. Fig. 3 presents a sample scenario of task assignment procedures with *AssignedTable.* We design the simple yet effective algorithm by leveraging the table search and least loaded mechanism and redefining

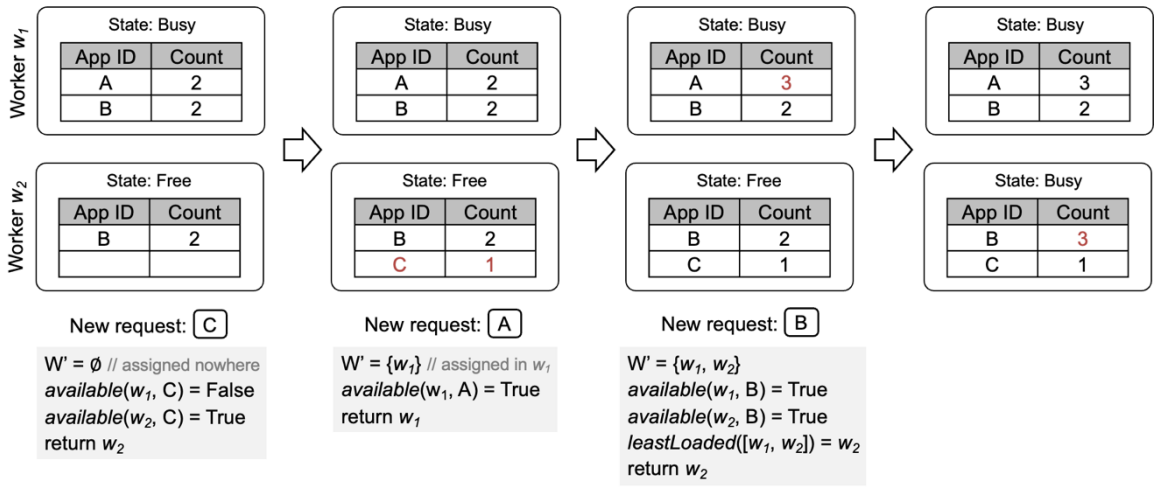

Fig. 3. Sample scenario of application assignment procedures and changes of entries in *AssignmentTable* in GRAF.

the available function.

Now looking back to Fig. 1, then we can find that GRAF could work as an optimal algorithm because GRAF preserves the space of the *Workers 1* and *2* as they are in the *busy* state. We argue that GRAF could increase the cache-hit ratio with the greedy approach even when the hash-based algorithms have limitations. Furthermore, GRAF could achieve a more uniform load balance with the notion of the three states of the workers. In Section 6, we demonstrate that GRAF performs well in terms of locality and load balance with the experiments.

V. Experimental Design

We implemented a micro FaaS framework with caching techniques and a simulator to assess the performance of the load balancing algorithm in a FaaS environment. We introduce three caching policies and a realistic workload simulation in this section.

A. Three Caching Policies

Cloud providers or private companies can apply various caching techniques depending on their situation. Therefore, instead of measuring the performance on a particular FaaS framework, we demonstrate the effect of the locality on various caching policies that can be adopted in various FaaS platforms. Among them, we introduce three policies as representative examples where we implemented them on our micro FaaS framework.

P1. **When there is a limitation in cache size of downloaded user codes and performing automated removal** - Oakes *et al.* [16] show that large libraries of FaaS functions could harm the overall performance. Further, Aumala *et al.* [13] propose a new load balancing algorithm to map applications that use the same packages to run on the same node. In Aumala's experiments, they demonstrate that the cache-hit ratio of the downloaded user codes affects the overall performance of the FaaS platform and that load balancing algorithms could improve the performance by ensuring the locality for the heavy packages. In our first policy, we set a 100 MB size limit for downloaded user functions for each worker node and performed automated removal with the least recently used policy in this experiment. (LRU, size-based).

P2. **When the platform provides pre-warmed container pools** - Lin *et al.* [12] propose a pool-based approach to mitigate virtual machine initialization overhead on FaaS platforms. Their idea is to reduce the preparation time by pre-building the environment before the function is executed. Although they do not consider the locality in their work, we expect the locality to affect the performance in this technique because the pre-warmed container acts as a cache in this policy. As the worker node resources are limited to prepare few applications, the cache-hit ratio is directly related to the performance, and if the load balancing algorithm provides higher locality, higher performance might be achieved. In our second policy, we configured the worker node to create six pre-warmed containers for the two least recently used applications (LRU, number-based).

P3. **When the platform supports an application-level sandbox** - The concept of an application-level sandbox is suggested by Akkus *et al.* [11] to mitigate virtual machine overhead. This notation is also called two levels of isolation: 1) isolation between different applications and 2) isolation between the functions of the same application. In this policy, FaaS platforms share the environment between the functions of the same application, and they indicate that the overall overhead of virtualization can be reduced. However, if the functions of the same application are mapped into different nodes, this technique does not work. To maximize the efficiency of this idea, a higher locality of the load balancing algorithm should be followed. In our third policy, we implemented the application-level sandbox in our FaaS framework and set the lifetime of an idle container to 5 seconds (lifetime-based).

B. Realistic Workloads Simulation

In the FaaS platform, applications(functions) are called by specific events after developers register them. There are many kinds of events, but they can be classified into two types: periodically triggered events (e.g., a batch process called daily) and unpredictable events (e.g., an API call). We configured the experiment according to the hypothesis that

TABLE I: COMPARISON OF THE STATE-OF-THE-ART LOAD BALANCING ALGORITHMS. COEFFICIENT OF VARIATION (c_v): LOWER IS BETTER. CACHE-HIT RATIO: HIGHER IS BETTER. AVERAGE EXECUTION TIME: LOWER IS BETTER. SPEEDUP: HIGHER IS BETTER

	Algorithm	Load balance(c_v)	Cache-hit	Avg. Time	Speedup
P1	Random	0.21	44%	6.5s	baseline
	Round Robin	0.16	44%	5.9s	1.1x
	Least Loaded	**0.05**	41%	5.2s	1.2x
	Consistent Hashing [14]	2.45	80%	31.7s	0.2x
	Consistent Hashing w/ BL [15]	0.51	78%	3.7s	1.7x
	PASch [13] Variation[1]	0.45	79%	3.8s	1.7x
	GRAF (ours)	0.17	**82%**	**3.2s**	**2.0x**
P2	Random	0.34	22%	5.5s	baseline
	Round Robin	0.43	20%	3.6s	1.5x
	Least Loaded	**0.08**	17%	3.0s	1.8x
	Consistent Hashing	2.35	62%	22.5s	0.2x
	Consistent Hashing w/ BL	0.47	65%	2.8s	2.0x
	PASch Variation	0.42	66%	2.8s	2.0x
	GRAF (ours)	**0.10**	**78%**	**2.4s**	**2.3x**
P3	Random	0.35	68%	2.5s	baseline
	Round Robin	**0.12**	66%	1.8s	1.4x
	Least Loaded	0.15	65%	1.8s	1.4x
	Consistent Hashing	1.65	**95%**	22.0s	0.1x
	Consistent Hashing w/ BL	0.45	89%	1.5s	1.7x
	PASch Variation	0.33	90%	**1.4s**	**1.8x**
	GRAF (ours)	0.26	**94%**	**1.4s**	**1.8x**

these application requests are not entirely arbitrary but follow a specific pattern. Based on the published data, we designed a function call scenario that is as close as possible to the real FaaS eco-system.

We created 30 different sample applications to conduct the simulation, distributed according to purpose as follows: 36% for the web server, 27% for data processing, 20% for third-party integration, and 17% for internal tooling. This ratio was determined based on the serverless community survey results [2] introduced by Jonas *et al.* [17]. The application request frequency follows an exponential distribution, where the top 20% of applications account for 78% of the entire requests. Applications belonging to a web server and third-party integration are set to have short life cycles and a high frequency since unpredictable events are usually called frequently. Data processing and internal tooling applications are set to have a long duration, and they are set to be called at specific, predetermined times.

The sample applications are written in three languages (Python, NodeJS, and Java), which are the most popular in FaaS platforms.[2] As the programming language might affect the performance due to the nature of the language itself [18], composing the test environment with multiple languages may make the results closer to the real-world results. The detailed configuration is accessible at https://github.com/Prev/HotFunctions.

VI. EVALUATION

We ran real experiments on AWS EC2. We used nine m4.xlarge instances, one for the load balancer and eight for the worker nodes. To demonstrate our hypothesis that algorithms providing a higher locality and load balance could improve the FaaS performance, we measured load balance, locality, and the overall performance of the FaaS framework separately. For the load balance, we used the coefficient of variation (c_v) as an indicator, which is a measure of dispersion defined as the ratio of the standard deviation to the mean (σ/μ). For the locality, we used the cache-hit ratio of three caching policies. Last, we measured the average execution time of the applications to show the overall performance of the FaaS platform. We configured the parameters of the load balancing algorithms to the value that gives the highest performance.

Table 1 presents the results of our experiment. We compared the GRAF algorithm with popular load balancing algorithms state-of-the-art algorithms. As a result, GRAF performs best among state-of-the-art load balancing algorithms on FaaS, achieving 1.8x~2.3x speedup compared to the Random algorithm. Notably, GRAF outperforms state-of-the-art algorithms not only in the cache-hit ratio but also in load balance(c_v), which leads to an entire performance acceleration. While GRAF exhibits the best performance, the algorithms that consider both locality and load balance (consistent hashing w/ BL and PASch) also perform well. However, the algorithm that considers only the locality and not the load balance (consistent hashing) shows very poor performance.

In addition to the performance of the state-of-the-art algorithms, we further generalize the effects of the locality and load balance by conducting experiments using various algorithms. We generated algorithms by mixing up the algorithms with random functions and varying the threshold values. Fig. 4 illustrates the algorithms plotted by their locality (cache-hit-ratio) and load balance (c_v). Algorithms with a higher locality and better load balance achieve better performance. To attain good performance, both load balance and locality should be satisfied in all three experiments.

[1] As PASch is originally written to seek package-reuse of the applications, we modified it to use application ID instead of package ID in our experiments.

[2] Python and NodeJS are supported in AWS Lambda, Azure Functions, Google Cloud Functions, Fission, and Apache OpenWhisk. Java is supported in AWS Lambda, Azure Functions, and Apache OpenWhisk.

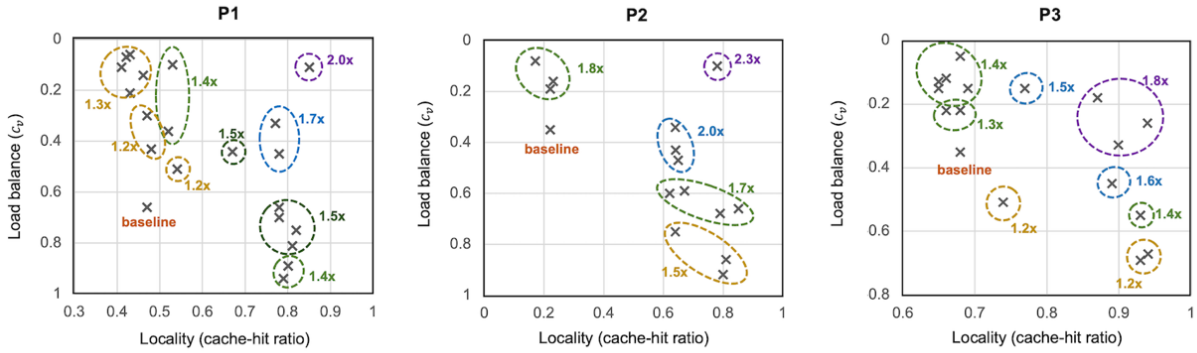

Fig. 4. Performance comparison of various algorithms by locality and load balance. Algorithms with higher locality and load balance achieve better performance.

VII. Conclusion

Function-as-a-Service (FaaS) is a novel concept that could change the paradigm of software product development. To vitalize FaaS computing, a load balancing algorithm that maximizes locality and load balance should be implemented. In this paper, we evaluated the effect of the FaaS load balancing algorithms by simulating realistic workloads with three different caching configurations that could be adopted in both public and private FaaS platforms. Towards higher FaaS performance, we further proposed a novel load balancing algorithm called GRAF, which employs a tabular data structure and greedy approach. We demonstrated that GRAF could achieve better results in the locality, load balance, and overall performance compared to existing schemes. Moreover, our evaluations revealed that the same caching technique can yield different results depending on the load balancing algorithm. In summary, we explored the relationship between locality and performance in FaaS platforms using various caching techniques, with the hope to provide further insight into the field of cloud computing.

References

[1] C. L. Abad, E. F. Boza, and E. Van Eyk, "Package-aware scheduling of faas functions," in Companion of the 2018 ACM/SPEC International Conference on Performance Engineering. ACM, 2018, pp. 101–106.

[2] "2018 serverless community survey: huge growth in serverless," https://serverless.com/blog/2018-serverless-community-survey-huge-growth-usage.

[3] I. Baldini, P. Castro, K. Chang, P. Cheng, S. Fink, V. Ishakian, N. Mitchell, V. Muthusamy, R. Rabbah, A. Slominskiet al., "Serverless computing: Current trends and open problems," in Research Advances in Cloud Computing. Springer, 2017, pp. 1–20.

[4] W. Ling, L. Ma, C. Tian, and Z. Hu, "Pigeon: A dynamic and efficient serverless and faas framework for private cloud," in 2019 International Conference on Computational Science and Computational Intelligence(CSCI). IEEE, 2019, pp. 1416–1421.

[5] S. Hendrickson, S. Sturdevant, T. Harter, V. Venkataramani, A. C. Arpaci-Dusseau, and R. H. Arpaci-Dusseau, "Serverless computation with openlambda," in 8th USENIX Workshop on Hot Topics in Cloud Computing (HotCloud 16), 2016.

[6] E. Van Eyk, A. Iosup, S. Seif, and M. Th¨ommes, "The spec cloud group's research vision on faas and serverless architectures," in Proceedings of the 2nd International Workshop on Serverless Computing. ACM, 2017, pp. 1–4.

[7] K. Solaiman and M. A. Adnan, "Wlec: A not so cold architecture to mitigate cold start problem in serverless computing," in 2020 IEEE International Conference on Cloud Engineering (IC2E). IEEE, 2020, pp. 144–153.

[8] T. H. Johannes Manner, Martin Endreß and G. Wirtz, "Cold start influencing factors in function as a service," in 2018 IEEE/ACM International Conference on Utility and Cloud Computing Companion (UCC Companion), 2018, pp. 181–188.

[9] W. Lloyd, S. Ramesh, S. Chinthalapati, L. Ly, and S. Pallickara, "Serverless computing: An investigation of factors influencing microservice performance," in 2018 IEEE International Conference on Cloud Engineering (IC2E). IEEE, 2018, pp. 159–169.

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

[11] I. E. Akkus, R. Chen, I. Rimac, M. Stein, K. Satzke, A. Beck, P. Aditya, and V. Hilt, "{SAND}: Towards high-performance serverless computing," in 2018 USENIX Annual Technical Conference (USENIX ATC 18), 2018, pp. 923–935.

[12] P.-M. Lin and A. Glikson, "Mitigating Cold Starts in Serverless Platforms: A Pool-Based Approach," arXiv e-prints, p arXiv:1903.12221, Mar 2019.

[13] G. Aumala, E. Boza, L. Ortiz-Avil´es, G. Totoy, and C. Abad, "Beyond load balancing: Package-aware scheduling for serverless platforms," in 2019 19th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing (CCGRID), May 2019, pp. 282–291.

[14] D. Karger, A. Sherman, A. Berkheimer, B. Bogstad, R. Dhanidina,K. Iwamoto, B. Kim, L. Matkins, and Y. Yerushalmi, "Web caching with consistent hashing," Computer Networks, vol. 31, no. 11-16, pp.1203–1213, 1999.

[15] V. Mirrokni, M. Thorup, and M. Zadimoghaddam, "Consistent hashing with bounded loads," in Proceedings of the Twenty-Ninth Annual ACM-SIAM Symposium on Discrete Algorithms, 2018, pp. 587–604. [Online]. Available: https://epubs.siam.org/doi/abs/10.1137/1.9781611975031.39

[16] E. Oakes, L. Yang, K. Houck, T. Harter, A. C. Arpaci-Dusseau, and R. H. Arpaci-Dusseau, "Pipsqueak: Lean lambdas with large libraries," in 2017 IEEE 37th International Conference on Distributed Computing Systems Workshops (ICDCSW), June 2017, pp. 395–400.

[17] E. Jonas, J. Schleier-Smith, V. Sreekanti, C.-C. Tsai, A. Khandelwal, Q. Pu, V. Shankar, J. Carreira, K. Krauth, N. Yadwadkar, J. E. Gonzalez, R. A. Popa, I. Stoica, and D. A. Patterson, "Cloud Programming Simplified: A Berkeley View on Serverless Computing", arXiv e-prints, p. arXiv:1902.03383, Feb 2019.

[18] D. Jackson and G. Clynch, "An investigation of the impact of language runtime on the performance and cost of serverless functions", in 2018 IEEE/ACM International Conference on Utility and Cloud Computing Companion (UCC Companion). IEEE, 2018, pp. 154–160

[19] S. Boucher, A. Kalia, D. G. Andersen, and M. Kaminsky, "Putting the "micro" back in microservice," in2018 USENIX Annual Technical Conference (USENIXATC 18). Boston, MA: USENIX Association, Jul. 2018, pp. 645–650.

[20] Apache OpenWhisk is a serverless, open source cloud platform, [online] Available: https://openwhisk.apache.org/ [Last accessed: Mar 09, 2021]

[21] fission/fission: Fast and Simple Serverless Functions for Kubernetes, [online] Available: https://github.com/fission/fission [Last accessed: Mar 09, 2021]

[22] open-lambda/open-lambda: An open source serverless computing platform, [online] Available: https://github.com/open-lambda/open-lambda [Last accessed: Mar 09, 2021]