A decentralized serverless orchestrator with in-memory data store

Kai-Siang Wang, Chun-Chieh Chang
University of Illinois at Urbana-Champaign
{kw37,cc132}@illinois.edu

Abstract

Serverless computing has become the dominant paradigm for building microservice applications in cloud environments due to its pay-as-you-go model and event-based triggering, which essentially reduce costs and provide an easy way to connect more and more functions as the applications grow. However, the standalone orchestrator, despite its usefulness, breaks this virtuous nature of serverless since it has to be hosted on on-demand VMs, which increases the cost for either cloud providers or customers. The decentralized orchestrator addresses this issue by completely removing the centralized node and taking advantage of the in-cloud datastore to checkpoint the current progress of the execution. Nonetheless, it is limited by the latency of the datastore and may. Therefore, the service level objective might be violated. In this work, we propose a hybrid checkpointing approach that utilizes the low-latency property from the in-memory data store to respect SLO while keeping the overall costs minimal with a critical node selection algorithm. The experimental results show that our approach can reduce the cost by up to 25%.

1 Introduction

Serverless computing has gained more and more popularity with its simple abstraction and low costs. Developers using serverless do not need to maintain the computing servers on their own and can benefit from the scalability of serverless computing while only paying for the exact compute time and memory usage [7]. The event-triggered abstraction [10] also allows developers to easily modify the applications without much effort. In addition, though the functions themselves are stateless, modern applications based on serverless computing usually adopt an in-cloud data store to persist all the intermediate data generated by functions.

While serverless platforms were initially designed to handle simple workflows such as HTTP requests, uploaded image processing, and database triggering, this paradigm has been adopted to even more complicated applications and has been proven useful. However, compositing such complex applications with the basic serverless platform is challenging [11, 12, 14, 15, 20]. Specifically, it is hard to ensure the exactly-once semantic with such a decentralized structure. As a result, cloud providers usually provide the standalone orchestrators [2, 3, 5, 6] to provide a central view of the application. The users may also host the custom orchestrators [8, 11, 12] on their own to provide further customization and optimization for their applications.

Nonetheless, both orchestrations fall short when it comes to the costs. From the cloud providers' perspectives, providing a standalone orchestrator incurs the costs of hardware and on-call engineering teams. In addition, the orchestrator may lack customization capabilities and thus may not be an option for running some specific applications. On the users' side, hosting a custom orchestrator can also become a burden due to under-utilization and does not benefit from the pay-as-you-go model. As a result, Unum [17] proposes a decentralized orchestrator where the system uses consistent data stores from cloud providers to do checkpointing and, therefore, ensure the exactly-once semantic provided by standalone orchestrators.

While decentralized orchestrations seem to be a promising solution, they may suffer from high latency due to the use of on-dist data stores. In fact, in-memory data stores have recently gained its popularity due to the increasing demands on service-level objectives (SLOs). ElasticCache [4], for example, is largely deployed as a caching layer for DynamoDB. A decentralized orchestration should take advantage of the low latency these data stores provide.

In this paper, we notice that a complex graph usually contains several fan-in patterns [2, 3, 5, 6, 11, 12] where the data store may become the bottleneck. This work utilizes in-memory data stores with on-disk data stores (e.g., S3, DynamoDB) and proposes a critical node selection algorithm to minimize the cost while respecting user-defined SLOs.

2 Background & Motivation

2.1 Decentralized orchestrators

Decentralized orchestrators have been proposed to address the issues of standalone orchestrators, including high hosting overhead and the lack of user-specific customization. The key insight of decentralized orchestrators is the use of the "already-consistent" data stores that usually exist in most clouds. For each function invocation, it keeps the result in the data store as a checkpoint. Thus, even though the function is triggered multiple times due to failures and other reasons, the function simply reads from the existing checkpoint, which ensures the downstream functions have a consistent input. To support fan-in patterns, decentralized orchestrators use a similar mechanism as checkpoints where

it stores the checkpoints of the parents of the fan-in node in a *set*. When the parent node tries to invoke the fan-in node, it first checks if the size of the set is equal to the number of fan-ins. Though proved to be viable, it inevitably incurs a higher latency when the number of fan-ins increases.

2.2 Ephemeral serverless caching

To achieve higher performance, the applications can adopt a caching layer on top of the data store. For example, in AWS, people use ElasticCache or DAX to reduce the read/write latency to a microsecond level. However, neither of them provides a consistent view when multiple nodes are accessing the same data. On the other hand, AWS also provides an in-memory durable datastore, MemoryDB, to act as a standalone data store. This provides an opportunity to improve the performance of serverless applications.

In terms of performance, while DynamoDB has, on average, 20ms latency, MemoryDB can achieve 0.6ms read latency and 4.5ms write latency, respectively [1]. This motivates us to design an in-memory decentralized orchestrator where, in a fan-in function, it must wait for multiple parents to finish, which results in reading checkpoint data multiple times.

Many studies [13, 16, 19] aim to utilize the cache on the VMs on which the serverless functions are running to improve the application performance. They adopt some mechanisms to periodically warm up the containers to make sure the cache will be kept on the containers for some time. In fact, we think these approaches are orthogonal to our approach and might be able to be integrated to further minimize the cost.

3 Proposed solutions

In this section, we describe the details of our design and implementation, highlighting how it minimizes costs while respecting user-defined SLOs.

Problem definitions. To minimize the end-to-end latency, the insight is to replace the existing checkpointing data stores along with the critical path with in-memory data stores. In this work, we assume there are only two types of data stores, one on disk and one in memory. The on-disk data store has an access latency L_{rd} and L_{wd} for read and write operations, respectively. Similarly, the in-memory data store's latency is denoted as L_{rm} and L_{wm}. The cost per GB of the data stores is defined as C_d and C_m. Given an application $G(V, E)$ where V are the functions and E are the dependencies among these functions, we can calculate the end-to-end latency of the application L_{e2e} and the total checkpointing time T_{ckpt}. A user-defined SLO, denoted as S, represents the ratio of $(L_{e2e} - T_{ckpt})/L_{e2e}$. Our goal is to find a plan P where each layer can decide which data store to do the checkpointing with a minimized cost while respecting the SLO.

Assumption: We assume we know the execution times of all serverless functions. In the real world, we can profile the application to obtain this information.

Critical nodes selection algorithm. We first analyze the application graph at compile time, calculate the end-to-end execution time, and find the critical path, assuming we are using DynamoDB for checkpointing. Largely inspired by this work [18], we calculate the upward rank and the downward rank for each node, denoted as $rank_u$ and $rank_d$, as follows:

$$rank_u(n_i) = \overline{w_i} \times deg_{in}(i) + \max_{n_j \in succ(n_i)} (\overline{c_{i,j}} + rank_u(n_j)) \quad (1)$$

$$rank_d(n_i) = \max_{n_j \in pred(n_i)} \{rank_d(n_j) + \overline{w_j} + \overline{c_{j,i}}\} \quad (2)$$

where $succ(n_i)$ is the set of immediate successors of task n_i, $\overline{c_{i,j}}$ is the average communication cost of edge (i, j), and $\overline{w_i}$ is the profiled execution time of task ni. Since the rank is computed recursively by traversing the graph upward, starting from the end node, it is called upward rank. For the end node n_{end}, the upward rank value is equal to

$$rank_u(n_{end}) = \overline{w_{end}} \quad (3)$$

The priority of each node is defined as $rank_u + rank_d$. The insight here is to approximate the distances of a node to the start node and the end node. If a node has a larger sum of distances, it is on the critical path.

After that, we replace the data store of some critical nodes with MemoryDB. To minimize the cost, we try to minimize the usage of MemoryDB. We design a dynamic programming algorithm to calculate the minimal number of critical nodes that need to be replaced, as shown in Algorithm 1. In lines 12 to 28, we divide the problem into several subproblems. Suppose we already obtain the solution of the subproblem; we then calculate, by excluding different nodes in *nodes*, the solution that not only satisfies the user-defined SLO but also has a minimal cost. Recursively, we can get the optimal selection of the critical nodes, which gives us minimal cost while respecting the SLO.

4 Preliminary results

In this section, we present some preliminary results to justify the concept of hybrid data stores for decentralized orchestrations. We implemented our solution based on the simulation. We conduct the following experiments with an application containing 10 functions and 15 dependencies.

4.1 End-to-end latency

We first evaluate our solution on end-to-end latency compared with the baseline solution, where it first sorts all the nodes based on their numbers of in-degrees and then iteratively sets the node with the highest in-degrees to use the in-memory data store. The result is shown in Figure **??**. Since our goal is to minimize the cost given an SLO, it is not fair to compare the end-to-end latency directly. Instead, we compare **latency-per-cost** (Latency/Cost), which represents the

Algorithm 1 Critical Node Selection Algorithm

Require: $G(v, e), slo$
$critical\_nodes \leftarrow getCriticalNode(G)$
$A \leftarrow table[node\_ids]$
$s \leftarrow selectNodes(critical\_nodes, slo, A)$
for $node \in s$ **do**
  $node.use\_memory\_store \leftarrow True$
end for
procedure SELECTNODES($nodes, slo, A$)
  $key \leftarrow node\_ids$
  if $key \in A$ **then**
    return $A[key]$
  end if
  for $node \in nodes$ **do**
    $subnodes \leftarrow nodes\_exclude\_node$
    $s, L_{e2e}, T_{ckpt}, c = selectNodes(subnodes, slo, A)$
    $dl \leftarrow L_{wd} + L_{rd} \times deg_{in}(node)$
    $new\_L_{e2e} \leftarrow L_{e2e} + dl + node.exec\_time$
    $new\_T_{ckpt} \leftarrow T_{ckpt} + dl$
    if $(new\_L_{e2e} - new\_T_{ckpt})/new\_L_{e2e} < slo$ **then**
      $new\_c \leftarrow c + C_m * node\_space$
      $ml \leftarrow L_{wm} + L_{rm} \times deg_{in}(node)$
      $new\_L_{e2e} \leftarrow L_{e2e} + ml + node.exec\_time$
      $new\_T_{ckpt} \leftarrow T_{ckpt} + ml$
      $s.insert(node)$
    end if
    $minarg_c(new\_c, new\_L_{e2e}, new\_T_{ckpt}, s)$
  end for
  $A[key] = s, new\_L_{e2e}, new\_T_{ckpt}, c$
  return $s, new\_L_{e2e}, new\_T_{ckpt}, c$
end procedure

latency over the same additional cost. Table 1 shows that our approach is more cost-efficient than the baseline.

Algo.	Latency/Cost
Baseline	0.32 s/$
Hybrid	0.24 s/$

Table 1. Latency to cost ratio comparison with baseline.

4.2 Cost Analysis

We evaluate our solution with different user-defined SLOs. To fulfill a higher SLO, we need a higher additional cost (i.e., the cost of using MemoryDB) and the corresponding end-to-end latency reduction, as depicted in Figure 1. Compared with the baseline approach, as shown in Figure 2, we can also see that when the user requires a higher SLO, our solution can reduce the additional costs by up to 25%.

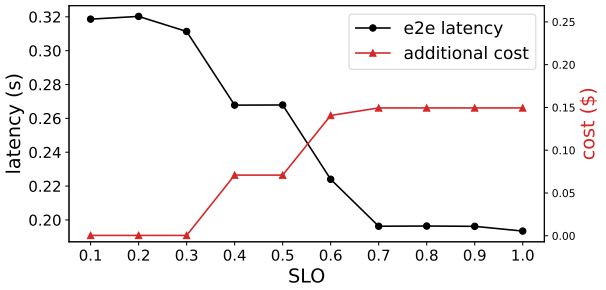

Figure 1. Cost analysis of hybrid data stores approach.

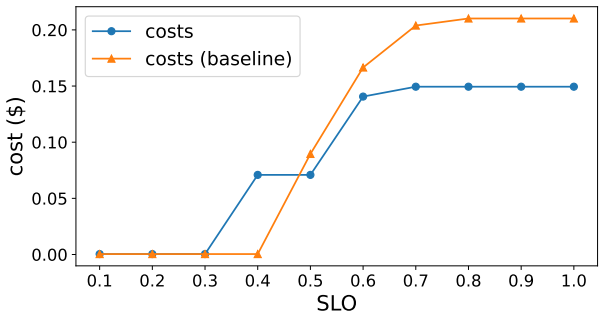

Figure 2. Cost comparison of critical nodes selection algorithm with baseline.

5 Challenges

There are some arguments [9] saying that MemoryDB might be the worst service ever due to its extremely high cost. As a result, it is unclear if the latency reduction will be beneficial enough. In fact, since now MemoryDB still has to be run on the on-demand nodes. In this work, we do not consider the hosting costs which are required in the real-world situation. However, there are several works [13, 16, 19] trying to utilize the internal cache of the serverless function and might be possible to replace MemoryDB in this work.

6 Conclusion

This paper presented a hybrid checkpointing approach for the decentralized orchestrator. At its core, the critical node selection algorithm splits the execution graph into multiple layers and decide the type of data store of each layer by a dynamic programming algorithm to minimize the cost.

7 Metadata

The presentation of the project can be found at:

https://go.cs.illinois.edu/CS523Finalreport

The code/data of the project can be found at:

https://github.com/kwang1012/unum

References

[1] Amazon memorydb latency. https://aws.amazon.com/blogs/database/measuring-database-performance-of-amazon-memorydb-for-redis/.

[2] Aws step functions. https://docs.aws.amazon.com/step-functions/latest/dg/limits-overview.html.

[3] Azure durable functions. https://docs.microsoft.com/en-us/azure/azure-functions/durable/durable-functions-overview?tabs=csharp.

[4] Elasticcache. https://aws.amazon.com/elasticache/.

[5] Google cloud composer (gcc). https://cloud.google.com/composer.

[6] Google workflows. https://cloud.google.com/workflows.

[7] Lambda pricing. https://docs.aws.amazon.com/whitepapers/latest/how-aws-pricing-works/lambda.html.

[8] Temporal platform. https://docs.temporal.io/.

[9] 'worst' aws service ever? cloud giant introduces redis-compatible memorydb – to mixed response. https://www.theregister.com/2021/08/23/aws_memorydb_reaction/.

[10] Burckhardt, S., Chandramouli, B., Gillum, C., Justo, D., Kallas, K., McMahon, C., Meiklejohn, C. S., and Zhu, X. Netherite: efficient execution of serverless workflows. *Proc. VLDB Endow. 15*, 8 (apr 2022), 1591–1604.

[11] Fouladi, S., Romero, F., Iter, D., Li, Q., Chatterjee, S., Kozyrakis, C., Zaharia, M., and Winstein, K. From laptop to lambda: Outsourcing everyday jobs to thousands of transient functional containers. In *2019 USENIX Annual Technical Conference (USENIX ATC 19)* (Renton, WA, July 2019), USENIX Association, pp. 475–488.

[12] Fouladi, S., Wahby, R. S., Shacklett, B., Balasubramaniam, K. V., Zeng, W., Bhalerao, R., Sivaraman, A., Porter, G., and Winstein, K. Encoding, fast and slow: Low-Latency video processing using thousands of tiny threads. In *14th USENIX Symposium on Networked Systems Design and Implementation (NSDI 17)* (Boston, MA, Mar. 2017), USENIX Association, pp. 363–376.

[13] Ghosh, B. C., Addya, S. K., Somy, N. B., Nath, S. B., Chakraborty, S., and Ghosh, S. K. Caching techniques to improve latency in serverless architectures. In *2020 International Conference on COMmunication Systems NETworkS (COMSNETS)* (2020), pp. 666–669.

[14] Jia, Z., and Witchel, E. Boki: Stateful serverless computing with shared logs. In *Proceedings of the ACM SIGOPS 28th Symposium on Operating Systems Principles* (New York, NY, USA, 2021), SOSP '21, Association for Computing Machinery, p. 691–707.

[15] Jonas, E., Pu, Q., Venkataraman, S., Stoica, I., and Recht, B. Occupy the cloud: Distributed computing for the 99

[16] Klimovic, A., Wang, Y., Stuedi, P., Trivedi, A., Pfefferle, J., and Kozyrakis, C. Pocket: Elastic ephemeral storage for serverless analytics. In *13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 18)* (Carlsbad, CA, Oct. 2018), USENIX Association, pp. 427–444.

[17] Liu, D. H., Levy, A., Noghabi, S., and Burckhardt, S. Doing more with less: Orchestrating serverless applications without an orchestrator. In *20th USENIX Symposium on Networked Systems Design and Implementation (NSDI 23)* (Boston, MA, Apr. 2023), USENIX Association, pp. 1505–1519.

[18] Topcuoglu, H., Hariri, S., and Wu, M.-Y. Performance-effective and low-complexity task scheduling for heterogeneous computing. *IEEE Transactions on Parallel and Distributed Systems 13*, 3 (March 2002), 260–274.

[19] Wang, A., Zhang, J., Ma, X., Anwar, A., Rupprecht, L., Skourtis, D., Tarasov, V., Yan, F., and Cheng, Y. InfiniCache: Exploiting ephemeral serverless functions to build a Cost-Effective memory cache. In *18th USENIX Conference on File and Storage Technologies (FAST 20)* (Santa Clara, CA, Feb. 2020), USENIX Association, pp. 267–281.

[20] Zhang, H., Cardoza, A., Chen, P. B., Angel, S., and Liu, V. Fault-tolerant and transactional stateful serverless workflows. In *Proceedings of the USENIX Symposium on Operating Systems Design and Implementation (OSDI)* (November 2020).