CacheFlow: Enhancing Data Flow Efficiency in Serverless Computing by Local Caching

Yi-Syuan Ke*, Zhan-Wei Wu*, Chih-Tai Tsai*, Sao-Hsuan Lin* and Jerry Chou†
Department of Computing Science, National Tsing Hua University, Hsinchu, Taiwan

Keywords: Cloud Computing, Function as a Service, Serverless Computing, Caching.

Abstract: In Serverless workflows, it is common for various functions to share duplicate files or for the output of one function to be the input for the following function. Currently, storing these files in remote storage is common, and transferring a large amount of data over the network can be inefficient and time-consuming. However, the state of the art has not yet optimized this aspect, resulting in time wastage. In this paper, we present an improved data transfer solution that reduced data transfer time by up to 82.55% in our experiment. This improvement is achieved by replacing the time spent on remote network access with local disk access time. To reduce the data transfer route, we implement per-node caching, utilizing disk storage as a local cache space.

1 INTRODUCTION

Serverless computing has become increasingly popular in cloud computing. It empowers developers to execute specific functions or code snippets in response to events without the burden of managing servers or infrastructure. This approach offers users the advantage of liberating themselves from the complexities of infrastructure management. They can simply deploy their functions to the cloud platform of their choice. Serverless events are triggered by HTTP requests, data changes, or scheduled tasks. Also, the auto-scaling mechanism will automatically scale up or down resources as needed.

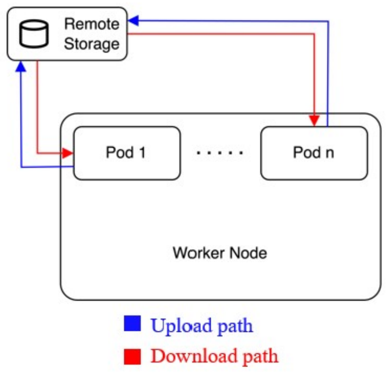

Figure 1: Original Data transfer route.

FaaS, short for Function-as-a-Service, focuses on the event-driven computing paradigm. Users only need to focus on the function code they deploy onto the cloud platform. Several established FaaS platforms, such as AWS Lambda (Amazon, 2014), Azure Functions (Microsoft, 2010), and Google Cloud Functions (Google, 2011), offer users the advantage of a Pay-as-You-Go model. With FaaS, users are only charged for the actual compute time consumed by your functions. There are no upfront expenses or ongoing costs associated with maintaining servers that sit idle when their functions are not actively running.

In the FaaS paradigm, functions typically follow a stateless approach, meaning they do not retain data between executions. As a result, any necessary state or data management tasks are typically offloaded to

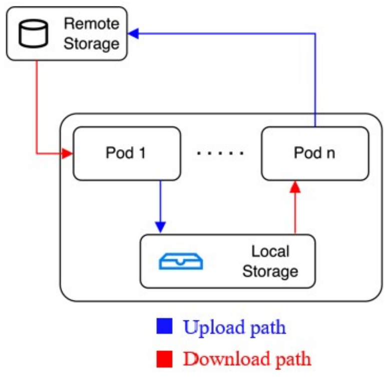

Figure 2: Optimized Data transfer route.

*Contributed equally to this work.
†corresponding author.

Ke, Y., Wu, Z., Tsai, C., Lin, S. and Chou, J.
CacheFlow: Enhancing Data Flow Efficiency in Serverless Computing by Local Caching.
DOI: 10.5220/0012688600003711
Paper published under CC license (CC BY-NC-ND 4.0)
In *Proceedings of the 14th International Conference on Cloud Computing and Services Science (CLOSER 2024)*, pages 224-230
ISBN: 978-989-758-701-6; ISSN: 2184-5042
Proceedings Copyright © 2024 by SCITEPRESS – Science and Technology Publications, Lda.

external services, such as databases or storage systems. Consequently, for workflows that generate intermediate state or data, the common practice is to utilize remote storage solutions like AWS S3 (Amazon, 2006) or MinIO (MINIO, 2016) to store this intermediate data (Ana Klimovic, 2018).

We show the above behavior in Figure 1. We define the data download route, the route of downloading data to runtime from remote storage, and the data upload route, the route of uploading data to remote storage from runtime. The data download/upload time represents the execution time of the defined routes.

However, this approach can lead to several challenges. Firstly, it incurs a notable amount of time dedicated to data transfer in both data downloading and uploading operations since it uses the network as the transmission method. Additionally, this approach consumes storage space, in cloud computing usually cloud storage for maintaining the intermediate data, which may grow significantly over time, the cost of renting a cloud storage is inescapable.

In this scenario, both remote database I/O and network condition become potential bottlenecks (Eric Jonas, 2017; SINGHVI A, 2017). This results in prolonged data transfer times and an overall increase in latency. Furthermore, as the number of pods or functions increases, the demand on the network infrastructure also escalates, leading to a substantial surge in network requirements.

The primary focus of our research is to optimize data transfer efficiency through the implementation of data caching mechanisms. To achieve this, we leverage the local disk storage available on each node to establish a local storage. We incorporate the Least Recently Used (LRU) caching algorithm to effectively cache intermediate data generated throughout the workflow processes in the distributed system.

In Figure 2, we illustrate the new data transfer route after our optimization. This route is not only significantly quicker than the previous one but also involves the replacement of a portion of the time spent on remote network access with local disk access. Note that this route is in the case that desired data is cached in the same node. For data cached in other nodes, we have components to realize the caching mechanism as well, we will introduce it in the latter chapters.

Our experimental result demonstrates a remarkable reduction in transfer times, with a substantial at most decrease of 82.55% in our experiment. We also show that the data transfer speedup is positively correlated with the disk/network speed ratio, which means the optimization will be better as the disk performance is enhanced or the network is congested.

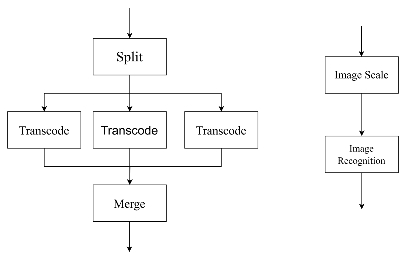

Figure 3: Data transfer route for current workflows.

2 SERVERLESS WORKFLOW BACKGROUND

One of the behaviors of serverless computing is using remote storage as centralized storage. We have found that for workflows requiring remote storage, existing serverless platforms spend a significant amount of time on data transfer, which is the total time of downloading and uploading data. In this kind of workflow, both database I/O and network become bottlenecks, resulting in extended data transfer times and overall increased latency.

We benchmark two real-world data-driven workflows, image-processing and video-processing. These workflows are common in serverless benchmarks. They share a common characteristic in serverless workflow: intermediate data is only used between functions, and only the initial and final data need to be in remote storage. Figure 3 shows their workflow in DAGs.

2.1 Image Processing

This workflow is a classic image recognition workflow. It first downloads data from the remote storage, scales it to a specific size, uses a pre-trained model to get the result, and uploads it back to the remote storage.

2.2 Video Processing

This use case comes from Alibaba Cloud. It first splits video data into small clips, transcodes them parallelly, and merges them together. The initial data and results are downloaded/uploaded from/to the remote storage (Alibaba, 2021).

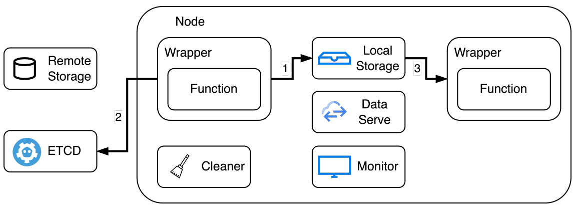

Figure 4: Cache architecture when data exist in local storage.

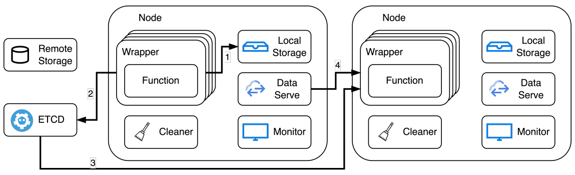

Figure 5: Cache architecture when data exist in other nodes.

3 ARCHITECTURE

In this section, we introduce our innovative two-speedup path architecture designed to optimize data retrieval efficiency in a serverless environment.

Figure 4 depicts the initial acceleration pathway, activated when essential data resides in local storage. **Path 1** optimizes the data retrieval process by seamlessly copying it from local storage, subsequently storing the cached metadata in ETCD through **path 2**. Ultimately, users can leverage our cache mechanism via **path 3** to avoid the associated overhead of transferring data to remote storage. This function facilitates direct access to data in local storage, eliminating the necessity for remote downloads.

In cases where the data function required resides on a different node within the cluster, an alternative flow is initiated, as depicted in Figure 5. When local storage fails to yield the required data in **path 1**, the system queries ETCD via **path 2**, which we use to store the information of the data and its position(node IP) to ascertain the node storing the data. Subsequently, a data retrieval request is dispatched to the identified node by the Data Server(**path 3 and 4**), ensuring efficient data access across distributed environments.

The success of our architecture hinges on the seamless collaboration of multiple components, ensuring optimized data retrieval and storage resource utilization.

3.1 Monitor

The Monitor component tracks changes (creation, write, removal) in cached data and records these events within a data structure. When storage space approaches a predefined threshold, it initiates the removal of the least recently used cached data. This component plays a crucial role in efficiently managing storage space usage.

3.2 Cleaner

The Cleaner component serves as the secondary cache eviction mechanism, periodically removing outdated cached data from the disk. Its purpose is to prevent outdated data from persisting in local storage, thus safeguarding node performance.

3.3 Wrapper

The Wrapper provides portability, acting as a bridge to the MinIO API, allowing users to seamlessly integrate our caching mechanism with minimal modifications while ensuring data integrity. In workflows, each pod will have its designated space for storing data, preventing multiple pods from accessing the same data to maintain **cache coherence**. It will use the hash value created when the data is cached to verify the **cache consistency**. It also tells ETCD the location of the data, for functions in other nodes to query to Data Serve in this node.

3.4 Data Serve

The Data Serve component facilitates data access for pods residing on other nodes within the cluster. In scenarios where data is cached in local storage but required by pods deployed on different nodes, this component serves as the intermediary. Pods query ETCD to discover the IP address of the Data Serve, enabling them to retrieve data via TCP connections.

4 IMPLEMENTATION

4.1 Data Event Monitoring

The current Linux Inotify subsystem lacks support for recursive watches within sub-directories. Consequently, we have developed our own file notification system to monitor newly created directories automatically. Each read, write, and other file-related operation triggers notifications to our Monitor, allowing it to efficiently update the least recently used (LRU) queue with minimal overhead.

4.2 Shared Storage Space Management

To optimize the caching hit rate, it is essential to devise a mechanism that ensures the retention of frequently used data in our shared storage while removing items that are less likely to be accessed.

```
Input: $filepath$
Result: Update the least recently used queue
if $LRUqueue.exist(filepath)$ then
    LRUqueue.updatePosition($filepath$)
else
    LRUqueue.addToFront($filepath$)
    if $LRUqueue.size > capacity$ then
        LRUqueue.removeLeastRecentlyUsed()
    end
end
```

Algorithm 1: Least Recently Used.

Our primary cache eviction method employs the Least Recently Used (LRU) algorithm, managed with a doubly linked list. This approach provides amortized $O(1)$ complexity for push and pop operations, as well as efficient removal of deleted files in amortized $O(1)$ time by hashing and locating the file's position in the linked list. The capacity here is to reserve space to prevent a large amount of data from being synchronously stored in shared storage. For secondary cache eviction, the Cleaner component runs periodically, removing outdated data once a day. This proactive approach ensures that files not tracked by the Monitor can also be efficiently cleaned, maintaining optimal storage resource utilization.

4.3 Distrubuted Data

There is another challenge when caching data. In order to achieve load balance, Kubernetes will assign requests to different nodes through its scheduler. So, if two functions in the same workflow are located on different nodes, caching data in their local storage can lead to a file-missing problem. To address this issue, we have implemented our data server to transfer data over the network and utilize ETCD to store file locations on each node. However, given that the data is on different nodes, we need to ensure its integrity. Therefore, we use a hash algorithm to assist in verifying the correctness of the file.

This algorithm verifies the integrity of a file by comparing its current hash value with the correct hash value stored in a designated file. The correct hash value is calculated and recorded when the file is initially cached in our local storage. If the hash values do not match during the verification process, we will rerun to confirm the accuracy of the file.

```
Input:
  $correct\_hash\_file, file\_path, hash\_value$
Output: $IsDataCorrupted(Boolean)$
  $calculated\_hash \leftarrow$
  CALCULATE_HASH($file\_path, hash$)
with open($correct\_hash\_file$, "r") as $file$:
  $hash\_value \leftarrow file.read()$
if $hash\_value == calculated\_hash$ then
    $verifySuccess \leftarrow$ True
else
    $verifySuccess \leftarrow$ False
end
Return: $verifySuccess$
```

Algorithm 2: Hash Verification.

4.4 Uniform API

We hope users can utilize our implementation without modifying their code. Therefore, we have abstracted our implementation into the MinIO library. This allows users to employ the same function code to achieve our caching mechanism. Another advantage of using the library is that it helps us avoid high concurrency issues compared to centrally deciding whether to cache or not.

In our current implementation, pods independently determine whether to cache data based on local storage usage. This approach helps us avoid large synchronous requests to a single node.

5 EVALUATION

In the following sections, we will comprehensively present the performance of our work through real-world applications, examining its efficacy across both single-node and multi-node environments. Our primary objective is to assess its practical effectiveness in these diverse contexts, providing a thorough analysis of its capabilities and demonstrating its adaptability to varied computing scenarios.

We employed two applications, namely image processing and video processing, to conduct benchmarking. Our experiments were executed on a Kubernetes cluster using Kubeadm on multiple virtual machines for the infrastructure, and table 1 shows the configuration of our virtual machines.

In the image processing application, we integrated two distinct functions: image scaling (IS) and image recognition (IR)—within both single-node and multi-node environments. As illustrated in Figure 6 and Figure 7, our optimizations resulted in substantial reduc-

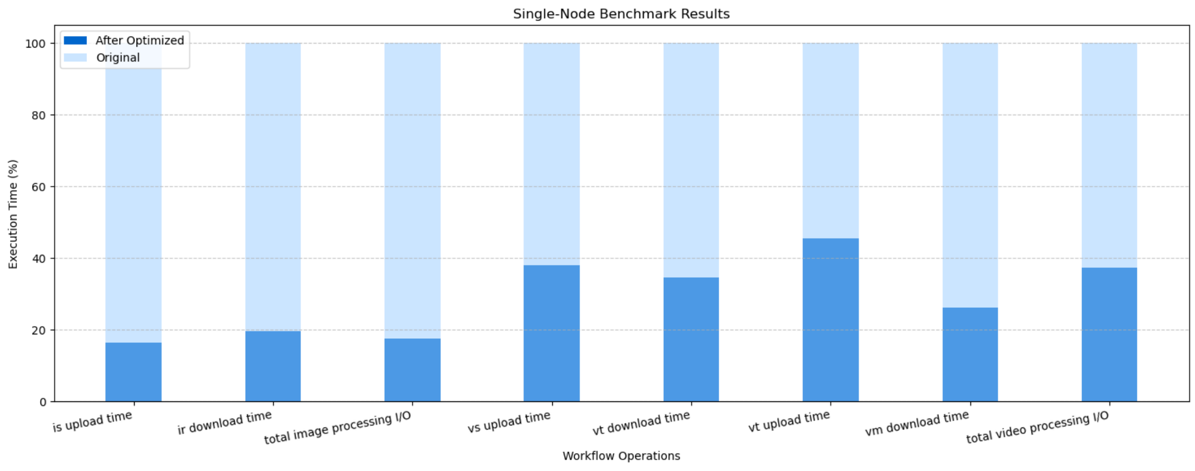

Figure 6: Single Node benchmark result.

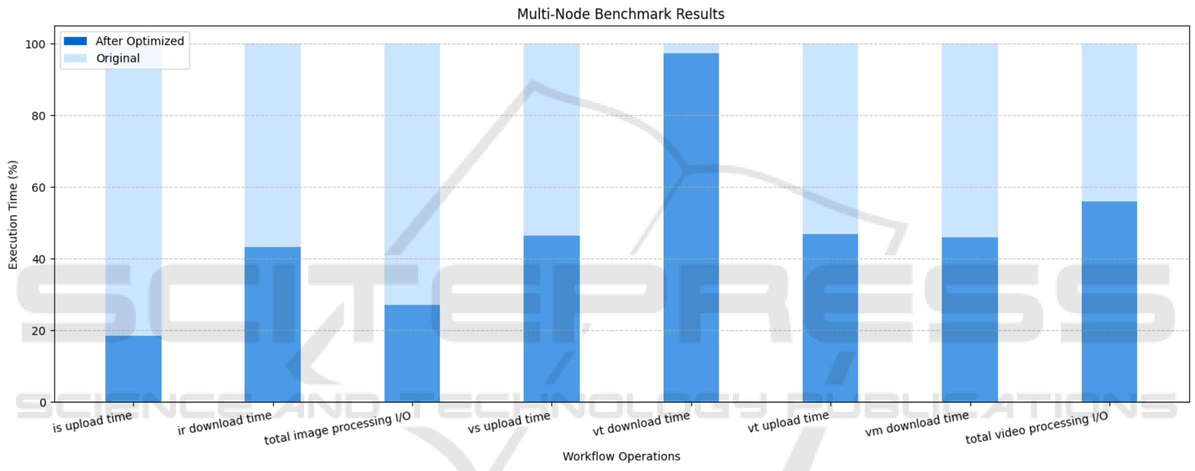

Figure 7: Multi Node benchmark result.

tions in both the upload time for IS and the download time for IR. Notably, these improvements were achieved due to the successful implementation of a cache mechanism.

It is noteworthy that both single-node and multi-node configurations exhibit great performance, with single-node slightly faster than multi-node. This superiority can be attributed to the absence of inter-node data transfer and associated mechanisms in the single-node setup. The overall optimization of data transfer time across the entire function chain has led to an impressive 82.55% reduction.

Figure 6 and Figure 7 also illustrate the video processing application, consisting of three key functions: video splitting (VS), video transcoding (VT), and video merging (VM). As depicted in the figures, almost all data transfer times exhibited significant reductions, with the most notable optimization reaching up to 3.84x faster data transfer time. As a result, the total data transfer time for the entire function chain decreased by 62.64%.

The illustrated Figure 8 presents the correlation between acceleration effects and the disk/network speed ratio. However, it is imperative to emphasize that the acceleration, despite exhibiting an upward trend, does not follow a linear proportional relationship. This deviation arises due to the diminishing execution time, leading to a more pronounced proportion of overhead.

6 CONCLUSION

In this paper, we introduce a data transfer solution designed to reduce overall data transfer times significantly. We achieve this by implementing a data caching mechanism that effectively transforms network access times into disk access times, thereby enhancing the efficiency of data transmission. Our

Table 1: VM Configuration.

	VM Configuration
Hardware	CPU: Intel(R) Xeon(R) CPU E5-2675 v3 @ 1.80GHz Cores: 16 Memory: 16GB Disk: NFS, size=150 GB, speed=573 MB/s Internet speed: download: 85MB/s, upload: 75MB/s
Software	OS: Ubuntu 20.04.3 LTS (GNU/Linux 5.4.0-163-generic x86_64) MinIO version: RELEASE.2023-04-28T18-11-17Z Docker version: 20.10.24 kind version: 0.17.0 knative version: v1.8.1 etcd version: 3.4.27 container runtime: python:3.10-slim

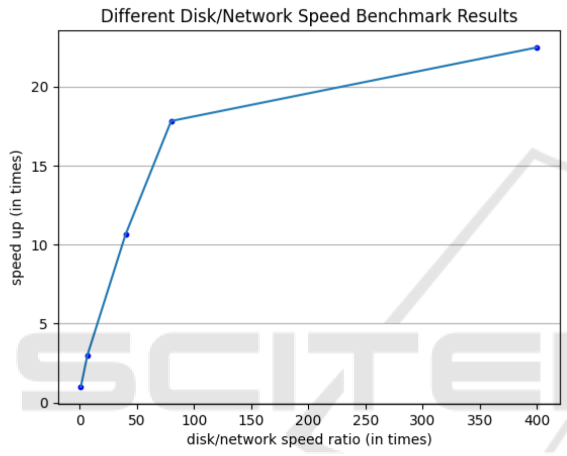

Figure 8: Relation between speedup and disk/network ratio.

experiments demonstrate a remarkable reduction in transfer times, with a substantial at most decrease of 82.55%.

7 RELATED WORK

The concept of caching has been used in various aspects and applications in FaaS. **FaaSFlow** (Li, 2022) enhances traditional serverless master-side workflow scheduling by introducing a worker-side engine paired with a per-node local cache. They use Reddis, an in-memory key-value database, as their local storage, while we choose disk as our cache storage. The benefit of using a disk is we will have larger space to cache more data since our target workflows have large data transmission. Plus, the data we can store on disk is more flexible compared to key-value pairs. **SOCK** (Edward Oakes, 2018) addresses slow Lambda startup by identifying two common causes: container initialization and package dependencies. It optimizes startup time with a package-aware caching system to reduce redundant package imports. **FaasCache** (Alexander Furest, 2021) consider keep-alive is analogous to caching. By adopting concepts such as reuse distances and hit-ratio curves, they present caching-based keep-alive and resource provisioning policies to reduce the cold-start overhead. **Faa$T** (Francisco Romero, 2021) point out that recent caching work disregards the widely different characteristics of FaaS applications. They present a transparent auto-scaling distributed cache that scales and pre-warms based on computing demands and data access patterns. **Tetris** (Jie Li, 2022) aims to solve extensive memory usage of serverless inferences through a specified serverless inference platform. The platform shares tensors between pods through memory mapping on a shared tensor storage.

REFERENCES

Alexander Furest, P. S. (2021). Faascache: keeping serverless computing alive with greedy-dual caching. In *ASPLOS '21 Proceedings of the 26th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*.

Alibaba (2021). Use ffmpeg in function compute to process audio and video files in function compute. In *https://www.alibabacloud.com/help/doc-detail/146712.htm?spm=a2c63.l28256.b99.313.5c293c94dPLJV1*.

Amazon (2006). Aws s3. In *https://aws.amazon.com/tw/s3/*.

Amazon (2014). Aws lambda. In *https://aws.amazon.com/tw/lambda/*.

Ana Klimovic, e. a. (2018). Understanding ephemeral storage for serverless analytics. In *USENIX'18 Annual Technical Conference*.

Edward Oakes, e. a. (2018). Sock: Rapid task provisioning with serverless-optimized containers. In *USENIX'18 Annual Technical Conference*.

Eric Jonas, e. a. (2017). Occupy the cloud: distributed computing for the 99%. In *SOCC'17 In Proceedings of the 2017 Symposium on Cloud Computing*.

Francisco Romero, e. a. (2021). Faa$t: A transparent auto-scaling cache for serverless applications. In *SoCC '21 Proceedings of the ACM Symposium on Cloud Computing*.

Google (2011). Google cloud function. In *https://cloud.google.com/functions*.

Jie Li, e. a. (2022). Tetris: Memory-efficient serverless inference through tensor sharing. In *2022 USENIX Annual Technical Conference (USENIX ATC 22)*.

Li, Zijun, e. a. (2022). Faasflow: Enable efficient workflow execution for function-as-a-service. In *ASPLOS '22 Proceedings of the 27th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*.

Microsoft (2010). Microsoft azure. In *https://azure.microsoft.com/zh-tw*.

MINIO (2016). In *https://min.io/*.

SINGHVI A, e. a. (2017). Granular computing and network intensive applications: Friends or foes? In *In Proc. of the 16th ACM Workshop on Hot Topics in Networks, HotNetsXVI*.