A Greedy Load Balancing Algorithm on Serverless Platforms Maximizing Locality

Youngsoo Lee Sunghee Choi
Korea Advanced Institute of Science and Technology

Abstract

This paper proposes a new load balancing algorithm that maximizes locality on serverless platforms. The existing load balancing algorithms aim to minimize load imbalance, distributing tasks as equally as possible. However, in serverless area, *locality* greatly affects performance since heavy operations such as virtualization and initialization can be reduced with caching techniques. In this paper, we demonstrate load balancing algorithms that provide higher locality could improve the performance of serverless platforms by increasing the cache-hit ratio. We conducted three experiments under different caching techniques which were proposed in recent studies. In all of three experiments, load balancing algorithms with higher locality achieve better result, and our proposing algorithm shows best performance compared to the existing algorithms.

1 Introduction

Serverless computing is a term suggested by the industry to describe a programming model and architecture. The serverless paradigm offers the novel concept of a platform in the cloud, where developers no longer have to manage its servers and platform executes codes on their behalf as needed at any scale [2]. As a result, the serverless platforms can provide outstanding performance that makes writing scalable microservices easier and cost-effective, positioning themselves as the next step in the evolution of cloud computing architectures [5].

In serverless computing, an application usually consists of one or more functions — small, standalone, stateless components dedicated to handle specific tasks [16], which is the reason why serverless computing is also called Function-as-a-Service (FaaS). As functions are stateless and supposed to return all resources after execution, cloud providers can execute functions on any of the nodes and are able to more optimally manage resources [15]. However, assigning numerous applications to limited resources has become a new challenge. Several papers [2, 6, 15] that have already addressed with this issue argue that cloud providers should aim to minimize load imbalance and maximize code and data locality when designing the load balancer in a FaaS platform. As far as we know, the latest load balancers already achieve outstanding performance under load imbalance. However, locality has not yet been identified as an important goal.

Locality is mainly related to the cache-hit ratio, which on a FaaS platform is closely related to performance. One of the advantages of FaaS is that the application can be run on any of the nodes, although there will be a number of applications registered by the users, making it impossible to prepare all the environments of the applications in the worker nodes. This means that, each time the function is executed, it involves the initialization and virtualization process [11]. As this overhead causes a long function execution time, both industry and academia are trying to mitigate the problem with caching techniques [4, 7, 8, 10, 11]. In this situation, the locality of the load balancing algorithm might greatly affect the performance of the FaaS system.

In this paper, we propose a new load balancing algorithm that maximizes locality while also minimizing load imbalance for serverless platforms (section 3). We hypothesize that increasing the locality of applications would improve the performance on serverless platforms by increasing the cache-hit ratio and reducing the redundant procedures (section 2). To show the relationship, we implemented three other caching techniques that are proposed in recent studies on our micro FaaS framework (section 4). Our evaluation shows that the same caching technique can yield different results depending on the load balancing algorithm, while our algorithm performs best compared to the existing algorithms (section 5).

2 Background and Related Works

Locality and performance on serverless platforms Increasing locality in serverless computing could improve performance in various ways. We will show the relationship between locality and performance by illustrating three recent studies.

First, locality is related to the frequency of downloading codes and libraries for function execution. Oakes et al. [14] showed that large libraries of FaaS functions could harm the overall performance, while Aumala *et al.* [4] proposed a new load balancing algorithm to map applications that uses the same packages to run on the same node. By ensuring locality for the heavy packages, they showed that they could lower the overhead of downloading packages, improving performance overall.

Second, higher locality could increase the cache-hit ratio when the serverless platform supports pre-warmed environments or pooled containers. Lin *et al.* [10] proposed a pool-based approach to mitigate VM initialization overhead on serverless platforms. Their idea was to reduce the preparation time by pre-building the environment before the function is executed. Because the pre-warmed container acts as a cache in this technique, the cache-hit ratio would impact the overall performance, and if the same functions are mapped to the same node, a higher cache-hit ratio might be achieved.

Third, the locality is useful when the platform can share the environment between the functions of the same application. Akkus *et al.* [3] proposed an idea to mitigate VM overhead through a policy called two levels of isolation: 1) isolation between different applications, and 2) isolation between the functions of the same application. With this policy, they showed that the overhead of virtualization can be reduced since a low isolation level is much faster than providing higher isolation level. However, if the functions of the same application are mapped to different nodes, this technique will not work. To maximize the efficiency of this idea, higher locality should be followed.

Load balancing algorithms Existing load balancers in FaaS platforms, such as OpenWhisk[1], Fission[2], and OpenLambda[3], use simple algorithms that only deal with the load imbalance. A solution could be to use an available technique such as Consistent Hashing [9], to route all functions of the same application to the same worker node, thus maximizing the cache-hit ratio. However, this approach suffers from load imbalance, particularly under skewed workloads [4, 13].

Recently, a load balancing algorithm that considers locality in serverless platforms has been studied by Aumala *et al.* [4]. Nevertheless, their algorithm has a limitation, since they only aimed to increase the cache-hit ratio of large packages, and the method yielded an even higher load imbalance compared to existing algorithms. Our design is slightly more complex than theirs, although our algorithm shows higher locality and lower load imbalance and can be applied in general situations. What is more, instead of the hash-based algorithm used by Aumala *et al.*, we opted for the table-based greedy method. The details of our design is discussed in the sections below.

[1] https://openwhisk.apache.org
[2] https://github.com/fission/fission
[3] https://github.com/open-lambda/open-lambda/

3 Proposed Design

To maximize locality while also minimizing load imbalance, we propose a novel algorithm design. First, we use a table data structure, and second, we adopted a greedy approach to optimize the table.

3.1 Table-based Load Balancing Algorithm

We use a table-like data structure for load balancing. Until now, hash or random-based algorithms that do not require centralized data structures have been used for load balancing to minimize overhead. However, we propose a table-based algorithm that has previously not been used, we hypothesize that the side effects from using data structures will be lower than the advantages of high locality and low node imbalance.

Algorithm 1: Node selection algorithm

```
Data: List of worker nodes $W$,
Table consists of function and assigned nodes $T$
Input: Application $a$
$S \leftarrow lookup(T, a)$
if $S \neq \emptyset$ then
    /* Select the least loaded worker among
       assigned nodes                        */
    $w_t \leftarrow null$
    for $w_i \in S$ do
        if $available(w_i, a)$ then
            if $w_t = null$ or $load(w_t) > load(w_i)$ then
                $w_t \leftarrow w_i$
            end
        end
    end
    if $w_t \neq null$ then
        return $w_t$
    end
end
/* Select the least loaded worker among
   all nodes                                 */
$w_n \leftarrow null$
for $w_i \in W$ do
    if $available(w_i, a)$ then
        if $w_n = null$ or $load(w_n) > load(w_i)$ then
            $w_n \leftarrow w_i$
        end
    end
end
/* Assign the node to the table              */
$T[a] \leftarrow T[a] \cup \{w_n\}$
return $w_n$
```

The *AssignedTable* in our load balancer contains information about which nodes are assigned to the application. Based on this table, the load balancer maps the tasks of the same application to the same nodes as much as possible.

Algorithm 1 shows our basic node selection process with using the *AssignedTable*. If there are nodes assigned to application a and any of the assigned nodes is available, the node with the least loads is selected. If there is no assigned node, the load balancer picks the least loaded node and registers it to the table. In summary, our algorithm is designed primarily to increase locality using the table, and secondarily, it seeks to minimize node imbalance as musch as possible.

3.2 Greedy Approach to Optimize the Table

The above algorithm includes the assigning process of the table, yet there is no process for deleting the assigned nodes. It might be seen that cleaning up the table is required after the task is executed, nonetheless, there is a problem for which it is almost impossible to find an optimal solution: how to delete entries in dynamic situations. Specifically, it is not possible in FaaS to predict future requests from the users, and it is also difficult to find out when the running tasks will be finished.

To deal with this problem, we adopted a *greedy* approach by breaking the problem into small local problems. In our design, instead of performing the table optimization centrally, worker nodes are responsible for optimizing the table and increasing the locality. More specifically, we grant permissions to the worker nodes to deny some tasks and to delete entries from the table under certain conditions.

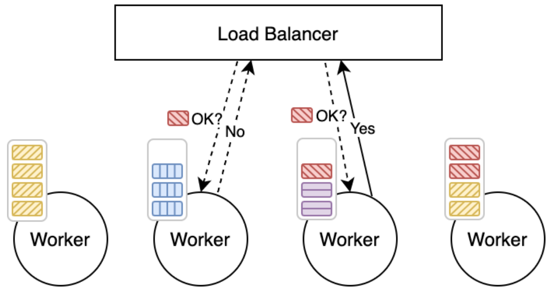

Figure 1: Simplified architecture of our load balancer design. Worker nodes can reject the tasks in order to maximize locality.

Three states of the worker A worker node has three states, called *Full, Busy, and Free*, depending on the number of running tasks (i.e., functions). The *Full* state means that there are too many running tasks so a new task cannot be run on the node anymore. The *Busy* state is an intermediate state in which the worker node is not full, although there are quite a lot of tasks 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 job is always rejected. In the *Free* state, workers always accept the tasks, while in the *Busy* state, workers selectively accept the tasks by their own decisions.

Selective execution in the *Busy* state From the node's point of view, the best way to maximize locality is to receive an as small as possible number of distinct applications. We note that, in the *Busy* state, worker nodes have permission to decide whether to accept or reject a task to achieve their goal. In our algorithm, worker nodes accept the task only if it is one of the *major applications*.

Let all the applications running at the worker node be $A = \{a_1, a_2, ...a_n\}$, and the number of running tasks of application a_i be $N[a_i]$. In our algorithm, we consider the *major applications* as the top t_m applications sorted by $N[a_i]$, where t_m is variable. For example, if $A = \{a_1, a_2, a_3, a_4\}$, $N[a_1] = 4, N[a_2] = 3, N[a_3] = 2, N[a_4] = 1$, and $t_m = 2$, the major applications would be a_1 and a_2. In this algorithm, the number of major applications is decided by the variable t_m, which closely related to the cache size.

Algorithm 2: Available function of the worker node

```
Data: All applications running in the worker $A$,
Number of running tasks of the application $a_i$ $N[a_i]$
Input: Worker node $w$, Application $a$
$n_{total} = \sum_{a_i}^{A} N[a_i]$
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                                   */
    $S \leftarrow sorted(A, (a_1, a_2) => N[a_1] > N[a_2])$
    for $i \leftarrow 0$ to $min(t_m, |S|)$ do
        if $a = S[i]$ then
            return $true$
        end
    end
    /* De-assign the node if the
       application is not the major one               */
    $T[a] \leftarrow T[a] - \{w\}$
    return $false$
else
    /* Free state: Always accept                      */
    return $true$
end
```

Algorithm 2 shows the task acceptance process performed by each worker node. If the worker node rejects the application in the *Busy* state, the application is also removed from the assigned table. With this process, the entries in the table will no longer grow indefinitely. Not only does it help with the deletion of the table entries, but this algorithm also prevents extra assignment processes in the table. Looking back to the Algorithm 1, we can see that there are two scenarios in which a task is assigned to a new node: 1) there are no nodes assigned to the task, or 2) the assigned nodes are not available. The first case is inevitable, although for the second case, the situation is different. If the assigned node for the application is not available, it leads to other nodes being assigned the additional application, obviously reducing the locality. With the rejection and de-assignment process based on the three states of the worker, our algorithm prevents the nodes from being unavailable, and leads to reducing unnecessary extra assignment processes to the table.

Consequently, our algorithm for optimizing the table was designed to see the problem only from the worker node's point of view. However, we claim that this greedy approach could improve the performance of the whole system. Following sections shows how this algorithm actually performs well in terms of locality and load imbalance.

4 Implementation

The relation between locality and performance in serverless platforms depend a great deal on policies and internal techniques, which makes it hard to generalize the result. What is more, major serverless providers, such as Amazon and Microsoft, have not opened their codes and policies, which makes it more difficult to conduct experiments. Rather than measuring the performance of a particular FaaS framework, we focused on how load balancing algorithms could affect *caching techniques* or policies that can be used in serverless platforms. We developed a micro FaaS framework with the core features, to which we added three different optional features as caching policies based on the studies discussed in Section 2:

P1: When there is a limitation of cache size of downloaded user codes (conducted by [4])

P2: When the framework provides pre-warmed container pools (suggested by [10])

P3: When the framework supports two-levels of isolation technique (suggested by [3])

For each policy, we measured the performance with five different load balancing algorithms as follows:

- Round Robin: Distributes the requests uniformly between the worker nodes.
- Least Loaded: Assigns a task to the worker node that has the least number of active connections (akin to NGINX's least-connected).
- Consistent Hashing with bounded load [12]: Variation of the Consistent Hashing algorithm that limits the load per each node.
- PASch Extended: PASch is an algorithm that maximizes the cache-hit ratio of the packages in FaaS [4]; we implemented an extended version of it by changing the package ID to an application name.
- Ours: Proposing algorithm in this paper.

In our framework, user applications are run in Docker containers as virtual environments. Each application has a unique name and is written in Python, Java, or NodeJS. Our framework is written in Go. The detailed descriptions and codes are accessible at github.com/Prev/HotFunctions.

5 Experimental Evaluation

We ran real experiments on AWS EC2. We used nine m4.xlarge instances, one for the load balancer and eight for worker nodes. To conduct the simulation, we created 30 different sample applications, distributed by 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 result [1] introduced by Jonas *et al.* [8]. The application request frequency follows the exponential distribution, and applications with short execution times are set to be executed more frequently[4]. The parameters t_{full}, t_{busy} and t_m of our algorithm were set as 8, 5, and 3 respectively. Similarly, the load threshold of PASch and ε of Consistent Hashing with bounded load were both set as 8.

Metrics To prove our hypothesis, we measured load imbalance, locality, and the real performance of the FaaS framework. For the load imbalance, we used the *coefficient of variation* as an indicator, which is a measure of dispersion defined as the ratio of the standard deviation to the mean: $c_v = \sigma/\mu$, while the *cache-hit ratio* was used to measure locality. The meaning of the cache-hit depends on the policy, which will be explained in detail in the paragraph below. Last, we measured the average execution time of the applications to show the real performance.

Detailed configuration of the three policies *P1*: When limiting the cache size of downloaded user codes, cache-hit means that the code of the application is cached locally. We set an 100MB-size limit for downloaded user functions for each worker node with the least recently used policy in this experiment. (LRU, size-based) *P2*: In this case, cache-hit means that a pre-warmed container for the application is present. We conducted experiments with the policy that each worker node creates six pre-warmed containers for the two least recently used applications. (LRU, number-based) *P3*: The isolation between different applications and isolation between functions

[4]Detailed configuration is accessible at `https://github.com/Prev/HotFunctions/tree/master/sample_functions`

	Algorithm	**Loads per second**			**Cache-hit ratio**	**Execution Time**		
		μ	σ	c_v		Cache-hit	Cache-miss	Avg.
P1	Round Robin	4.77	2.44	0.51	44%	3748ms	8122ms	6193ms
	Least Loaded	4.49	2.26	**0.50**	39%	3596ms	7236ms	5806ms
	Consistent Hashing w/ BL	3.36	2.39	0.71	74%	2964ms	8681ms	4404ms
	PASch Extended	3.30	2.06	0.62	75%	2595ms	9253ms	4235ms
	Ours	2.92	1.50	**0.51**	**77%**	2546ms	7645ms	**3714ms**
P2	Round Robin	2.87	1.58	0.55	20%	2914ms	3716ms	3552ms
	Least Loaded	2.73	1.40	**0.51**	18%	2707ms	3413ms	3284ms
	Consistent Hashing w/ BL	3.09	2.48	0.80	65%	3403ms	4588ms	3811ms
	PASch Extended	2.67	1.79	0.67	64%	2571ms	4125ms	3130ms
	Ours	2.42	1.33	**0.55**	**80%**	2311ms	4719ms	**2774ms**
P3	Round Robin	1.67	0.89	**0.53**	64%	537ms	3573ms	1606ms
	Least Loaded	1.66	1.00	0.61	67%	578ms	3655ms	1569ms
	Consistent Hashing w/ BL	1.46	1.52	1.04	**94%**	764ms	10531ms	1262ms
	PASch Extended	1.45	0.96	0.66	91%	612ms	8349ms	1251ms
	Ours	1.45	0.89	**0.61**	**94%**	758ms	9381ms	**1237ms**

Table 1: Overall experimental results. c_v: lower is better. Cache-hit ratio: higher is better. Execution Time: lower is better.

of the same application are treated differently in this experiment. In other words, multiple tasks of the same application are run in a single container. However, regarding resource optimization, idle containers have limited lifetime; they are killed when there is no further request of the application. In this experiment, cache-hit means that live containers exist, and that the lifetime of an idle container is set to 10 seconds. (lifetime-based)

Analysis Table 1 shows the result of our experiments. Regarding how the load imbalance is addressed, the Least Loaded algorithm and ours produce similar coefficients of variation (c_v). Considering the former algorithm focuses only on the load imbalance, this is an important achivement of our algorithm. Compared to the Consistent Hashing with bounded load and PASch Extended, our algorithm shows 33% lower c_v and 15% lower c_v, respectively.

Second, our algorithm achieves the highest cache-hit ratio in all experiments compared to the other four algorithms. In particular, the difference is largest in *P2*, which suggests that the concept of *major applications* in our algorithm fits well with the caching policy of this experiment.

Finally, we found that load imbalance and locality affect the overall performance in the serverless platform. In all three experiments, our algorithm has the shortest execution time; there is an 1.14x to 1.6x speedup for *P1*, 1.1x to 1.4x for *P2*, and 1.01x to 1.3x for *P3* compared to the other algorithms. Consistent Hashing with bounded load and PASch Extended also perform better than Round Robin and Least Loaded, as they show rather high locality. As we have seen, cache-based techniques on serverless platforms yield even greater performance when combining better load balancing algorithms in terms of locality and load imbalance.

In addition, we recorded the execution times of the load balancing algorithms themselves. The execution time of the Least Loaded and Round Robin algorithms were about 2μs, of Consistent Hashing with bounded loads 4μs, and of our algorithm 5μs. We found that the overhead of the load balancing algorithm is negligible in serverless computing, latencies for function execution introduce an overhead of milliseconds, while load balancing algorithms of only a few microseconds.

6 Conclusions

Serverless computing is a novel concept that could change the paradigm of software product development. To vitalize serverless computing, a load balancing algorithm that maximizes locality while minimizing load imbalance should be implemented. In this paper, we propose a novel load balancing algorithm using a table-based greedy approach, which shows better results in terms of locality, load imbalance and the performance compared to existing schemes. Additionally, our evaluations show 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 serverless platforms using various caching techniques, with the hope to provide further insight into the field of serverless computing.

References

[1] 2018 serverless community survey: huge growth in serverless. https://serverless.com/blog/2018-serverless-community-survey-huge-growth-usage.

[2] Cristina L Abad, Edwin F Boza, and Erwin Van Eyk. Package-aware scheduling of faas functions. In *Companion of the 2018 ACM/SPEC International Conference on Performance Engineering*, pages 101–106. ACM, 2018.

[3] Istemi Ekin Akkus, Ruichuan Chen, Ivica Rimac, Manuel Stein, Klaus Satzke, Andre Beck, Paarijaat Aditya, and Volker Hilt. {SAND}: Towards high-performance serverless computing. In *2018 {USENIX} Annual Technical Conference ({USENIX}{ATC} 18)*, pages 923–935, 2018.

[4] Gabriel Aumala, Edwin Boza, Luis Ortiz-Avilés, Gustavo Totoy, and Cristina Abad. Beyond load balancing: Package-aware scheduling for serverless platforms. In *2019 19th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing (CCGRID)*, pages 282–291, May 2019.

[5] Ioana Baldini, Paul Castro, Kerry Chang, Perry Cheng, Stephen Fink, Vatche Ishakian, Nick Mitchell, Vinod Muthusamy, Rodric Rabbah, Aleksander Slominski, et al. Serverless computing: Current trends and open problems. In *Research Advances in Cloud Computing*, pages 1–20. Springer, 2017.

[6] Scott Hendrickson, Stephen Sturdevant, Tyler Harter, Venkateshwaran Venkataramani, Andrea C Arpaci-Dusseau, and Remzi H Arpaci-Dusseau. Serverless computation with openlambda. In *8th {USENIX} Workshop on Hot Topics in Cloud Computing (HotCloud 16)*, 2016.

[7] Tobias Heckel Johannes Manner, Martin Endreß and Guido Wirtz. Cold start influencing factors in function as a service. In *2018 IEEE/ACM International Conference on Utility and Cloud Computing Companion (UCC Companion)*, pages 181–188, 2018.

[8] Eric Jonas, Johann Schleier-Smith, Vikram Sreekanti, Chia-Che Tsai, Anurag Khandelwal, Qifan Pu, Vaishaal Shankar, Joao Carreira, Karl Krauth, Neeraja Yadwadkar, Joseph E. Gonzalez, Raluca Ada Popa, Ion Stoica, and David A. Patterson. Cloud Programming Simplified: A Berkeley View on Serverless Computing. *arXiv e-prints*, page arXiv:1902.03383, Feb 2019.

[9] David Karger, Alex Sherman, Andy Berkheimer, Bill Bogstad, Rizwan Dhanidina, Ken Iwamoto, Brian Kim, Luke Matkins, and Yoav Yerushalmi. Web caching with consistent hashing. *Computer Networks*, 31(11-16):1203–1213, 1999.

[10] Ping-Min Lin and Alex Glikson. Mitigating Cold Starts in Serverless Platforms: A Pool-Based Approach. *arXiv e-prints*, page arXiv:1903.12221, Mar 2019.

[11] Wes Lloyd, Shruti Ramesh, Swetha Chinthalapati, Lan Ly, and Shrideep Pallickara. Serverless computing: An investigation of factors influencing microservice performance. In *2018 IEEE International Conference on Cloud Engineering (IC2E)*, pages 159–169. IEEE, 2018.

[12] Vahab Mirrokni, Mikkel Thorup, and Morteza Zadimoghaddam. Consistent hashing with bounded loads. In *Proceedings of the Twenty-Ninth Annual ACM-SIAM Symposium on Discrete Algorithms*, pages 587–604, 2018.

[13] Muhammad Anis Uddin Nasir, Gianmarco De Francisci Morales, David García-Soriano, Nicolas Kourtellis, and Marco Serafini. The power of both choices: Practical load balancing for distributed stream processing engines. In *2015 IEEE 31st International Conference on Data Engineering*, pages 137–148, April 2015.

[14] Edward Oakes, Leon Yang, Kevin Houck, Tyler Harter, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau. Pipsqueak: Lean lambdas with large libraries. In *2017 IEEE 37th International Conference on Distributed Computing Systems Workshops (ICDCSW)*, pages 395–400, June 2017.

[15] Erwin Van Eyk, Alexandru Iosup, Simon Seif, and Markus Thömmes. The spec cloud group's research vision on faas and serverless architectures. In *Proceedings of the 2nd International Workshop on Serverless Computing*, pages 1–4. ACM, 2017.

[16] Liang Wang, Mengyuan Li, Yinqian Zhang, Thomas Ristenpart, and Michael Swift. Peeking behind the curtains of serverless platforms. In *2018 {USENIX} Annual Technical Conference ({USENIX}{ATC} 18)*, pages 133–146, 2018.