CGHit: A Content-oriented Generative-hit Framework for Content Delivery Networks

1st Peng Wang
WNLO, HUST
Wuhan, China
wangpeng@hust.edu.cn

2nd Yu Liu✉
SCST, HUST
Wuhan, China
liu_yu@hust.edu.cn

3th Kai Han
SCST, HUST
Wuhan, China
kylehan@hust.edu.cn

4th Ziqi Liu
SCST, HUST
Wuhan, China
liuziqi@hust.edu.cn

5rd Ke Liu
WNLO, HUST
Wuhan, China
liu_ke@hust.edu.cn

6th Mingyang Wang
SCST, HUST
Wuhan, China
wangmy@hust.edu.cn

7th Ke Zhou
WNLO, HUST
Wuhan, China
zhke@hust.edu.cn

8th Zhihai Huang
Tencent Technology Co., Ltd.
Shenzhen, China
tommyhuang@tencent.com

2024 International Conference on Networking, Architecture and Storage (NAS) | 979-8-3315-2050-2/24/$31.00 ©2024 IEEE | DOI: 10.1109/NAS63802.2024.10781374

***Abstract*—The service provided by content delivery networks (CDNs) may overlook content locality, leaving the potential to improve performance. In this study, we explore the feasibility of leveraging generated data as a replacement for fetching data in missing scenarios based on content locality. Due to sufficient local computing resources and reliable generation efficiency, we propose a content-oriented generative-hit framework (CGHit) for CDNs. CGHit utilizes idle computing resources on edge nodes to generate requested data based on similar or related cached data, achieving hits. Extensive experiments in a real-world system demonstrate that CGHit reduces the average access latency by half. In addition, experiments conducted on a simulator confirm that CGHit can enhance current caching algorithms, leading to lower latency and reduced bandwidth usage.**

***Index Terms*—generative-hit, content locality, content delivery network**

I. Introduction

Users can access data stored in Content Delivery Networks (CDNs) to enjoy high-quality services with low latency. However, if the requested data is unavailable on CDNs, it will be fetched from a remote data center, losing benefits on latency. To maintain these benefits, classic caching algorithms [16], [20] are widely implemented in CDNs to maximize the caching of requested data based on data locality [15]. Nevertheless, data locality may lose its magic as users prioritize content over specific data in some contexts. For example, users are willing to change the video resolution from high to low when encountering buffering, which enables smoother video playback. This behavior may lead to a situation where CDNs cache videos with the same content but different resolutions, straining the CDNs' capacity to accommodate other data. Therefore, it is necessary to consider content locality for the caching scheme of CDNs.

Content locality refers to the fact that many data items in storage share similar or even the same content [11], which has been leveraged in the design of backup storage and data deduplication strategies [9], [10]. CDNs can also follow content locality to generate missing data based on cached data with similar or related content rather than fetching it. We refer to this specific context as a "pseudo-missing" scenario and propose a "generative-hit" scheme in this scenario. There are two reasons for adopting this scheme.

TABLE I
The latency monitored from *Tencent* CDNs.

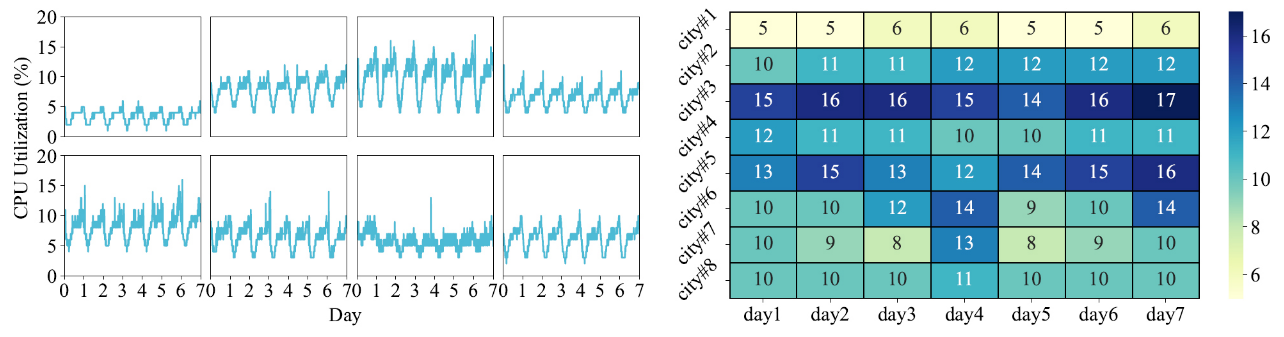
	Cache Hit	Cache Missing
P99 Latency (ms)	9.70	218.06
P99.9 Latency (ms)	15.31	283.71
Average Latency (ms)	1.90	231.07

Reason ❶: Generating data is faster than fetching data in some scenarios, and CDNs have sufficient computing resources to support data generation. The advancement of technology for generating data, especially Large Language Models (LLMs) [4], [18], has made it possible to control data generation precision and efficiency in most scenarios, typically within a range of 100 ms [14]. By comparing the missing time displayed in Table I, it can be concluded that the latency on CDNs can be reduced by returning locally generated data. Moreover, As shown in Fig. 1(a), the CPU utilization basically shows a daily cycle trend. Fig. 1(b) counts the peak CPU utilization on each day in the cycle shown in Fig. 1(a). There are idle computing resources on CDNs. Therefore, generating data locally is feasible. The latency of fetching from data centers can be reduced by delivering locally generated data.

Reason ❷: The generative-hit scheme can effectively enhance the content-carrying capacity of CDNs. Content redundancy is often observed in CDNs. As shown in Fig. 2, we presented the distribution of data with identical content but varying specifications or formats on two CDNs, *i.e.*, Case A [25] and Case B [26]. Generative-hit can generate data based on content, which in turn reduces content redundancy to a reasonable level. This leads to an increase in cache utilization in terms of content and improves the quality of content-oriented services in CDNs.

Based on the above reasons, we propose CGHit to implement generative hit in CDNs and deliver the generated data

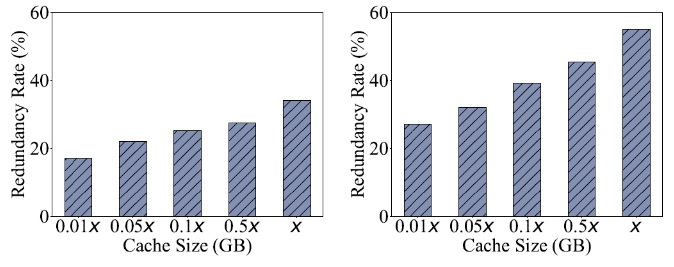
Authorized licensed use limited to: Huazhong University of Science and Technology. Downloaded on April 15,2025 at 12:04:00 UTC from IEEE Xplore. Restrictions apply.



(a) CPU utilization variation during a week

(b) Peak CPU Utilization

Fig. 1. The CPU utilization variation on different edge nodes during a week. City#1-8 are the names of the cities where edge nodes are located. The numbers in the (b) heatmap represent the peak CPU utilization for the day.



(a) Case A

(b) Case B

Fig. 2. Content redundancy rates of two CDN instances. $x = 512$GB in Case A and $x = 2048$GB in Case B.

to the user. To prevent content redundancy, `CGHit` prohibits the storage of generated data in CDNs. However, it may result in frequent generative operations and burdensome workloads. To prevent the above case from occurring, we propose a learning model that predicts subsequent requests and fetches the requested data asynchronously.

Our contributions can be summarized as follows:

① Based on content locality, we propose a generative-hit framework to deliver generated data using cached data and computing resources on the edge nodes of CDNs.

② Our prototype system has been deployed at *Tencent* [1]. The results show that the average access latency decreases by 56.04% and the bandwidth used for fetching data (*i.e.*, the back-to-COS bandwidth) reduces by 60.28%.

③ Experimental results on the simulator show that running caching algorithms on `CGHit` can further reduce the average access latency, back-to-COS bandwidth, and content redundancy.

④ We offer a new idea and solution to improve the service quality of CDNs using data generation. It will open the window for the application of large language models, which are good at generating data, on CDNs.

[1]https://cloud.tencent.com/product/cdn

II. Observations and Motivation

Observation ❶ : There are idle computing resources on CDNs. We have surveyed CDN providers such as *Akamai*[2], *Google*[3], *Alibaba*[4], *Tencent*, *etc.*, and find that computing resources on CDNs are underutilized. Fig. 1 shows the average CPU utilization rates on different edge nodes of *Tencent* CDNs. They are all lower than 20%. It is consistent with the results of existing studies [6], [17]. As computing resources at the edge become abundant and CPU resources in the cloud become rare [17], [20], [22], offloading computational functions to the edge nodes becomes essential.

Observation ❷ : There are profits for replacing "fetching" with "generation". On *Tencent* CDNs, the fetching time is around 230ms, which is more than 100 times the hit. This gap is sufficient to implement generation and provide a generative hit. Furthermore, we found that image format conversion or scaling takes around 40ms, while the transformations of data blocks are even shorter, taking less than 10ms.

Observation ❸ : There are security frameworks to generate data without copyright disputes. Generating data on the edge node involves accessing the data in the cache and may raise security concerns, which is beyond the scope of this paper. While we cannot guarantee complete protection against all illegal behavior, we have made adjustments suitable for the CDN environment by employing the Secure Content Delivery and Deduplication (SCD2) scheme [22]. In addition, the generated data only offers browsing services and cannot be stored in CDNs. Meanwhile, the generation tool is automatic and open source. The copyright in the generated data, in principle, belongs to the owner of the original data.

Motivation. Based on the observations above, we believe that the availability of computing resources, along with feasibility in terms of time and security guarantees, all support cache hits through data generation on CDNs. By utilizing idle computing resources on edges, near-data processing may be achieved, fulfilling the functional requirements of edge nodes in the design of next-generation distributed systems [17]. As

[2]https://www.akamai.com/solutions/content-delivery-network
[3]https://cloud.google.com/cdn
[4]https://www.alibabacloud.com

Authorized licensed use limited to: Huazhong University of Science and Technology. Downloaded on April 15,2025 at 12:04:00 UTC from IEEE Xplore. Restrictions apply.

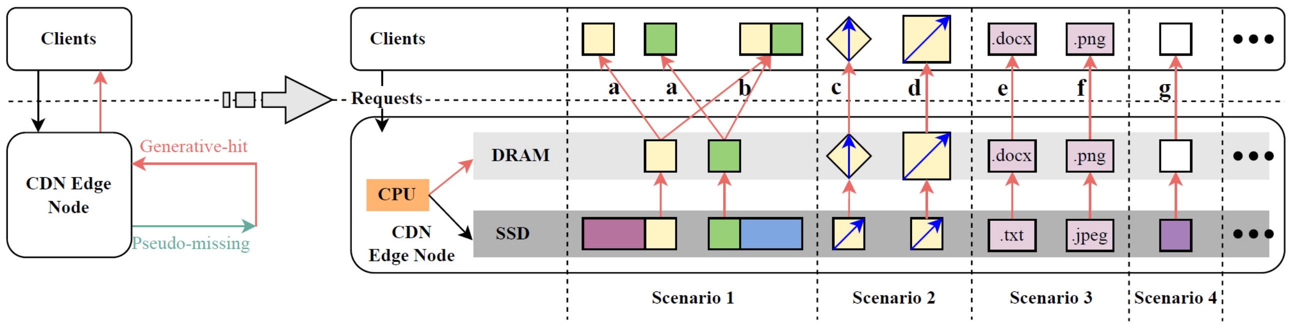


Fig. 3. Several pseudo-missing scenarios and their generative-hit schemes.

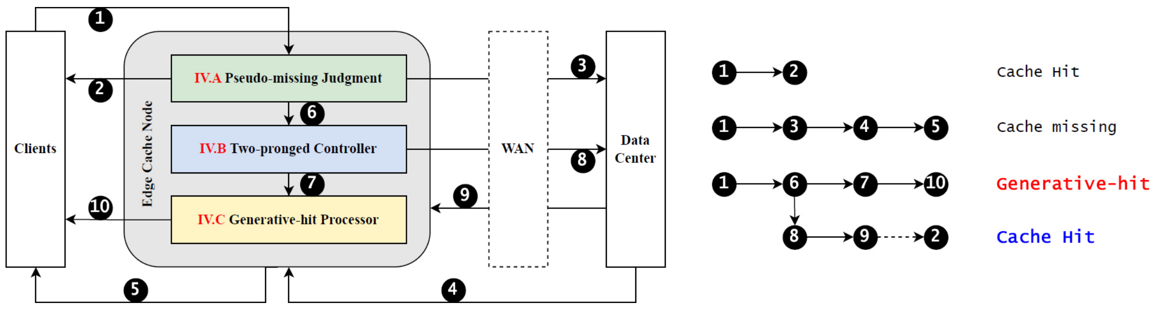


Fig. 4. The workflow of cache hit, cache missing, and generative-hit in the architecture of `CGHit`. In particular, the right side gives the specific process for the three cases. The `Cache Hit` marked in blue are fundamentally different from the `Cache Hit` in black. The essential difference between the two is that the data hit by the former is fetched from the data center after pseudo-missing, while the data hit by the latter exists in the cache itself.

the pioneering study in this field, we will outline scenarios for utilizing generated data to facilitate hits on CDNs.

III. Generative Hit

A. Pseudo-missing Scenarios

The pseudo-missing scenario occurs when the requested data is unavailable in the cache but can be obtained through a generative hit. The generative hit refers to the generation of requested data using cached data and local computing resources. When generation is faster than fetching, a missing request can be responded to by a generative hit, resulting in decreased latency. We call this request a pseudo-missing request. Four pseudo-missing scenarios are shown in Fig. 3 and corresponding generative-hit schemes are given below.

• **Scenario 1: The request data is a segment or combination of segments from the cached data.** The corresponding part of the cached data is disassembled in DRAM and delivered to the user using computing resources on the edge node (*i.e.*, **a** extraction). The corresponding parts of data can also be combined in DRAM and then the results are delivered to the user (*i.e.*, **b** splicing).

• **Scenario 2: The requested data is identical to the cached data in terms of content, but these data have different forms (*e.g.*, shape, size, resolution, *etc.*).** The copy of the corresponding cached data will be processed (*i.e.*, **c** rotating, **d** scaling, enhancing, *etc.*) to fit the user's request and delivered to the user.

• **Scenario 3: The requested data is identical to the cached data in terms of content, but these data have different coding formats (*i.e.*, e .txt to .docx, f .jpeg to .png, *etc.*).** The copy of the corresponding cached data will be converted into the requested data and delivered.

• **Scenario 4: The requested data is similar or related to the cached data regarding content.** When the request data is consistent with the cached data in modality, the copy of the corresponding cached data will be updated (*e.g.*, **g** removing an object from an image, changing the subject in a document, *etc.*). Otherwise, the requested data will be generated from random vectors using information from both the request data and the cached data, and then delivered.

The first three scenarios compile cached data with a high degree of reliability. The last scenario is only applicable to data reads for fuzzy queries or recommendations. The data generation for the above scenarios draws on content locality. To reduce content redundancy and SSD write times, the generated data is only used for responding to the user's pseudo-missing requests and is not stored on SSDs.

B. Rationale

Fig. 4 represents the workflow dealing with hit, missing, and pseudo-missing. We use the order number in Fig. 4 to represent the latency resulting from each corresponding operation/behavior. We define the latency for a pseudo-missing request and a missing request as $\mathcal{T}^{*}$=❶+❻+❼+❿ and

Authorized licensed use limited to: Huazhong University of Science and Technology. Downloaded on April 15,2025 at 12:04:00 UTC from IEEE Xplore. Restrictions apply.

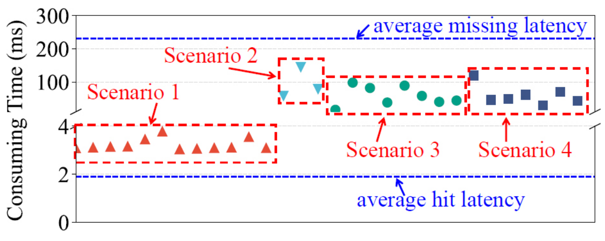


Fig. 5. The latency of generative hits in different pseudo-missing scenarios.

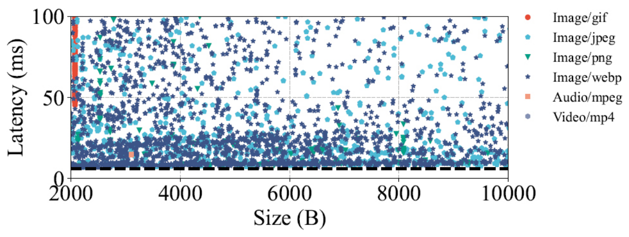


Fig. 6. The latency caused by missing six types of data in Tencent CDN.

$\mathcal{T}$=❶+❸+❹+❺, respectively. Given n requests for the data with the same content and their access times $t_1, t_2, ..., t_n$, we discuss the average access latency of these requests in the missing and pseudo-missing scenarios. Note that we assume that ❶+❷=0 since the latency of a cache hit is much smaller than that of a missing request.

• **Missing:** According to the traditional caching policy, if a cache missing occurs at t_1, the latency of this access is $\mathcal{T}$. At t_2, if $t_2 - t_1 > \mathcal{T}$, the second access will hit with a latency of 0. If ensuing requests at $t_3, ..., t_n$ are hit before the data with the requested content is evicted, the optimal average latency of these n requests is $\frac{\mathcal{T}}{n}$.

• **Pseudo-missing:** Assuming that a pseudo-missing request occurs at t_1, resulting in a latency of $\mathcal{T}^*$. At t_2, the pseudo-missing will occur again and yield a latency of $\mathcal{T}^*$. The same latency will be yielded at $t_3, ..., t_n$. As a result, the average latency of these n requests is $\mathcal{T}^*$.

★ **Analysis:** When $\frac{\mathcal{T}}{n} > \mathcal{T}^*$, the latency is lowered because of the new policy designed for pseudo-missing. Note that as n increases, it is possible for $n \times \mathcal{T}^* > \mathcal{T}$. We attribute it to giving up on writing SSDs for generated data. However, it is conducive to the reduction of data redundancy and the preservation of content diversity. To achieve a balance, we have developed further optimization strategies.

• **Further optimization for pseudo-missing:** To further reduce the average access latency caused by pseudo-missing without excessively reducing content diversity, we adopt a two-pronged strategy. As ❽ and ❾ shown in Fig. 4, while using the generated data to respond to user requests promptly, the edge node still fetches data and writes it to SSDs. Assume that at the p-th request, the two-pronged strategy achieves the above operation before the arrival of the q-th request. The average latency of pseudo-missing can be optimized to $\frac{q-1}{n} \times \mathcal{T}^*$ because of $t_{q-1} - t_p < \mathcal{T} < t_q - t_p$. Furthermore, since the original data obtained asynchronously can be provided for a hit in the next request, there is no need to write the generated data to SSDs.

★ **Analysis:** Although the average latency can be smaller as q decreases, this strategy is constrained by the actual situation. For example, if there is only one access, fetching is meaningless. Even if $q > 1$ but the latency for fetching data is greater than that of pseudo-missing processing, it still causes performance degradation. Consequently, to prevent meaningless fetching, it is essential to establish a decision model that determines whether the strategy should be executed. In addition, the model should consider the availability of idle computing resources. The implementation of the decision model is described in § IV-B.

IV. CGHit Design

We illustrate the overall architecture of `CGHit` in Fig. 4. Specifically, we introduce three modules: the pseudo-missing judgment module (see § IV-A), the two-pronged controller (see § IV-B), and the generative-hit processor (see § IV-C). Based on the `CGHit` architecture, the request-response state of the client has fundamentally changed. The original client request has only two response states, namely cache hit (❶→❷) and cache missing(❶→❸→❹→❺), and the introduction of `CGHit` architecture adds a new response mode (*i.e.*, generative-hit ❶→❻→❼→❿), while adding a new cache hit case (❶→❻→❽→❾- -→❷).

A. Pseudo-missing Judgment Module

This module considers the time spent generating data and idle CPU resources. When generating data takes longer than fetching it, or there aren't enough CPU resources, the module will bypass pseudo-missing operations and handle the request using standard methods. To do this, we design a latency predictor based on historical data (such as data type, data size, etc.) to predict the response time for the missing request, denoted by $t_{predict}$. Meanwhile, we tested and counted the time (*i.e.*, $t_{generate}$) it took to generate hits in different pseudo-missing scenarios, as shown in Fig. 5. Once $t_{predict} > t_{generate}$, the generating hit is a better operation than fetching in latency. Note that there are limitations, *i.e.*, if the edge node's computing resources are too constrained, the module will not execute even if the previous conditions are met. For the implementation of the predictor, we recorded the missing latency of six common data types in Tencent CDN over a period, as shown in Fig. 6. The latency for Gif and PNG data types is independent of data size, while the distribution for jpeg and webp is more varied. Our predictor uses historical access information to make predictions based on data size and type, and integrates the findings from actual analysis to make comprehensive decisions.

Then, we develop a set of global identifiers in the judgment function (*i.e.*, `get_metadat(URLReq req,`

Authorized licensed use limited to: Huazhong University of Science and Technology. Downloaded on April 15,2025 at 12:04:00 UTC from IEEE Xplore. Restrictions apply.

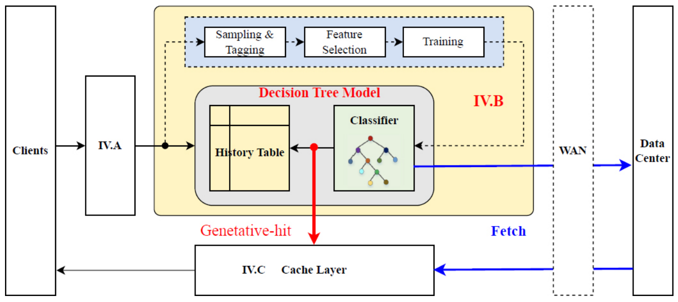


Fig. 7. The architecture of two-pronged controller.

`TNode &node)`) to represent the hit, missing, and pseudo-missing scenarios. Each identifier for block data comprises the file name (*i.e.*, `req.key`) along with specific parameters (*i.e.*, `req.par`). If the file name in the URL request does not match any of the cached files, the requested data is marked as missing. The requested data will be fetched from the data center, written to the edge node, and then sent to the client. These behaviors correspond to ❸, ❹, and ❺ shown in Fig. 4, respectively. For requests where the file name matches but the related parameters differ, they are processed as pseudo-missing. The behavior ❻ means the start of pseudo-missing processing. If both the file name and parameters match exactly, it is considered a hit. The hit data will be sent to the client according to the behavior ❷.

B. Two-pronged Controller

The two-pronged controller performs two functions simultaneously. It issues instructions to generate the requested data. Meanwhile, it asynchronously fetches data from the data center and writes it to the edge nodes when it predicts that the pseudo-missing data will be frequently accessed. As the prediction component of the controller, the decision model should accurately predict the reusability of requested files. Compared to other prediction models, decision trees offer quick decision-making, adaptability to diverse features, and no assumption of linearity. The experimental test results in [19] also demonstrate that decision trees outperform Naive Bayes [21], BP Neural Network [3], KNN [5], Random Forest [12], and Stochastic Regression in terms of decision accuracy for cache decision problems. Additionally, decision trees have a relatively low cost and faster speed.

Thus, we employ a decision tree model shown in Fig. 7 to achieve prediction, referring to [19], [27]. To ensure effective predictions, we extract several features, including file type, file size, age of file, recency, and frequency, to learn the prediction model. To ensure the performance of the main business, we set the access confidence sampling, tagging, and model training offline, where the training update cycle can be modified according to the actual situation. The pseudo-missing request is input to the decision tree model, and the decision tree model (*i.e.*, Classifier in Fig. 7) makes the decision. There are two possible decisions here: fetching and generative-hit. Based on the historical data in the history table, if the decision tree model identifies a pattern suggesting a high number of future requests for the same content, it triggers the fetching operation, retrieving the data from the data center and writing it to the cache layer. Otherwise, it directly executes IV.C shown in Fig. 7, using local data for content generation and responding to pseudo-missing requests.

C. Extensible Generative-hit Processor

In this module, we focus on generating the requested data for the pseudo-missing scenarios. According to the available open-source dataset, we initially combine the processing functions for image data and block data. To facilitate the extension for handling other types of data (*e.g.*, text, video, *etc.*), we record the format and modality of the data in `metadata`. For multimedia data, we will additionally record the 128-bit similarity hash codes [7] that represent the content of the data. According to the extended `metadata`, all types of cached data can be evaluated to determine their eligibility for participating in the pseudo-missing processing. If latency is acceptable, the data will be sent to the appropriate algorithm model, such as LLM, for compilation or generation.

- **For image data.** We introduced a third-party open-source image processing library[5] to achieve image generation in Scenario 2, Scenario 3, and Scenario 4 mentioned in § III-A. This library is built on image-processing techniques developed in C++. To achieve on-the-fly processing, we prefer to execute efficient algorithms. If high data quality is indispensable, more complex algorithms can be selected instead.
- **For block data.** In Scenario 1 mentioned in § III-A, the required data blocks are separated according to the parameters given by the user URL. If a merge operation is required, data is successively read from the source files into the merge file after the files that need to be merged have been sorted and an empty merge file has been created.

As shown in Fig. 5, we measured the time taken by generative hits in scenarios mentioned in § III-A. For block data, we measured the generative hit time in Scenario 1 using different block sizes (ranging from 4KB to 1MB). It can be observed that the generative-hit time is around 1ms. In addition, the generative-hit time in Scenario 2, Scenario 3, and Scenario 4 for image data remains below 150ms.

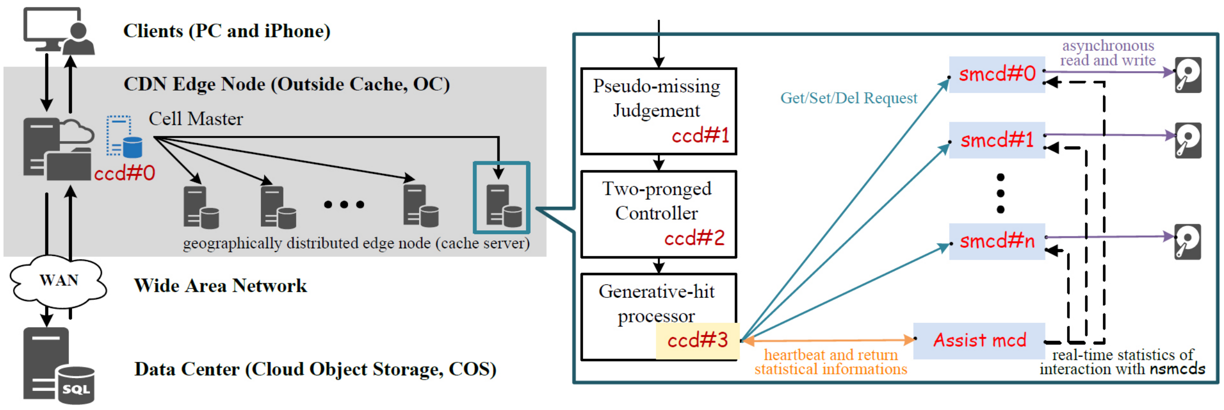
D. Deployment of prototype

As shown in Fig. 8, *PicCloud* is a multithreaded, event-based CDN that configures memory and SSDs. *PicCloud* consists primarily of two modules: *Cell Master* and *Cache Server*. As a resource management module, the *Cell Master* manages the routing information of the entire system. It runs a process independently (*i.e.*, `ccd#0`), periodically checks the running status of each *cache server*, and receives statistics reported by each *cache server*. The *Cache Servers* are distributed edge nodes, used for the storage and processing of data.

`CGHit` is distributed on each edge node that runs three processes. The three processes manage the pseudo-missing

[5]https://github.com/nackily/imglib

Authorized licensed use limited to: Huazhong University of Science and Technology. Downloaded on April 15,2025 at 12:04:00 UTC from IEEE Xplore. Restrictions apply.



Clients (PC and iPhone)
CDN Edge Node (Outside Cache, OC)
Cell Master
ccd#0
geographically distributed edge node (cache server)
WAN
Wide Area Network
SQL
Data Center (Cloud Object Storage, COS)
Pseudo-missing Judgement
ccd#1
Two-pronged Controller
ccd#2
Generative-hit processor
ccd#3
Get/Set/Del Request
smcd#0
smcd#1
smcd#n
Assist mcd
asynchronous read and write
heartbeat and return statistical informations
real-time statistics of interaction with nsmcds

Fig. 8. Deploy `CGHit` on *Tencent* CDN (*PicCloud*).

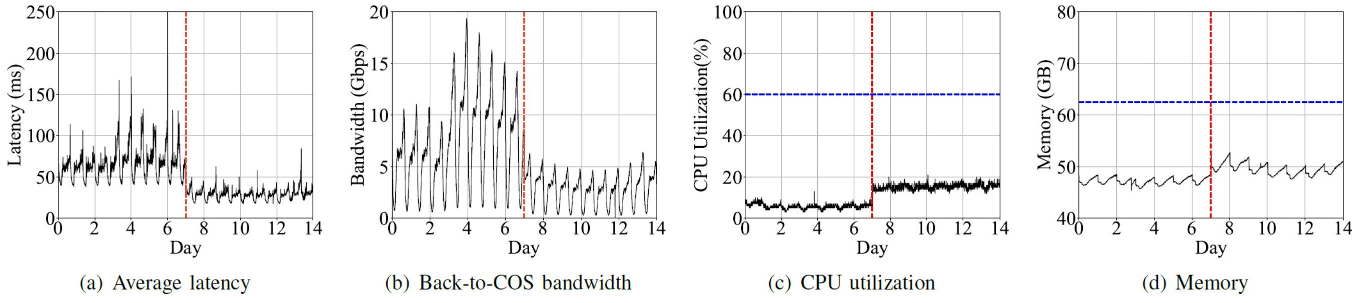


Fig. 9. Comparison of average latency and back-to-COS bandwidth, CPU utilization, and memory usage before and after deploying `CGHit`. The red vertical dashed line indicates the date when `CGHit` was deployed. The blue horizontal dashed lines in (c) and (d) are the default upper limits (*i.e.*, 60% and 64GB) for CPU utilization and memory usage, respectively, which are used to reserve buffers to deal with sudden traffic surges.

TABLE II
THE TAIL LATENCY MONITORED FROM *PicCloud*.

	Original System	**`CGHit` Deployed**
P99 Latency (ms)	118.37	43.46
P99.9 Latency (ms)	171.23	62.67

judgment module (*i.e.*, `ccd#1`), the two-pronged controller module (*i.e.*, `ccd#2`), and the generative-hit processor module (*i.e.*, `ccd#3`), respectively. Each thread `smcd` corresponds to a physical disk and manages the corresponding disk in a single thread. The thread `Assist mcd` process is responsible for managing `n smcd`s, including collecting heartbeat information and statistics on the latency of get/set/del operation, disk data elimination information, workload, *etc*.

V. Evaluation

We have implemented a prototype within *Tencent* CDNs (*PicCloud*) and tests `CGHit` on the simulator [1] based on two open-source traces [25], [26]. To determine the cached data involved in the generative hit for block data, we check the prefix of the file names. For image data, we check the Hamming distances between similarity hash codes calculated by the DSTH model [7]. When `CGHit` was deployed, we maintained the cache replacement policy (LRU) used in the original system. The metrics we focus on are access latency and back-to-COS bandwidth.

A. Performance on the real system

Improvement. As shown in Fig. 9(a) and Fig. 9(b), we measure the performance changes in the average access latency and back-to-COS traffic in *PicCloud*. After updating the system architecture, the average access latency drops 56.04%, and the average back-to-COS bandwidth drops 60.28%. We also obtain the changes of the P99 and P99.9 latency after `CGHit` deployment from the monitoring system. As shown in Table II, the P99 latency drops by 74.91ms and the P99.9 latency drops by 108.56ms. These results confirm that `CGHit` can bring performance improvements.

Overhead. To visualize the changes in physical machine metrics after the deployment of `CGHit`, we depict the changes in CPU utilization and memory usage in Fig. 9(c) and Fig. 9(d), respectively. After deploying `CGHit`, the CPU utilization increases from 5.73% to 15.23%, and the memory usage increases from 47.16GB to 49.49GB. As a result, we believe that trading idle resources for an increase in cache performance is worth it, albeit in additional CPU and memory resources required for data processing.

B. Performance on the simulator

We modify the simulator used in LRU-MAD [1] to test the metrics yielded by caching algorithms. The algorithms include LRU, ARC [8], LHD [2], and LRU-MAD [1], where LRU-MAD is a latency-sensitive algorithm. For Case A and Case

Authorized licensed use limited to: Huazhong University of Science and Technology. Downloaded on April 15,2025 at 12:04:00 UTC from IEEE Xplore. Restrictions apply.

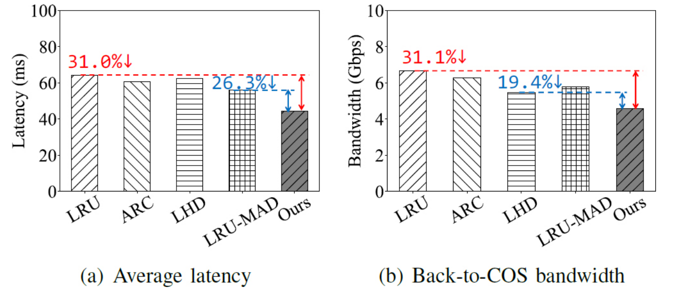


Fig. 10. Compared LRU on `CGHit` with other algorithms on the original system in terms of average access latency and back-to-COS bandwidth.

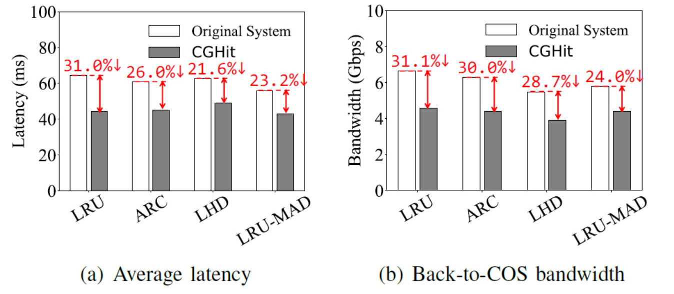


Fig. 11. Average access latency and back-to-COS bandwidth yielded by different cache algorithms on the original system and `CGHit`.

B, we handle pseudo-missing in Scenario 3 and Scenario 4 as well as in Scenario 1 and Scenario 2, respectively.

For superiority, we compare running LRU on `CGHit` with running other caching algorithms on the original system. As shown in Fig. 10(a), our scheme's average access latency decreased by 31.0% compared to LRU. Compared to the best LRU-MAD, our scheme has a 26.3% reduction in latency. In terms of back-to-COS bandwidth shown in Fig. 10(b), our result is 1.21Gbps lower than the optimal result of other algorithms, with an improvement of 19.4%.

For enhancement effect, we compare `CGHit` with the original system using the same caching algorithm. As shown in Fig. 11, the latency and back-to-COS bandwidth provided by different algorithms both decrease on `CGHit`. This suggests that any caching algorithm has the potential to enhance performance on `CGHit`.

For content redundancy, we compare our scheme (*i.e.*, LRU running on `CGHit`) to the baseline (*i.e.*, LRU running on the original system) on two open-source datasets [25], [26]. We do not compare the data deduplication schemes because they do not prioritize access latency. As shown in Fig. 12, our scheme achieves an average reduction of 20.0% and 25.2% for Case A and Case B, respectively. This reduction involving content redundancy can reduce the number of "invalid writes" of SSDs. In this experiment, the rates of fetching data in the two-pronged control strategy are 5.2% and 14.6% for Case A and Case B, respectively. Due to a lack of computing resources, 3.8% and 6.1% of pseudo-missing requests are treated as traditional missing requests, respectively.

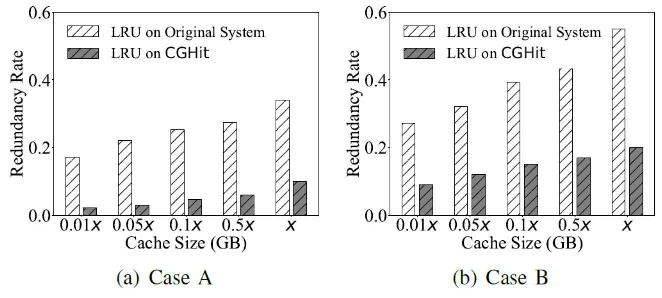


Fig. 12. Comparison of content redundancy rates yielded on the simulator. $x = 512$GB in Case A and $x = 2048$GB in Case B.

VI. Related Work

A. Improving QoE

Enhancing the quality of experience (QoE) in caching is a well-explored area. Traditional cache algorithms, like replacement and admission algorithms, mainly emphasize data locality. CACHEUS [13] and LHD [2] refine algorithms through local access rules, while LRB [15] and RLB [23] boost hit rates with machine learning models. LRU-MAD [1] is notable for tackling the delay-sensitive issue. Another strategy involves moving computation closer to the data in disaggregated storage [24].

B. Edge Computing

Edge computing deploys computational resources closer to data generation points to reduce latency and enhance efficiency. Processing data at edges, such as on routers or gateways, enables quicker decision-making, benefiting applications that require real-time processing and low latency.

HSBS [28] has been specifically tailored for solving the integration problem of edge computing architecture and blockchain technology. This model employs a master-slave structure, establishing a master chain in the cloud center and multiple shared chains at the edge layer. HSBS enhances system throughput and reduces storage redundancy, offering significant advantages for data management in edge computing environments. Yao *et al.* [29] studied the joint modeling of caching and request routing in collaborative edge servers, addressing model loading latency issues and effectively reducing context switching costs. In addressing the request distribution and caching challenges in serverless edge computing, S-Cache [30] has devised an online request dispatching algorithm and an adaptive caching strategy adjusted through comprehensive multifactor considerations, achieving significant advantages in reducing average response times and cold start frequencies. Yang *et al.* [31] investigated the mobile edge caching problem under storage capacity constraints, aiming to minimize user request latency, and achieved remarkable results. For the single timeslot scenario in mobile edge caching, we proposed a heuristic strategy to determine which files should be cached on edge servers. In the multi-timeslot scenario, we introduced a caching strategy called MF-ECS, based on multi-agent deep reinforcement learning, where each edge server acts as an agent to decide which files to

Authorized licensed use limited to: Huazhong University of Science and Technology. Downloaded on April 15,2025 at 12:04:00 UTC from IEEE Xplore. Restrictions apply.

cache. To tackle cold start issues, a heuristic strategy was also proposed to optimize outcomes during the initial phase.

VII. Conclusion

We observe the pseudo-missing phenomenon, define the generative hit, and propose a content-oriented generative-hit framework (`CGHit`) for CDNs. `CGHit` provides the requested data during pseudo-missing scenarios by generating it at the cache layer when there are idle computational resources. To prevent overloading CPU utilization due to frequent access for pseudo-missing data, `CGHit` employs a decision tree model to determine whether to fetch data. `CGHit` offers a fresh perspective on enhancing caching QoE through data generation. We believe that `CGHit` is able to improve the performance of CDNs based on increasing computing resources in edge nodes and the advancement of large language models.

VIII. Acknowledgments

This work was achieved in the Key Laboratory of Information Storage System and Ministry of Education of China. It was supported by the National key research and development program Grant No. 2023YFB4502701, National Natural Science Foundation of China (Grant No.62232007, No.61821003), National Natural Science Foundation of China No.61902135, and the Natural Science Foundation of Hubei Province No.2022CFB060.

References

[1] N. Atre, J. Sherry, W. Wang, and D. S. Berger, "Caching with delayed hits," in Proceedings of the Annual conference of the ACM Special Interest Group on Data Communication on the applications, technologies, architectures, and protocols for computer communication, pp. 495–513, 2020.

[2] N. Beckmann, H. Chen, and A. Cidon, "{LHD}: Improving cache hit rate by maximizing hit density," in 15th USENIX Symposium on Networked Systems Design and Implementation (NSDI 18), pp. 389–403, 2018.

[3] M. Buscema, "Back propagation neural networks," in Substance use & misuse, vol. 33, no. 2, pp. 233-270, 1998.

[4] R. C. Fernandez, A. J. Elmore, M. J. Franklin, S. Krishnan, and C. Tan, "How large language models will disrupt data management," in Proceedings of the VLDB Endowment, vol. 16, no. 11, pp. 3302–3309, 2023.

[5] G. Guo, H. Wang, D. Bell, Y. Bi, and K. Greer, "KNN model-based approach in classification," On The Move to Meaningful Internet Systems 2003: CoopIS, DOA, and ODBASE: OTM Confederated International Conferences, CoopIS, DOA, and ODBASE 2003, Catania, Sicily, Italy, November 3-7, 2003. Proceedings, pp. 986–996, 2003.

[6] A. Khansoltani, S. Jamali, and R. Fotohi, "A request redirection algorithm in content delivery network: Using promethee approach," in Wireless Personal Communications, vol126, no. 2, pp. 1145–1175, 2022.

[7] Y. Liu, J. Song, K. Zhou, L. Yan, L. Liu, F. Zou, and L. Shao, "Deep self-taught hashing for image retrieval," in IEEE transactions on cybernetics, vol. 49, no. 61, pp. 2229–2241, 2018.

[8] N. Megiddo and D. S. Modha, "{ARC}: A {Self-Tuning}, low overhead replacement cache," in 2nd USENIX Conference on File and Storage Technologies (FAST 03), 2003.

[9] I. Morrey B. Charles, and D. Grunwald, "Content-based block caching," in Proceedings of the 23rd IEEE Conference on Mass Storage Systems and Technologies, 2006.

[10] A. Nachman, S. Sheinvald, A. Kolikant, and G. Yadgar, "GoSeed: Optimal seeding plan for deduplicated storage," in ACM Transactions on Storage (TOS), vol. 17, no. 3, pp. 1–28, 2021.

[11] J. Ren and Q. Yang, "A new buffer cache design exploiting both temporal and content localities," in 2010 IEEE 30th International Conference on Distributed Computing Systems, pp. 273–282, 2010.

[12] S. J. Rigatti, "Random forest," in Journal of Insurance Medicine, vol. 47, no. 1, pp. 31–39, 2017.

[13] L. V. Rodriguez, F. Yusuf, S. Lyons, E. Paz, R. Rangaswami, J. Liu, M. Zhao, and G. Narasimhan, "Learning cache replacement with {CACHEUS}," in 19th USENIX Conference on File and Storage Technologies (FAST 21), pp. 341–354, 2021.

[14] M. S. Sajjadi, R. Vemulapalli, and M. Brown, "Frame-recurrent video super-resolution," in Proceedings of the IEEE conference on computer vision and pattern recognition, pp. 6626–6634, 2018.

[15] Z. Song, D. S. Berger, K. Li, A. Shaikh, W. Lloyd, S. Ghorbani, C. Kim, A. Akella, A. Krishnamurthy, and E. Witchel, "Learning relaxed belady for content distribution network caching," in 17th USENIX Symposium on Networked Systems Design and Implementation (NSDI 20), pp. 529–544, 2020.

[16] Z. Song, K. Chen, N. Sarda, D. Altınbüken, E. Brevdo, J. Coleman, X. Ju, P. Jurczyk, R. Schooler, and R. Gummadi, "{HALP}: Heuristic aided learned preference eviction policy for {YouTube} content delivery network," in 20th USENIX Symposium on Networked Systems Design and Implementation (NSDI 23), pp. 1149–1163, 2023.

[17] T. Taleb, P. A. Frangoudis, I. Benkacem, and A. Ksentini, "CDN slicing over a multi-domain edge cloud," in IEEE Transactions on Mobile Computing, vol. 19, no. 9, pp. 2010–2027, 2019.

[18] I. Trummer, "Can Large Language Models Predict Data Correlations from Column Names?" in Proceedings of the VLDB Endowment, vol 16, no. 13, pp. 4310–4323, 2023.

[19] H. Wang, X. Yi, P. Huang, B. Cheng, and K. Zhou, "Efficient SSD caching by avoiding unnecessary writes using machine learning," in Proceedings of the 47th International Conference on Parallel Processing, pp. 1–10, 2018.

[20] Y. Wang, H. Dai, X. Han, P. Wang, Y. Zhang, and C. Xu, "Cost-driven data caching in edge-based content delivery networks," in IEEE Transactions on Mobile Computing, vol 22, no. 3, pp. 1384–140, 2021.

[21] G. I. Webb, E. Keogh, and R. Miikkulainen, "Naïve Bayes," in Encyclopedia of machine learning, vol 15, no. 1, pp. 713–714, 2010.

[22] K. Xue, P. He, J. Yang, Q. Xia, and D. S. Wei, "SCD2: Secure content delivery and deduplication with multiple content providers in information centric networking," in IEEE/ACM Transactions on Networking, vol. 30, no. 4, pp. 1849–1864, 2022.

[23] G. Yan and J. Li, "RL-Bélády: A unified learning framework for content caching," in Proceedings of the 28th ACM International Conference on Multimedia, pp. 1009–1017, 2020.

[24] Q. Zhang, P. A. Bernstein, D. S. Berger, B. Chandramouli, V. Liu, and B. T. Loo, "Compucache: Remote computable caching using spot vms," in Annual Conference on Innovative Data Systems Research (CIDR'22), 2022.

[25] K. Zhou, S. Sun, H. Wang, P. Huang, X. He, R. Lan, W. Li, W. Liu, and T. Yang, "Demystifying cache policies for photo stores at scale: A tencent case study," in Proceedings of the 2018 International Conference on Supercomputing, pp. 284–294, 2018.

[26] K. Liu, K. Wu, H. Wang, K. Zhou, J. Zhang, and C. Li, "SLAP: An adaptive, learned admission policy for content delivery network caching," in 2023 IEEE International Parallel and Distributed Processing Symposium (IPDPS), pp. 457–467, 2023.

[27] H. Wang, J. Zhang, P. Huang, X. Yi, B. Cheng, and K. Zhou, "Cache what you need to cache: Reducing write traffic in cloud cache via "one-time-access-exclusion" policy," in ACM Transactions on Storage (TOS), vol. 16, no. 3, pp. 1–24, 2020.

[28] C. Li, H. Pan, H. Qian, Y. Li, X. Si, K. Li, and B. Zhang, "Hierarchical sharding blockchain storage solution for edge computing," in Future Generation Computer Systems (FGCS), 2024.

[29] M. Yao, L. Chen, J. Zhang, J. Huang, and J. Wu, "Loading Cost-Aware Model Caching and Request Routing for Cooperative Edge Inference," in IEEE International Conference on Communications, pp. 2327–2332, 2022.

[30] C. Chen, L. Nagel, L. Cui, and F. P. Tso, "S-Cache: Function Caching for Serverless Edge Computing," in Proceedings of the 6th International Workshop on Edge Systems, Analytics and Networking (EdgeSys '23), pp. 1–6, 2023.

[31] Y. Yang, K. Lou, E. Wang, W. Liu, J. Shang, X. Song, D. Li, and J. Wu, "Multi-Agent Reinforcement Learning Based File Caching Strategy in Mobile Edge Computing," in IEEE/ACM Transactions on Networking (TON), vol. 31, no. 6, pp. 3159–3174, 2023.

Authorized licensed use limited to: Huazhong University of Science and Technology. Downloaded on April 15,2025 at 12:04:00 UTC from IEEE Xplore. Restrictions apply.