SMURF: Efficient and Scalable Metadata Access for Distributed Applications

Bing Zhang and Tevfik Kosar, *Senior Member, IEEE*

Abstract—In parallel with big data processing and analysis dominating the usage of distributed and Cloud infrastructures, the demand for distributed metadata access and transfer has increased. The volume of data generated by many application domains exceeds petabytes, while the corresponding metadata amounts to terabytes or even more. This paper proposes a novel solution for efficient and scalable metadata access for distributed applications across wide-area networks, dubbed SMURF. Our solution combines novel pipelining and concurrent transfer mechanisms with reliability, provides distributed continuum caching and semantic locality-aware prefetching strategies to sidestep fetching latency, and achieves scalable and high-performance metadata fetch/prefetch services in the Cloud. We incorporate the phenomenon of semantic locality awareness for increased prefetch prediction rate using real-life application I/O traces from Yahoo! Hadoop audit logs and propose a novel prefetch predictor. By effectively caching and prefetching metadata based on the access patterns, our continuum caching and prefetching mechanism significantly improves the local cache hit rate and reduces the average fetching latency. We replay approximately 20 Million metadata access operations from real audit traces, where SMURF achieves 90% accuracy during prefetch prediction and reduced the average fetch latency by 50% compared to the state-of-the-art mechanisms.

Index Terms—Heterogeneity, scalability, metadata access, prefetch prediction, continuum caching, semantic locality.

1 Introduction

We are witnessing a new era that offers new opportunities to conduct data-intensive scientific research with the help of recent advancements in computational, storage, and network technologies. With the rapid deployment of distributed infrastructures and the collaborations between different organizations (e.g., XSEDE [1], OSG [2], Chameleon [3] and Cloudlab [4]), it is feasible and promising to run scientific applications on these large-scale geo-distributed infrastructures. In many application domains, including environmental and coastal hazard prediction, climate modeling, high-energy physics, astronomy, and genome mapping, the volume of data generated has already exceeded petabytes, while the corresponding metadata (the data providing information about one or more aspects of the data) [5], [6] amounts to terabytes or even more [7]. According to Roselli [8]'s study, more than 50% of all I/O operations are due to metadata-intensive computing, and the requests to read file attributes dominate in all workloads. The data movement is the common operation between the data I/O nodes, compute clusters, and user workstations for reconstruction, analysis, and visualization of the data. The Cloud-hosted metadata catalog (e.g., Globus Catalog [9], iRODS Metadata Catalog [10]) mitigates the difficulty of browsing, tracking, and discovery of the data. Thus, remote metadata retrieval and searching always have been conducted frequently between the users and Cloud services, even over wide-area networks. Data lakes [11]–[13] have been proposed to meet the requirement that scientists and researchers are seeking broader access to different types of "raw data" organized in a contextual format that can be used across different projects. In contrast to the data warehouse schema [14], the data schema in a data lake is not predefined. With the help of a metadata description, a data lake system can annotate, integrate, and query the raw data. Without the metadata, data alone is not useful, and the data lake becomes a data swamp [11].

More recently, with the unprecedented growth of the Internet of Things (IoT) devices (e.g., sensors, virtual reality, smartphones, smart vehicles, smart homes, and smart grocery stores) connecting to the world [15], the drops in the cost of sensors [16], and new technologies in wired and wireless networks, more than 50 billion Edge devices are expected to be connected to the Cloud by 2022 [17]. IoT devices autonomously capture and ingest data and seamlessly integrate with the modern IT infrastructures, and it is tenable to argue that IoT data is becoming the Big Data. In the IoT data processing, the real-time response is the inherent core feature. Intelligent IoT applications such as camera-based monitoring systems collect real-time data and send the aggregated information to remote analytic platforms (e.g., Apache Storm [18], Apache Spark [19], and IBM Infosphere Streams [20]) for real-time decision-making. Traditional remote exchange of information from the distant Cloud cannot meet the ultra-low latency requirements of these time-sensitive and geographically dispersed IoT applications. Consequentially, the challenge beckons a paradigm shift in which the data and metadata can be accessed anywhere, anytime, and from any device.

Access to proper metadata is latency-sensitive also due to user experience and critical business operations: Google reported 20% revenue loss due to a specific experiment that

- *B. Zhang is with National Center for Supercomputing Applications University of Illinois at Urbana-Champaign, Champaign, IL 61801 E-mail: bing@illinois.edu*
- *T. Kosar is with the Department of Computer Science and Engineering at the University at Buffalo (SUNY), Buffalo, NY, 14260.*

increased the time to display search results by as little as 500 milliseconds; and Amazon reported 1% sales decrease for an additional delay of as little as 100 milliseconds [21]. Unfortunately, most of the existing studies have focused on the efficient and scalable transfer of large-scale data, and there has been little work focusing on the optimization of remote access and transferring of metadata [22] in wide-area networks. By considering long-distance network latency, the frequency of revalidation of metadata, and the rapid growth of IoT, efficient and scalable metadata access and transfer technologies are demanded and expected to become a cornerstone of modern distributed IT infrastructures.

In this paper, we present a novel metadata access and retrieval system, called SMURF, which is built on the distributed continuum caching and semantic locality-aware prefetching architecture to effectively fetch, prefetch, and cache metadata on different hierarchical layers (as shown in Figure 1) between clients and remote I/O servers in a wide-area network (WAN) setting. The merits of the SMURF system have been illustrated by the increased performance and scalability of the real-time metadata transferring and the low latency metadata access from heterogeneous remote I/O nodes. The main contributions of this paper include:

- An efficient, scalable, and interoperable metadata access and transferring technique for large-scale (i.e., millions of) metadata instances in WAN based on distributed continuum caching and prefetching to sidestep metadata access and transfer latency.
- A novel universal metadata transfer stream programming model which abstracts and reconstructs the definition of application-level transfer protocol request as a chain of commands and parsers. This enables the transfer stream to send and parse different protocol requests in a universal mechanism to provide maximum interoperability.
- A novel semantic locality-aware directory and file metadata prefetching and caching scheme which achieves over 90% accurate prefetch prediction rate.
- A detailed comparison of four different prefetch predictors (our semantic locality-aware prefetch predictor and three state-of-the-art predictors, namely, NEXUS, AMP and FARMER) and the legacy LRU cache on the real-life Yahoo HDFS traces.

The rest of the paper is organized as follows: Section 2 describes the proposed system architecture and discusses its design; Section 3 presents the simulation methodology and performance evaluations; Section 4 discusses existing work in this area; and Section 5 concludes the paper.

2 System Architecture

Two major goals of the SMURF system are (1) providing *interoperability* between heterogeneous and distributed nodes through on-the-fly inter-protocol translation and (2) improvement of the metadata transfer performance while reducing access latency and meeting the *scalability* demands to enable large-scale metadata access over WAN. Both of these capabilities are crucial in translating raw data into knowledge and discovery in an efficient way.

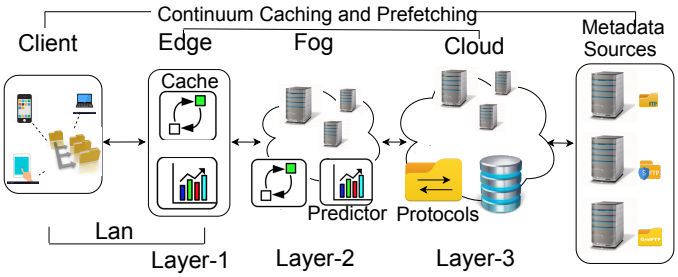

Fig. 1: Client devices fetch/prefetch metadata of interest in WAN via SMURF's distributed continuum cache and prefetch mechanism.

Interoperability is a critical need since valuable data may reside on different I/O servers or Cloud services. Especially, IoT devices and Edge nodes are highly heterogeneous [23]. In such heterogeneous and distributed environments, users have to install different protocol clients (e.g., HTTP [24], FTP [25], gsiFTP [26], IRODS [27], and Amazon S3 [28]) or implement and install the customized client and server with the help of well-known software libraries (e.g., gRPC [29], Apache Thrift [30]) , which makes the Edge devices of IoT ecosystem tightly coupled with specific services. Therefore, it is cumbersome and requires extra expertise to switch between the different services. SMURF deploys a Cloudlet Edge cluster to the proximity of IoT devices, where the Edge application can communicate with the Edge Cloudlet services through a a universal programming interface, and the remote metadata resources to be retrieved can be expressed as Uniform Resource Identifiers (URI) inside the requests.

Scalability is another central technical challenge in distributed metadata access. IoT applications, especially the sensor-based applications, have to process the dynamic workloads in real-time. The scalability of transferring large-scale metadata is a significant criterion to evaluate the performance of such systems. SMURF improves metadata transfer performance and meets scalability demands by using optimized pipelining and concurrency techniques. It also utilizes a hierarchical mechanism for the continuum caching and prefetching along the IoT-to-Cloud path, where the cache and the metadata prefetch predictor have been installed on each Edge/Fog layer. SMURF employs a novel approach based on semantic locality to predict the metadata access of distributed workloads over WAN.

In the following subsections, we introduce the details of the SMURF architecture, discuss each functional component of the system, and outline the system's end-to-end operation workflow.

2.1 SMURF Overview

SMURF has a hierarchical architecture, as shown in Figure 1, consisting of two major components: (1) a centralized Cloud cluster with the scalable fetch/prefetch services provides the universal pipelining and concurrent metadata transfer mechanisms with reliability; (2) the distributed continuum caching and smart prefetching strategies are deployed on Edge/Fog layers in clients' nearby networks, where the customized locality prefetch schemes utilize the local storage effectively to capture the future metadata to the proximity of clients.

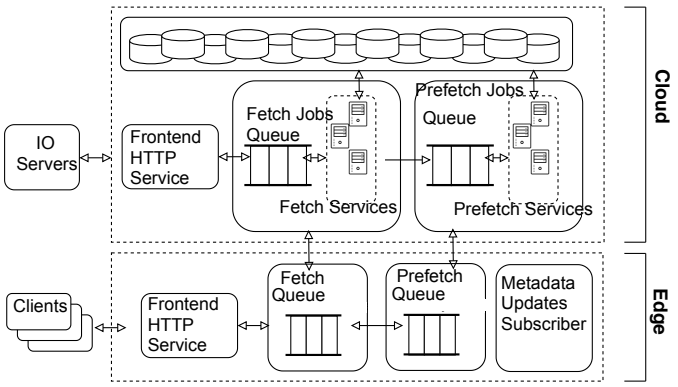

Fig. 2: High-level overview of SMURF's metadata fetching and prefetching between Edge server and Cloud.

Figure 2 shows the high-level metadata fetching and prefetching mechanism between the Edge and the Cloud. When clients submit the metadata fetch requests to the Edge server, the Edge server will first try to read the metadata from the local storage and reply to the clients. The Edge server will then send the requests to the prefetch predictor to analyze the request access patterns and predict the correlation metadata. The Edge server establishes connections to the Cloud server and sends/receives the fetch request and its correlation prefetch requests in pipelining. The Edge requests will be dispatched to the clusters of fetch/prefetch services in the Cloud and processed in parallel.

2.2 Universal Metadata Transfer Stream

The universal transfer stream is designed and implemented to coordinate and optimize the metadata access and transfer over different heterogeneous application-level transfer protocols over WAN. One universal transfer stream retrieves the metadata of interest from the remote I/O server using a single TCP connection, and it is novel in three aspects. First, quite different from traditional programming models, a proof of concept transfer stream programming model (Section 2.2.1) has been proposed by decoupling the application-level transfer protocol's definition (e.g., FTP [25], SFTP [31]) and the message transfer mechanism. We abstract and reconstruct the definition of protocol request as a chain of commands and parsers, and our transfer stream can send and parse different protocol requests in a universal mechanism. The protocol definition is provided as a library with programmability and extensibility; thus, users can follow the convention to customize their protocol to interact with the SMURF's universal transfer stream. Currently, SMURF supports application-layer transfer protocols such as FTP [25], SFTP [31], GSIFTP [32], IRODS [27], and Amazon S3 [28]. Second, our transfer stream can efficiently utilize the network bandwidth via metadata transfer pipelining. The value of pipeline capacity can configure transfer stream channel, where the *pipelining* capacity defines the maximum value of C requests (request is a user's logical activity, such as auth, login, and metadata retrieval) to be continuously sent over one TCP connection without blocking or waiting for the completion of the previous replies from remote servers. Third, the stream is aware of the transfer status and supports failure recovery. When the connection is broken, the stream can automatically re-establish the connection and notify the service to re-dispatch the pending requests.

2.2.1 Metadata Transfer Stream Programming Model

An application-level protocol defines how the application exchanges the information between its distributed components. Especially during end-to-end metadata transferring, one round of information exchange will be initiated by a command sent from a client to the remote server, waiting for the arrival of the reply from the remote server, and then terminated by parsing the reply based on the protocol definition. The completion of one metadata request will require at least one round of message exchange between the client and the server. This whole process can be formally expressed in $Traditional(Request) = f_1(c_1) \circ f_2(c_2) \cdots f_n(c_n)$, where the completion of one request takes n rounds of message exchanges and the notion of f denotes a function to pack/send the protocol command c and parse the reply. The operator $\circ$ concatenating two adjacent functions defines the order of message sending, receiving and parsing, thus it can denote the strict dependent relationship between two adjacent functions $f_i \circ f_{i+1} \implies f_{i+1} = g(f_i)$ where the sending of current message depends on the reply of previous messages. Moreover, the operator $\circ$ will take at least one round-trip-time (RTT) between two dependent adjacent command transmissions.

We decompose the function f into two parts: message sending s and message parsing p. Both functions of s and p can take a list of input parameters and apply the function execution over each element inside the given parameters. When the protocol definition does not require the dependency, then the pipeline transferring of one metadata request can be expressed as $Pipeline(request) = s(c_1, c_2, \cdots, c_n) \circ p(c_1, c_2, \cdots, c_n)$. This expression follows three constraints: (1) The sender s can continuously send request commands in the sequence order of $c_1, c_2, \cdots, c_n$ without blocking, and meanwhile, the parser p will parse the incoming replies in the same sequence order. (2) The operator $\circ$ strictly guarantees that the order of sending a command c_i happens before parsing the reply of this command c_i, namely, $order(s(c_i)) < order(p(c_i))$. The overlapping executions of sending messages and parsing replies can be denoted as $s(c_k \cdots c_n) \cap p(c_1 \cdots c_{k-1})$. It is possible because of the settings of the pipeline levels and the value of the RTT between the client and the server. (3) The completion of $Pipeline(request)$ will take at least one RTT.

If the protocol is stateless and requires the independent relationship between two adjacent commands' sending and receiving, the pipeline transferring of multiple m requests over one stream will be expressed as (assuming transfer of the same type of requests, each of which consists of n commands):

$$
\begin{aligned}
Pipeline(R_1, R_2, \cdots, R_m) = \\
& s(c_{11}, c_{12}, \cdots, c_{1n}) \circ p(c_{11}, c_{12}, \cdots, c_{1n}) \\
& s(c_{21}, c_{22}, \cdots, c_{2n}) \circ p(c_{21}, c_{22}, \cdots, c_{2n}) \\
& \qquad \vdots \\
& s(c_{m1}, c_{m2}, \cdots, c_{mn}) \circ p(c_{m1}, c_{m2}, \cdots, c_{mn})
\end{aligned}
$$

Algorithm 1: Send Metadata Requests

```
1: channel ← METACHANNEL(host, port, pipelineCapacity)
2: protocolType ← FTP, GSIFTP, IRODS ...
3: channel.OPEN(protocolType)

4: request ← REQUEST()
5: request.AUTHENTICATE(channel, credentials)
6: dependent ← True  False
7: request.SETDEPENDENTCHAIN(dependent)
8: wait ← True  False
9: channel.SEND(request, wait)

10: request ← REQUEST()
11: request.LIST(path)
12: response ← channel.SEND(request, wait)
    if response.wait = True then
13:  PRINT(response)
```

Algorithm 2: SMURF Protocol Request

```
1: procedure AUTHENTICATE(channel, credentials)
2:     cmdInfo ← PACKCRED(credentials)
3:     cmd ← COMMAND(''auth'', cmdInfo)
4:     parser ← PROTOCOLPARSER(request).AUTHCMDPARSER(cmd)
5:     pair ← PAIR(cmd, parser)
6:     pairs.APPEND(pair)
7: end procedure

8: procedure LIST(path)
9:     cmdInfo ← PACKPATH(path)
10:    cmd ← COMMAND(''list'', cmdInfo)
11:    parser ← PROTOCOLPARSER(request).LISTCMDPARSER(cmd)
12:    pair ← PAIR(cmd, parser)
13:    pairs.APPEND(pair)
14: end procedure
```

Algorithm 3: SMURF Protocol Parser

```
1: procedure PARSE(reply)
2:     myreply ← READ(reply)
3:     globalData1 ← PARSEREPLY(myreply)
4:     request.SAVE(globalData1)
5:     globalData2 ← request.GET()
6:     cmdInfo ← PACKNEXT(reply, globalData2)
7:     nextCmd ← COMMAND(''next'', cmdInfo)
8:     nextParser ←
       PROTOCOLPARSER(request).CMDPARSERNEXT(nextCmd)
9:     request.ADDPAIR(nextCmd, nextParser)
10: end procedure
```

This expression resembles a matrix where the sender(s) and parser(s) of one request R_i have been defined as row-wise, and the order of processing the requests has been defined as column-wise. The operator $\circ$ still follows the aforementioned notion to define the order between the sender s and the parser p on the same row. Moreover, the overlapping executions of sending messages and parsing will happen across the requests, increasing the pipeline system throughput. If the protocol is stateful that requires the dependent relationship between the commands' sending and parsing, then the transfer of one request cannot be interleaved by other requests. In this scenario, the pipeline transferring cannot give outstanding performance, but the system can still increase the transfer performance via concurrency. Namely, the system establishes multiple isolated connections to the remote server and transfers metadata messages simultaneously. Concurrent transferring will be discussed in more detail in section 2.3.1.

To maximize the pipeline system's performance, the real-time stream transfer is preferable to the batch transfer. The new request should be put into the pipeline immediately for transferring as long as the pipeline capacity is not full. Meanwhile, one request should be removed from the pipeline system once it is completed successfully or aborted. This real-time design of the pipeline system still needs to guarantee that the parser ordering is consistent with the sender ordering, which will be discussed in section 2.2.2.

Algorithm 1 shows the pseudocode for sending the metadata requests. First, the client establishes the metadata channel to the remote server (lines 1-3) with the provided information, such as host address, port number, and the pipeline capacity. It also needs to provide the type of protocol to be used for metadata retrieval. Then an authentication request is generated with the given credentials (line 4). When sending the request, it is optional to specify that this client will do a blocking wait to complete authentication. Line 10 - 12 is to send the metadata request, namely, a listing request to retrieve the metadata content denoted by the resource path. Every metadata request can be sent in this convention, and the client can populate more requests into the metadata channel without waiting for the completion of the previous requests. Moreover, the metadata channel will automatically handle the commands' sending/parsing in the pipeline mechanism.

In practice, users can rewrite a protocol by customizing SMURF protocol request and parser functions (a.k.a., method overriding in object-oriented programming), as shown in Algorithms 2 and 3. In Algorithm 2, the SMURF protocol request library packs the command message (line 3) and assigns the user defined parser to parse this reply (line 4). One request maintains a chain of pairs (line 6), where each pair is organized in the format of $\{command, parser\}$. The request decides the sequence order of commands in this pair chain and can append more pairs as the independent relationship (line 6). In Algorithm 3, each SMURF protocol parser can design and implement its logic to read the reply from the remote server (line 2). Parsers of this request share the data variables via the request space (lines 3 - 5). One parser can define the next dependent $\{command, parser\}$ based on the current result and append this pair into the pair chain (lines 6 - 9).

2.2.2 Matrix Ordering Guarantees the Rule of "You Parse What You Send"

Matrix ordering abstracts the messages' sending/parsing orders inside the universal transferring stream. One request with the chain of pairs is expressed as a matrix column, where each pair $\{command, parser\}$ is a one-row element. Each row element also contains the information to specify whether this command will require the next row element's dependent relationship.

The system can guarantee the correctness of the pipeline sending and parsing over multiple requests. Moreover, this correctness comes from two facts: (1) The underlying transfer connection guarantees that the order of sending messages is the same as the order of receiving the replies; (2) The matrix ordering guarantees that the order of parser in each pair is the same as sending its command message.

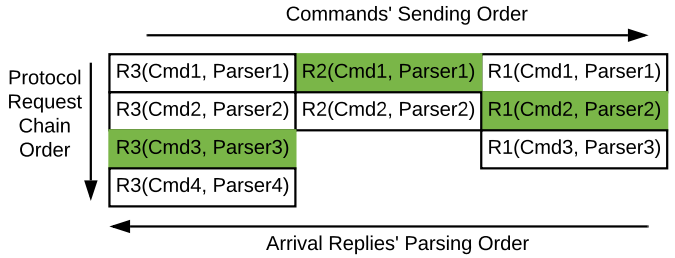

Fig. 3: One possible scenario of matrix ordering to send/parse the requests in pipeline. Green color denotes the request's inner cursor position to parse the arriving reply.

The system serves the message sending and parsing in parallel. The message sending is in Round Robin on the column order (as shown in Figure 3). We assume the request on the left-most column of the matrix will be served first without loss of generality. For example, followed by the sending of a command of R_1, the commands of R_2, R_3 will be sent. When a new request comes, it will be placed on the left-most column, and the first command Cmd_1 from its chain will be sent immediately. Each request maintains an internal cursor pointing to the pair whose parser will be invoked for its incoming reply. When a request's parser has completed the arriving reply's execution, it will check the dependent relationship of the next pair on the below row. This parser will send the next command, and meanwhile, this request will be removed from the current column position and appended to the right-most. If there is no dependent relationship, this request will send all of its commands, and its parsers will continue to parse the arriving replies. The matrix ordering is thread-safe and can be interleaved by multiple threads.

As shown in Algorithm 3, each parser has its own design and implementation to parse the incoming reply and decide the completion of parsing. Usually, this can be implemented as a state machine, where the transition of a state will be defined by the protocol's Request for Comments (RFC). For example, on retrieving the metadata of a filesystem folder containing millions of subfiles from a GSIFTP server, SMURF's transferring stream will receive a continuous intermediate part of metadata. The whole transferring will be terminated successfully when the parser can parse *code 250* of the reply. When one request's last parser has been completed, and then this request is finished. One request will be marked as success as long as its protocol commands have been sent and parsed correctly. Otherwise, this request will be regarded as a failure, either re-transferred or skipped according to parsers' results. One request failure will not block the next requests to be transferred over the same connection as long as this connection is not broken.

2.3 Fetch/Prefetch Services

SMURF-Cloud transfers the large-scale metadata using *concurrency* and *pipelining* and guarantees the reliability of the transfers in WAN. The details of sub-components and features are described below.

2.3.1 Metadata Transferring via a Cluster of Fetch/Prefetch

The dispatcher assigns the pending jobs to all available services in Round-Robin. One service will become available to process the next job as long as the service completes one fetch/prefetch job and sends back an ACK message to the queue dispatcher. When a service is terminated, the unacknowledged jobs will be re-dispatched to the rest of the available services. A fetch/prefetch service keeps at most one *singleton connection* to the remote server and serves up to C (pipelining capacity) fetch/prefetch jobs from the queue. If one established connection has been idle for a while, this connection will receive a TIMEOUT reply from a remote I/O node, and resources of this connection will be de-allocated from the service. Re-establishment of connection will be triggered by the next dispatch job and automatically handled by the Transferring Stream 2.2.

To fully exploit the computing power and network I/O bandwidth of the Cloud cluster, the Cloud backend deploys and launches multiple instances of fetch/prefetch services on a single cluster node. With N services running in the cluster, the Cloud establishes N TCP connections to remote servers and transfers N metadata requests concurrently. The Cloud backend controls the *concurrency* level by changing the number of fetch/prefetch service instances depending on the demand and the load. When the Cloud deploys services across M machines, the Cloud service can transfer N metadata requests from M nodes that exceed the single machine's limitation and bottleneck. Meanwhile, the Cloud service can tolerate the failure of $(M-1)$ nodes. The instances of services running on the failed nodes will be redeployed to the other available nodes.

2.3.2 Metadata Schema, Storage and Transfer Format

On the Cloud backend, the metadata is stored as a $\{$*key*, *value*$\}$ pair in a NoSQL database, where *key* is the hash value of the resource path and *value* is the metadata content in JSON-like format (schemaless data structure). The value consists of *Version*, *Attributes* and *Manifest of ID*. NoSQL database provides native storage and query support for JSON data. For example, the Cloud resolves the overwrite conflicts of metadata contents using the property of timestamp on remote I/O node as the "version". The underlying database clusters can guarantee the atomic read/write on the same metadata entry. Moreover, as long as the retrieval metadata's timestamp is newer than what has been cached, it is safe to overwrite it. Otherwise, the retrieved metadata with the stale timestamp will be discarded. The service will return currently cached metadata content to the Edge/Fog service that requested this metadata retrieval. On the Edge/Fog node, the metadata resides in the memory cache as ProtocolBuffer [33] objects. When transferring the metadata between the system nodes, the metadata content is encoded into bytestreams by ProtocolBuffer.

Attributes are the result of "list" command on a resource path. In file system, it contains access mode, ownership, size, timestamp and file name. When a "list" command has been executed on a directory, the result contains the attributes of all sub-files/sub-directories under this directory. We store the manifest of subfiles and subfolders as a part of a directory's metadata content. The metadata size of a directory is proportional to the number of subfiles/subfolders. The size of metadata can be as small as a few bytes or up to hundreds of megabytes. NTFS [34] allows the maximum number of files per directory to be 4,294,967,295. Amazon

S3 [28] sets the limits of kilobytes on the metadata size. The overhead of encoding, decoding, and transferring such large metadata content between the continuum cache layers will severely degrade the system response time and increase the user-perceived latency. Thus, the Cloud divides the large metadata object into fixed-size blocks and guarantees that each block object's size will not exceed the size limit. One block will be considered full when it contains more than 100,000 files. The large-sized metadata will be stored as a bunch of metadata blocks, and these metadata blocks will form into a logic tree structure, where the blocks of partial subfiles will be stored at the bottom as the leaf nodes. A manifest of sub-blocks will be stored as part of the metadata to locate the metadata blocks, and any metadata block can be accessed by a uniform resource identifier (URI).

The metadata block is stored in a distributed storage; thus, the services can access and update the metadata blocks in parallel. Moreover, the pipeline and concurrency transferring of the metadata blocks significantly increase the end-to-end system throughput between the hierarchical layers. This also benefits the prefetching since once the metadata block has been received and decoded, its content can be immediately available to the cache instead of waiting to transfer and decode the original metadata. Thus, the prefetch predictor can adjust its prefetching decision on time based on the real-time prefetch results. In our experiments, we set the block size to be 100,000 files per block and evaluate the average fetch latency and memory usage in Yahoo! Hadoop logs.

2.3.3 Directory Tree Structure Synchronization

SMURF provides a way to maintain the metadata consistency between Cloud and remote I/O nodes on the directory tree structure. It caches metadata content under the *key* of the request URL. If a folder has been renamed, deleted, or moved on the remote I/O node, then the subfolder metadata cached in SMURF will become dirty. If any metadata retrieval with force-refresh option has such an invalid path, the service will receive "No such file or directory" exception in the reply from remote I/O nodes.

Once a fetch/prefetch service gets such an error from a remote I/O node, then SMURF Cloud does *backtrace* synchronization to conservatively clean up the cached metadata under those invalid file paths. First, the fetch/prefetch service will try to read the currently cached metadata digest D. The atomic operation will compare and overwrite the "DELETE" status into current caching metadata D' if D is equal to D' without overwriting the metadata content of another success update D''. Finally, if this deletion has been successfully populated into Cloud DB, the deletion message will be sent to update all subscribed Edge/Fog servers which have fetched/prefetched on this invalid file path previously. Moreover, when deletion happens, the fetch/prefetch service will create a new fetch/prefetch request to do force-refresh on the parent file path and prefetch sublayer files (without force-refresh).

The subfile prefetching without a force-refresh option is to maximally reuse local cache and avoid redundant force-refresh retrieval of the cached metadata. If the parent file path is also invalid, then the fetch/prefetch service will repeat the above process to synchronize the metadata on the parent file path and increase the prefetch of subfolder layers by 1, e.g., prefetch 2-layer (prefetchTTL=2). The Cloud backend performs early-stop prefetch, which means such propagation of prefetching will terminate when a file path is valid or has not been cached yet.

2.4 Distributed Continuum Caching and Prefetching

The same metadata will be cached in the distributed layers-$\{1,2,3\}$, as shown in Figure 1. Our experimental analysis simulates the IoT network topology and assumes that the Cloud has unlimited storage space, and the Fog node can have a larger cache capacity than the Edge server. The system consists of the distributed continuum cache from Cloud to an Edge server, where the Cloud caches and stores all fetch/prefetch metadata, the Fog node caches partial Cloud metadata, and the Edge server only caches a small subset of the Fog metadata. The prefetch predictors can be installed on the Edge server and the Fog node with the judicious parameters to retrieve the locality metadata into each layer's local cache.

When the optional Fog node (layer-2) has been deployed between the Edge server and Cloud, this Edge server will fetch/prefetch metadata from the Fog node's local cache, which can be denoted as F_{edge} and $\{P_{edge}\}$. The Fog node can send the cached metadata back to the Edge server or forward the cache miss fetch request F_{edge} and prefetch requests $\{P_{edge}\}$ to the Cloud. The cache miss fetch request F_{edge} can cause the Fog node's prefetch framework (more details in 2.5) to consult its prefetch predictor on the aggressive prefetching $\{P_{fog}\}$. Thus there would be overlapping prefetch requests between $\{P_{edge}\}$ and $\{P_{fog}\}$ requests; however, the wait-notify queue (discussed in 2.4.1) will de-duplicate the overlapping prefetch requests to send them to the Cloud, and the Fog node will send back the Edge server's requested prefetch metadata $\{P_{edge}\}$ upon completion.

2.4.1 Layer Server's Request and Response Multiplexing

We design and implement a *wait-and-notify queue* to efficiently send and receive messages between the Edge server and the Cloud. The wait-and-notify queue consists of a sender thread and a receiver thread. Multiple worker threads can enqueue the requests and wait for the notification of completion concurrently. The sender thread allocates a unique context locally for each enqueued request and sends requests to the remote layers. The receiver thread extracts the context from the response and then uses this context to notify and wake up the waiting worker threads. Especially during one request R's sending and receiving, the similar queuing requests will be de-duplicated without sending them to the Cloud, and their worker threads will wait for the completion of R. The de-duplication is executed on the Edge/Fog layer to ease the Cloud's computing overhead and save the network bandwidth. The queue can also be configured to be in a "nowait" mode, which means worker threads do not wait for the completion. The wait-and-notifying queue mechanism exhibits high performance in multiple threading environments. Message sending and receiving are designed to be interleaved between multiple threads. Moreover, the queue has been implemented based

on a non-blocking queue, where *compare and swap (CAS)* strategy has been applied to improve the concurrent performance without the blocking synchronization. Note that the order of requests sending is not necessarily synchronized with the receiving order of the responses.

2.5 Prefetch Framework

SMURF employs a generic prefetch framework to apply the configurable prefetch predictor on Edge/Fog service. Users and system admins can easily configure and customize prefetch schemes for different types of applications. In this prefetch framework, each fetch request will be sent to the prefetch predictor to analyze and build a prefetch correlation relationship. For each fetch request, the prefetch framework maintains a cache miss counter and its metadata content in the cache with Least Recently Used (LRU) replacement policy. The cache miss counter denotes how many cache misses are on this request. The cache object can be evicted using the LRU replacement algorithm when the cache is full, and new metadata needs to be put into the cache. When the cache miss counter's value exceeds the threshold, the prefetch framework will consult the prefetch predictor for the potential prefetching candidates and execute aggressive prefetching on this request's correlation candidates. The prefetch framework checks whether each prefetch candidate exists in the current local cache. If there is a cache miss on this candidate, the prefetch framework will pack and send a prefetch request with the information (e.g., URI and priority). Prefetch framework does not maintain the cache miss counter for all the history requests since the essence of the LRU cache replacement algorithm is based on temporal locality, and the cache miss information of the coldest request will be replaced and cached out to reflect the temporal access locality and also save the memory usage.

2.6 Semantic Locality Prefetch Predictor

SMURF uses a novel prefetch predictor based on the directory semantic locality. The predictor uses a history window to predict the semantic locality of the trace log. This history window with the fixed window size stores the unique file path into segments. For one input file path, the predictor will find out the pattern of "A ? B" with the maximum matching number inside the history window, where "A" stands for the common prefix, "?" stands for one mismatch segment and "B" is the suffix, and sometimes this suffix can be empty. If the matching number exceeds the threshold, the predictor sends back this detected pattern to predict that the fetching file paths will follow this access pattern in the near future. When a request on a file path f causes a local cache miss, the predictor will check whether its pattern file path f_p object is cached or not. If it has not been cached, then the predictor will create an object of this pattern file path, put it into the local cache, and set the counter's value to one. If the pattern file path object has been cached, then the predictor will increase the cache miss counter by one. When this cache miss counter exceeds threshold T, the predictor decides to prefetch the correlation files of the pattern file path f_p and set the miss counter to zero.

If the pattern file path object has already been cached, the predictor will iterate the list of metadata of each subfile path f_{si} in the cache and send the prefetch requests of all cache missed subfile paths. In the prefetch request, the predictor can associate with the value of prefetch TTL, configured (by default, the value is 0) in the prediction property file. The number of prefetch TTL is to denote how many layers of subfiles to prefetch. Theoretically, the predictor can set an arbitrary large number to the value of prefetch TTL. However, upon completion of the prefetch on a file path, the queue system will automatically decrease the value of prefetch TTL by one and recreate a new prefetch request for each subfile and then re-queue those requests with the lower priority until TTL degrades to 0. In the competition of large-scale prefetching requests, higher priority prefetch requests will be given precedence and always preempt available prefetching services. It could be a large amount of lowest priority prefetch requests in the queue system, which can never be served or completed in a period and will be finally reclaimed and destroyed by queue cleaning. Section 3.4.2 discusses the prefetch settings of T and TTL.

TABLE 1: Specifications of the Cloud/Edge/Fog nodes in the experiments.

	Edge/Fog	Cloud	Remote IO
Location	Champaign, IL	Chicago, IL	Austin, TX
CPU	Intel Core i7-2600	Intel Xeon Gold 6126	Intel Xeon E5-2650 v3
RAM	32 GB	187 GB	62 GB
Disk	80 GB	210 GB	350 GB
OS	Ubuntu 16.04	Ubuntu 16.04	Ubuntu 16.04
# of nodes	5 KVM	5 Bare metal	1 Bare metal

3 Evaluation

We conduct our experiments over Yahoo! Hadoop grid trace logs from Yahoo! Webscope dataset [35]. This trace consists of more than 20 Million continuous daily metadata operations of the Hadoop name node throughout the year 2010. Geographical locations of the servers used in our experiments, the system settings and configuration are shown in Table 1. Round-trip-time (RTT) between Edge/Fog and Chameleon-UC is 8 millisecond and RTT between Chameleon-UC and Chameleon-TACC is 32 millisecond. Chameleon Cloud uses the dedicated 100G between Chameleon's UC and TACC sites. To simulate the heterogeneous remote I/O nodes, we setup FTP and iRODS servers at the Chameleon-TACC site and installed Globus Toolkit 6.0 [36] to configure the SimpleCA and GSIFTP server for the GSIFTP metadata transfers. The Minio [37] server is installed to simulate the Amazon S3 object storage service. In all experiments, SMURF protocol libraries are installed only on SMURF Cloud, deployed on the Chameleon-UC bare-metal cluster. SMURF's fetch/prefetch services are launched as Docker [38] services and managed under the Docker orchestration tool. Fetch/prefetch requests have been managed and dispatched by the advanced message queue protocol(AMQP) [39] broker. The Edge and Fog clusters are deployed into a Kernel-based Virtual Machine (KVM) cluster and configured with limited computing resources for more realistic experimental evaluation.

We organize our evaluation experiments as follows: Section 3.1 demonstrate the scalability of fetch/prefetch

TABLE 2: Yahoo! Hadoop log's 'list' command statistics.

Log Name	# of list cmds	unique file path	histogram=1
part-00000	4,750,645	49.72%	92.6%
part-00001	4,090,678	62.31%	92.98%
part-00002	3,732,058	62.52%	92.33%
part-00004	3,895,900	62.77%	91.85%
part-00005	4,148,414	54.23%	92.76%

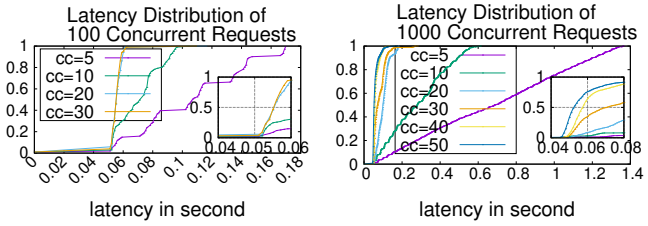

Fig. 4: Concurrent fetch latency distribution.

services by transferring the synthetic data without cache effects; Section 3.3 describes how we reconstruct the trace log of the real system in our experiments; Section 3.4 studies the prefetch schemes; Section 3.5 demonstrates the performance of continuum caching.

3.1 Scalability of Fetch/Prefetch Services

To emulate concurrent metadata transferring performance from remote servers, we use Yahoo! Cloud Serving Benchmark (YCSB) [40] to continuously send a large number of distinct requests from the client to the SMURF system and evaluate the latency distribution with the different number of Cloud fetch services. All the requests will be transferred along the Edge-Cloud I/O path with around 40ms accumulated RTT. In this experiment, we turn off the caching and prefetching effects in the testbed and configure the Edge cluster with different number of fetch services, and set the value of each Cloud fetch service's pipeline capacity to be 5. Figure 4 shows the latency measured on the Edge cluster and demonstrates that the latency distribution curves of 100 and 1000 concurrent requests with five Cloud fetch services are almost linear due to the queuing effect of Cloud, which means five Cloud fetch services are not sufficient enough to scale the number of concurrent requests. Thus, with more number of services in the Cloud, most of the requests can be made concurrently, and the latency of the majority of requests is within the small range between 40ms and 80ms.

Both fetch and prefech services use the aforementioned "universal metadata transfer stream" to retrieve the metadata from the remote I/O node. Each stream establishes one connection, thus we increase the level of concurrent connections (denoted as "CC") in Figures 4 and 5 by increasing the number of services. Figure 5 demonstrates the scalability of prefetching files metadata from heterogeneous I/O servers. We turn off caching effects in the testbed and let one Edge service initialize the sending of 100,000 distinct prefetch requests to the cloud, respectively, and calculated the average prefetching elapse time on the SMURF Edge side. We continue to increase the number of concurrency channels and each channel's maximum pipeline capacity until there are no noticeable performance gains. The scalability performance between the different protocols is similar,

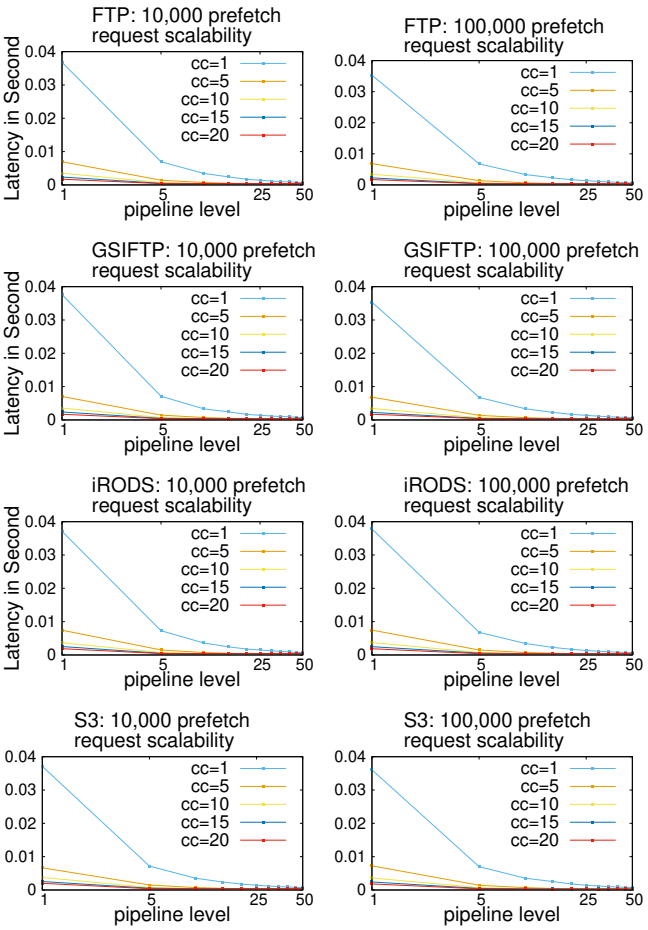

Fig. 5: Scalability of pipeline and concurrency.

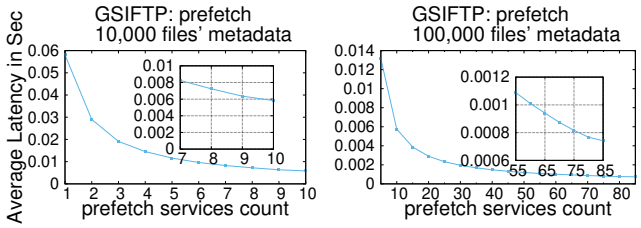

Fig. 6: Scalability of metadata prefetching from XSEDE-Comet@SDSC.

and the SMURF system can reduce the prefetching latency to 0.6 millisecond per request on average, which means the system can complete the prefetching of 100,000 metadata contents in 60 seconds.

We also evaluated the scalability of prefetching from XSEDE Comet [41] endpoints at San Diego Supercomputer Center (SDSC) in Figure 6, where the average prefetch latency per request is around 0.8 millisecond. Note that RTT between Cloud and XSEDE Comet is around 53ms and the transferring of files metadata over GSIFTP is conducted in the control channel by sending the command "MLSC". SMURF still supports FTP/GSIFTP data channel metadata transferring, which has been evaluated in our previous work [22].

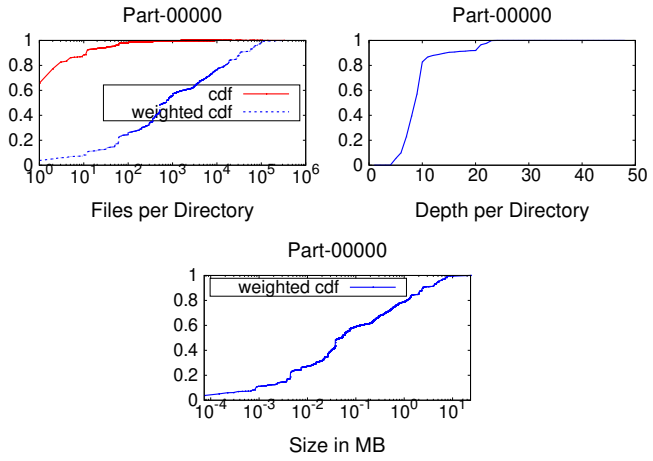

Fig. 7: Trace file system statistics in Yahoo! HDFS: (a) CDF and Weighted CDF of files per directory; (b) Distribution of files by directory depth; (c) Weighted CDF of metadata size per directory/file

3.2 Evaluation of Directory Tree Structure Synchronization

To evaluate the synchronization of directory tree structure between Cloud and the remote I/O node, we construct directory trees with different sizes m (i.e., the number of files/directories inside) and tree heights l (1, 5, 10), where each layer of directory tree contains $\frac{m}{l}$ files/directories. In experiments, we rename the root folder of one directory tree and fetch the metadata of file/directory on the longest path with the force refresh option to trigger the synchronization. It is trivial to know that the smaller size of directory tree costs the less synchronization time in total. In Figure 9, we consider to calculate the average synchronization latency per file/directory with the settings of tree size and height. For smaller size of directory trees, the height increases the latency (83 millisecond in the worst case) since it requires more rounds of metadata exchange between Cloud and remote I/O node. When the size (e.g., 10^4, 10^5) of tree is much larger than the height, the average synchronization latency per file/directory has been reduced up to 100 times (around 0.7 millisecond).

3.3 Trace File System Directory Tree Reconstruction

In trace logs, file path f always associates with types of operations, e.g., $open$, ls, $delete$, etc. Metadata read operations are directory tree idempotent (e.g., $open$ and ls) and will not change the directory tree structure on trace file system. The write operations (e.g., $mkdir$, $rename$ and $delete$) can change trace file system directory tree dynamically. Yahoo! Webscope dataset encrypts each segment of the file path into 27 bytes string. Thus the approximate directory tree size of each Hadoop trace log on disk will be more than 250GB.

We extract file paths from all types of operations for each audit log and construct them on the disk. This is the approximate emulation of trace file system directory tree structure in our prediction experiments. Figure 7 shows the cumulative distribution of directories by the number of files they contain and files by directory depth. This reconstructed file system's shape is flat: millions of files (nearly 90%) reside in the directories with a depth between 5 and 10. Most of the directories (around 95%) contain only a few files, and the majority of files (around 75%) are stored under a small portion of directories (about 3%), each of which contains more than hundreds of files and even up to hundreds of thousands of files. The metadata size ranges from few bytes to the maximum size of 25 MB, where the metadata size of the majority of files (around 80%) is less than 1 MB and nearly 3% files' metadata size is larger than 10 MB.

We extract requests containing the "listStatus" command from Yahoo! Webscope Hadoop audit trace logs in our experiments. In table 2, we statistically analyze the histogram of distinct file path in "listStatus" command. The histogram results show very skew access of "listStatus" metadata operation in Hadoop audit log: among the total number of four million "listStatus" operations, there are 50%-62% of unique file paths. The majority (92%) of unique file paths have been accessed only once, and only 8% of unique file paths contribute nearly half of total "listStatus" metadata operations. This skew access to behavior can cause prefetch predictors based on historical access sequence abysmal prediction rate. Their prediction rate is almost the same as that of LRU cache since most of their prefetch candidates are from history requests, but they will never appear again in the next "listStatus" requests, and the most frequent file paths will reside in the memory by the cache replacement policy.

3.4 Evaluation of Prefetch Schemes on Yahoo! Hadoop Grid Trace Logs

We conduct the experiments on the Edge node and replay the trace logs with different settings. The experiments have been evaluated on the Edge-Cloud I/O path (the abbreviation term "EC" in Figure 8) with the prefetch schemes installation on the Edge node. The cache size also has been taken into consideration, where the cache capacity on the Edge node has been increased by the percentage (10%, 20%, and 30%) of total requests in each trace log, and the average memory usages have been calculated by the oshi [42] software tool in Table 3. We also measure the average fetch time latency without the caching, and prefetching effects on the Edge node denoted as E (Edge I/O path) and EC in Figure 8). In the following sections, we will compare and discuss the prefetch scheme's performance on the criteria of cache hit rate, average fetch latency, and storage usages, respectively.

3.4.1 Cache Hit Rate

Figure 8(a) compares the cache hit rate between LRU cache and different prefetching schemes. Our directory locality scheme (denoted as "DLS") outperforms all other schemes on cache hit rate. It can achieve around 90%+ on all individual Hadoop audit logs because the directory locality scheme can successfully capture the access pattern of the "listStatus" operation in Yahoo! Webscope Hadoop audit log. AMP is another prediction scheme with a high prediction rate that can achieve around 65%+ accuracy. We train the AMP model on each day's trace and use that trained model for the next day's prefetching prediction. The AMP scheme's high prediction rate comes from the fact that there are many

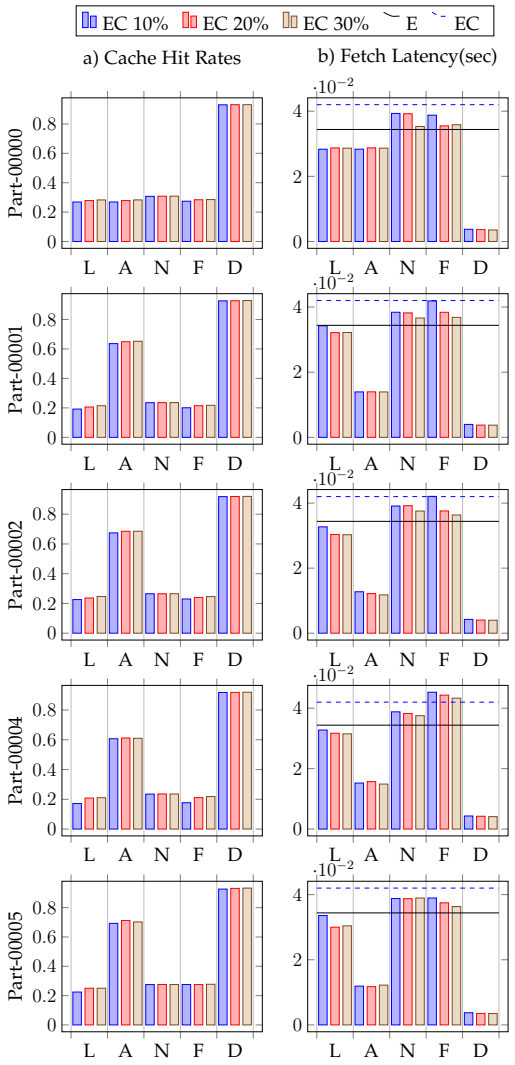

Fig. 8: Cache hit rate and average fetch latency between prediction schemes (**L**RU, **A**MP, **N**EXUS, **F**ARMER, **D**LS) on Yahoo! Hadoop trace.

overlapping file paths of "listStatus" metadata operations between successive days. The first day (Part-00000) of AMP performance has been set to the same value as that of the LRU cache since no previous day data is available. In Yahoo! Webscope datasets, day four (Part-00003) data is not available; hence, we use the day three (Part-00002) trained model for conducting the AMP performance prediction on day four trace. We also found that the cache hit rates of Nexus and Farmer are almost the same as that of LRU cache (below 25%) since the prefetching candidates suggested by the prediction schemes of Nexus and Farmer are all from the history requests. Simultaneously, the Hadoop audit log exhibits significantly skew popularity access in the "listStatus" metadata operation. Most of the "listStatus" operations execute on a file path once or occasionally, which inevitably causes the low prediction rate of prediction schemes based on history access sequence.

3.4.2 *Effects of T and TTL on Directory Locality Scheme*

In this section, we evaluate the effects of prefetching parameters: T and TTL with the different cache capacities on the Edge-Cloud I/O path. To evaluate the effects of Threshold

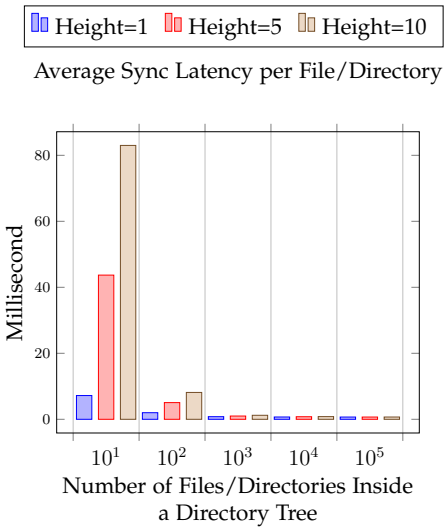

Fig. 9: Average sync latency per file/directory with the settings of size and height of directory trees.

TABLE 3: Prefetch Schemes Average Memory Usage (GB) on Edge Node.

Prefetch Scheme	10%	20%	30%
LRU	13	19	22
AMP	13	19	22
NEXUS	15	21	25
FARMER	15	21	25
DLS	13	20	22

T, the value of prefetch TTL has been set to 1. For each cache capacity (0.5%, 1% and 5%), we increase the value of Threshold T and calculate the average fetch latency and cache hit rate on the Edge node in Table 7 and 8. The results show that Threshold T = 3 gives better fetch latency when the cache capacity is relatively small (e.g., 0.5%, 1%). The smaller value of Threshold (T = 1, 2) causes an increase in the fetch latency because the cache with the smaller capacity is sensitive to the redundant prefetching results and such large amount of prefetch requests will occupy the network bandwidth and delay the transferring of fetching metadata. When the cache capacity has been increased to 5% (around 200,000 metadata entries), the smaller Threshold (T = 1, 2) provides the better performance both on the average fetch latency(nearly 50% improvement) and cache hit rate.

Accordingly, we set the value of Threshold T to be 3 and evaluate the effects of prefetch TTL in Table 9, 10. It is evident that the Prefetch TTL = 2 (i.e. 2-layer prefetchings) gives bad performance in fetch latency and cache hit rate. The smaller cache capacity causes the even worse results. The 2-layer prefetch does not match the workload pattern on the semantic directory tree structure. Moreover, the massive redundant 2-layer miss-prefetching can easily pollute the small cache. When the cache capacity has been set to 1%, the cache hit rate drops severely (under 20%). The fetch latency is ten times higher when TTL has been set to 2. The fetch latency is even higher than the accumulated RTT (40ms) on the Edge-Cloud I/O path because of the redundant prefetch results transferring over the network.

3.5 Performance of Continuum Caching

Based on the evaluation of prefetch schemes, we decide to choose DLS as the default prefetch scheme and repeat the experiments on the Yahoo! Hadoop trace logs to evaluate the continuum caching performance on the Edge-Cloud and Edge-Fog-Cloud I/O paths. The Edge node and the Fog cluster are deployed in the same site, and the distributed cache system is installed across all the Fog cluster nodes to provide the extra caching layer between the Edge node and the Cloud. We evaluate the average fetch latency and cache hit rate on the Edge node between two aforementioned I/O paths with the increasing continuum caching capacity in Tables 5 and 6 and the average system memory usage in Table 4, where on the Edge-Cloud I/O path we first set the relatively small cache size at 0.5% percentage of the total requests (around 20,000 metadata entries) on the Edge node and keep increasing its cache size to 10% until there are no obvious performance gains on the average fetch latency and cache hit rate. Accordingly, on the Edge-Fog-Cloud I/O path, we increase the Fog cluster cache size to the percentages of 1%, 5%, and 10%. It is clearly shown that the average fetch latency of the Edge node is significantly reduced when the system sets a relatively larger cache size on the Fog cluster. When the Edge node has been configured with the constant cache size, namely, 0.5% cache capacity, the Fog cluster caching and prefetching can reduce the Edge node average fetch latency up to 46% and slightly increases the cache hit rate. This result comes from the fact that the nearby Fog cluster can effectively cache and prefetch the demanded metadata shortly, and most cache-miss fetching requests on the Edge node will be directly retrieved from the nearby Fog cluster instead of the remote Cloud.

Compared with the EC settings, the average fetch elapse time has been delayed by the communication overhead between the Edge node and the Fog cluster but can be reduced by increasing the caching capacity on the Fog cluster. When the Fog cluster has been configured with 10% cache capacity, the average fetch latency of the Edge node with 0.5% cache capacity can be almost the same as that of the dge node with 10% cache capacity.

3.5.1 Average Fetch Latency

In Figure 8(b), we calculate the average fetch time between the prefetch schemes and measure the accumulated overhead of metadata transferring without any cache and prefetch installed. The setting of "E" (denoted with the solid horizontal line) shows the average latency of fetching performance that the edge node directly fetches metadata from the remote I/O server. The average fetching latency of the "EC" setting is denoted with the dashed horizontal line. The performance of LRU and all prefetch schemes are usually below the "EC" bar, which demonstrates that caching and prefetching is still an effective way to reduce the average fetching latency even with the low cache hit rates.

The prefetching scheme with a higher prediction rate can significantly reduce the average fetch latency since most metadata can be accessed locally. With the highest prediction rate (90+%) of the DLS prefetch scheme, the average fetching latency can be reduced to 0.004 seconds. The AMP (65+%) can achieve the average fetching latency of 0.015 seconds. Note that we have to use external storage to store the AMP training model, but the AMP model's high prediction rate can offset the overhead of database operations. The LRU with a larger amount of cache size can slightly reduce the average fetching latency. Nexus and Farmer's average latencies are relatively higher (above the solid bar). This may be due to two factors: 1) The RTT between client and remote I/O server is around 32 milliseconds, while the accumulated RTT of EC path is above 40 milliseconds; 2) Nexus and Farmer prediction rates are nearly the same as that of LRU cache, but there is extra computation to build relation graph on the fly. Moreover, the overhead of constructing and updating the relation graph in the Nexus and Farmer prefetch schemes is not ignorable.

TABLE 4: Edge Node Average Memory Usage (GB) with Increasing Cache Capacity.

Log Name	0.5%	1%	5%
part-00000	4.34	5.74	8.76
part-00001	3.07	4.41	7.79
part-00002	3.12	4.1	7.87
part-00004	4.15	5.35	8.31
part-00005	3.01	4.88	8.02

4 Related Work

Metadata prefetch prediction [43]–[45] studies developed different strategies to predict future requests as accurately as possible. NEXUS [44] applies a weighted-group-based prefetching algorithm to prefetch prediction. A weighted directed graph is built on the fly when the metadata server (MDS) receives requests from clients. The proposed polynomial time complexity algorithm looks up and analyzes requests in a predefined capacity history window. For a given access sequence, the history queue is populated with each request in the access sequence order. Each enqueued request is created as a vertex in this weighted relationship graph, where a directed Edge from any queuing request connects to this newly enqueued request. The weight of each Edge connection is calculated according to the successor relationship strength. For a given request, the prediction predictor looks up the graph to find out the directly connected vertices (requests) and predict top-k vertices with the largest Edge weight as the best prefetching candidates. Experiments show that their prefetching prediction can effectively reduce clients' average response time with reasonable overhead.

FARMER [46] investigates how a request's attributes information (e.g., "Host", "UserID", "ProcessID" and file path) can affect the file successor probability (the likelihood of successor being accessed if the predecessor has been accessed). The authors statistically analyze the average probabilities for the different trace sequences. They conclude that the same attribute will have a different successor probability between various traces. The access pattern without considering the semantic attributes is not sufficient to predict the file access probability. They apply a linear combination model to consider the combined effect of the history access sequence and the semantic attributes of requests. FARMER

TABLE 5: Edge Node Average Fetch Latency (milliseconds) Scalability with Increasing Continuum Caching Capacity.

Log Name	EC 0.5%	EC 1%	EC 5%	EC 10%	E 0.5% F 1%	E 0.5% F 5%	E 0.5% F 10%
part-00000	5.9	4.7	3.8	3.8	7.0	4.4	3.8
part-00001	6.4	4.8	4.0	4.0	6.2	4.5	4.4
part-00002	4.7	4.4	4.3	4.2	4.5	4.3	4.2
part-00004	5.4	5.0	4.4	4.3	5.2	5.0	4.7
part-00005	5.7	5.3	4.1	3.7	7.8	5.5	4.3

TABLE 6: Edge Node Cache Hit Rate (percentage) with Increasing Continuum Caching Capacity.

Log Name	EC 0.5%	EC 1%	EC 5%	EC 10%	E 0.5% F 1%	E 0.5% F 5%	E 0.5% F 10%
part-00000	83%	88%	93%	93%	77%	82%	84%
part-00001	85%	88%	93%	93%	79%	87%	88%
part-00002	89%	90%	92%	92%	88%	89%	89%
part-00004	84%	88%	92%	92%	83%	83%	84%
part-00005	78%	81%	88%	93%	73%	78%	77%

TABLE 7: Edge Node Average Fetch Latency (milliseconds) Scalability with the Effect of Threshold T.

Log Name	EC 0.5% (T = 1, 2, 3, 4)				EC 1%				EC 5%			
part-00000	7	6.3	5.9	7.3	5.7	5.1	4.7	5.8	3.7	3.9	3.8	4.1
part-00001	6.8	6.7	6.4	6.9	5.5	5.3	4.8	5.6	3.8	3.8	4.0	4.4
part-00002	5.4	5.4	4.7	5.5	4.9	4.6	4.4	5.1	3.6	3.5	4.3	4.6
part-00004	7.5	6.7	5.4	7.2	5.6	5.4	5.0	5.7	4.4	4.5	4.4	4.5
part-00005	8.1	7.1	5.7	6.7	6.1	5.6	5.3	5.9	4.2	4.3	4.1	4.4

TABLE 8: Edge Node Cache Hit Rate (percentage) with the Effect of Threshold T.

Log Name	EC 0.5% (T = 1, 2, 3, 4)				EC 1%				EC 5%			
part-00000	81%	84%	83%	82%	89%	89%	88%	86%	94%	94%	93%	91%
part-00001	87%	87%	85%	83%	88%	89%	88%	85%	94%	94%	93%	92%
part-00002	89%	90%	89%	85%	91%	91%	90%	87%	93%	93%	92%	90%
part-00004	80%	85%	84%	81%	89%	89%	88%	86%	93%	93%	92%	91%
part-00005	71%	73%	78%	75%	80%	82%	81%	79%	90%	90%	88%	87%

TABLE 9: Edge Node Average Fetch Latency (milliseconds) with the Effect of Prefetch TTL.

Log Name	EC 1% (TTL = 1, 2)		EC 5%		EC 10%	
part-00000	4.7	53	3.8	44	3.8	31
part-00001	4.8	49	4.0	38	4.0	27
part-00002	4.4	51	4.3	47	4.2	29
part-00004	5.0	52	4.4	43	4.3	28
part-00005	5.3	58	4.1	47	3.7	33

TABLE 10: Edge Node Cache Hit Rate (percentage) with the Effect of Prefetch TTL.

Log Name	EC 1% (TTL = 1, 2)		EC 5%		EC 10%	
part-00000	88%	15%	93%	29%	93%	37%
part-00001	88%	20%	93%	30%	93%	42%
part-00002	90%	18%	92%	31%	92%	38%
part-00004	88%	16%	92%	33%	92%	41%
part-00005	81%	14%	88%	27%	93%	36%

builds a relationship graph between predecessor and successors in a specific size history window similar to NEXUS and applies Integrated Path Algorithm (IPA) to detect the semantic attributes correlation between predecessor and each successor.

AMP [45] uses a different approach to predict the request pattern based on the analysis of the historical access sequence. The authors apply N-gram [47] model, which has been widely used in natural language processing, to train the prediction model in a quasi-online fashion (overnight training and use training result for the next day's prefetching prediction). AMP states that a 3-gram model can have more constraints on prediction and give more accurate predictions. They also claim that a 3-gram with up to 6 prefetching items can achieve a better hit ratio with less computation overhead.

Thrift [30] and gRPC [29] provide the high performance Remote Procedure Call (RPC) framework and employ an interface definition language (IDL) to compile the code written in a different programming language (e.g., C++, Python, Java, etc.). Their transport layer exchanges messages between client and server. Thrift and gRPC are client-server architectures and their framework requires the implementation of both client and server sides and deployment of their implementation to the I/O server or IoT end device. Without changing the server-side, there are challenges in the design and implementation. Our work provides the original and novel mechanisms to solve those challenges.

Hierarchical caching has been well studied in the literature. For the web caching systems, Wolman et al. [48] conducted their analysis on the hierarchical tree structure cache and evaluated the advantages and drawbacks of inter-proxy cooperation to demonstrate the performance benefits of cooperative caching. Sadeghi et al. [49] represented the popular tree hierarchical cache networks into a two-level network caching, where the network of caching nodes has been managed in a two-timescale approach. The researchers formulated the file transmission cost model as the Markov decision process (MDP) and propose a novel reinforcement learning (RL) to select the efficient caching policy to adapt to the dynamic evolution of file requests and caching policies of the network nodes. Jia et al. [50] considered a cached content placement problem in a hierarchical web proxies environment. The authors formulated the problem to minimize the data access costs by considering the distance between the source of requests and the closest destination with the requested data. Tran et al [51] introduced a novel cooperative hierarchical caching framework under the C-RAN [52] architecture. Inside the proposed framework, the complementary Cloud cache and Edge caches have been managed by a centralized controller at the Cloud. The authors evaluated the performance by configuring the cache installation on the different hierarchical layers. The experiments show that the proposed framework significantly outperforms traditional Edge-only caching schemes. SMURF employs a generic prefetch framework to apply the configurable prefetch predictor on Edge/Fog service. Users and system admins can easily configure and customize prefetch schemes for different types of applications.

5 Conclusion

This paper addresses two crucial IoT research challenges in accessing remote metadata: heterogeneity and scalability. We have presented a novel solution for efficient and scalable metadata access for distributed and heterogeneous applications across wide-area networks, called SMURF. Our solution combines novel pipelining and concurrent transfer mechanisms with reliability, provides distributed continuum caching and prefetching strategies to sidestep fetching latency, and achieves scalable and high-performance metadata fetch/prefetch services in the Cloud. We also studied the applicability of semantic locality in real trace logs, which is not well utilized in traditional metadata access prediction techniques, and implemented a novel prefetch predictor based on semantic locality. We compared it with three existing state-of-the-art prefetch schemes (NEXUS, FARMER, and AMP) on Yahoo! Hadoop audit traces. Our experimental results show that SMURF can achieve 90% accuracy during prefetch prediction and reduce the average fetch latency up to 50% compared to the other mechanisms.

Acknowledgments

This project is in part sponsored by the National Science Foundation (NSF) under award numbers OAC-1724898 and CCF-2007829. The results presented in this paper were obtained using the Chameleon Cloud and XSEDE resources.

References

[1] J. Towns, T. Cockerill, M. Dahan, I. Foster, K. Gaither, *et al.*, "Xsede: accelerating scientific discovery," *Computing in Science & Engineering*, vol. 16, no. 5, pp. 62–74, 2014.

[2] R. Pordes, D. Petravick, B. Kramer, D. Olson, M. Livny *et al.*, "The open science grid," in *Journal of Physics: Conference Series*, vol. 78, no. 1. IOP Publishing, 2007, p. 012057.

[3] J. Mambretti, J. Chen, and F. Yeh, "Next generation clouds, the chameleon cloud testbed, and software defined networking (sdn)," in *Proceedings of ICCCRI '15*, 2015.

[4] D. Duplyakin, R. Ricci, A. Maricq, G. Wong, J. Duerig *et al.*, "The design and operation of cloudlab," in *Proceedings of USENIX ATC*, 2019.

[5] DCMI Usage Board, "DCMI Metadata Terms," https://www.dublincore.org/specifications/dublin-core/dcmi-terms/, 2021.

[6] "Metadata," https://en.wikipedia.org/wiki/Metadata, 2021.

[7] S. A. Weil, K. T. Pollack, S. A. Brandt, and E. L. Miller, "Dynamic metadata management for petabyte-scale file systems," in *Proceedings of the 2004 ACM/IEEE conference on Supercomputing*, 2004, p. 4.

[8] D. S. Roselli, J. R. Lorch, T. E. Anderson *et al.*, "A comparison of file system workloads." in *USENIX Annual Technical Conference, General Track*, 2000, pp. 41–54.

[9] J. M. Wozniak, K. Chard, B. Blaiszik, R. Osborn, M. Wilde, and I. Foster, "Big data remote access interfaces for light source science," in *2015 IEEE/ACM 2nd International Symposium on Big Data Computing (BDC)*. IEEE, 2015, pp. 51–60.

[10] J. L. Schnase, D. Q. Duffy, G. S. Tamkin, D. Nadeau, J. H. Thompson *et al.*, "Merra analytic services: Meeting the big data challenges of climate science through cloud-enabled climate analytics-as-a-service," *Computers, Environment and Urban Systems*, vol. 61, pp. 198–211, 2017.

[11] R. Hai, S. Geisler, and C. Quix, "Constance: An intelligent data lake system," in *Proceedings of the 2016 International Conference on Management of Data*. ACM, 2016, pp. 2097–2100.

[12] C. Quix, R. Hai, and I. Vatov, "Gemms: A generic and extensible metadata management system for data lakes." in *CAiSE Forum*, 2016, pp. 129–136.

[13] I. G. Terrizzano, P. M. Schwarz, M. Roth, and J. E. Colino, "Data wrangling: The challenging journey from the wild to the lake." in *CIDR*, 2015, pp. 4–7.

[14] S. Chaudhuri and U. Dayal, "An overview of data warehousing and olap technology," *ACM Sigmod record*, vol. 26, no. 1, pp. 65–74, 1997.
[15] K. Ashton *et al.*, "That 'internet of things' thing," *RFID journal*, vol. 22, no. 7, pp. 97–114, 2009.
[16] D. K. Dhillon and R. S. Uppal, "Internet of things-making sense of the next mega-trend," *J. Netw. Comput. Appl*, vol. 67, p. 19, 2016.
[17] D. Evans, "The internet of things: How the next evolution of the internet is changing everything," *CISCO white paper*, vol. 1, no. 2011, pp. 1–11, 2011.
[18] "Storm," http://storm.apache.org, 2021.
[19] M. Zaharia, R. S. Xin, P. Wendell, T. Das, M. Armbrust *et al.*, "Apache spark: a unified engine for big data processing," *Communications of the ACM*, vol. 59, no. 11, pp. 56–65, 2016.
[20] A. Biem, E. Bouillet, H. Feng, A. Ranganathan, A. Riabov, O. Verscheure, H. Koutsopoulos, and C. Moran, "Ibm infosphere streams for scalable, real-time, intelligent transportation services," in *Proceedings of the 2010 ACM SIGMOD International Conference on Management of data*, 2010, pp. 1093–1104.
[21] R. Kohavi, R. M. Henne, and D. Sommerfield, "Practical guide to controlled experiments on the web: Listen to your customers not to the hippo," in *Proceedings of the 13th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, 2007.
[22] B. Zhang, B. Ross, and T. Kosar, "Dls: a cloud-hosted data caching and prefetching service for distributed metadata access," *International Journal of Big Data Intelligence*, vol. 2, no. 3, pp. 183–200, 2015.
[23] L. D. Xu, "Enterprise systems: State-of-the-art and future trends," *IEEE Transactions on Industrial Informatics*, vol. 7, no. 4, pp. 630–640, 2011.
[24] E. Rescorla *et al.*, "RFC 2812: HTTP over TLS," https://www.hjp.at/doc/rfc/rfc2818.html, 2021.
[25] J. Postel and J. Reynolds, "Rfc 959: File transfer protocol (ftp)," *InterNet Network Working Group*, 1985.
[26] G. Aloisio, M. Cafaro, and I. Epicoco, "Early experiences with the gridftp protocol using the grb-gsiftp library," *Future Generation Computer Systems*, vol. 18, no. 8, pp. 1053–1059, 2002.
[27] "The integrated rule oriented data system iRODS," http://www.irods.org/, 2021.
[28] M. R. Palankar, A. Iamnitchi, M. Ripeanu, and S. Garfinkel, "Amazon s3 for science grids: A viable solution?" in *Proceedings of DADC*, 2008, pp. 55–64.
[29] "Google remote procedure call gRPC," https://grpc.io/, 2021.
[30] "Apache thrift software framework Thrift," https://thrift.apache.org/, 2021.
[31] J. A. Moore, J. M. Johnson, S. F. T. P. Initiative *et al.*, "Transportation, land use and sustainability," 1994.
[32] W. Allcock, "Gridftp protocol specification," *GGF GridFTP working group document*, 2002.
[33] "Protocolbuffers," https://en.wikipedia.org/wiki/Protocol_Buffers, 2021.
[34] "Ntfs," https://en.wikipedia.org/wiki/NTFS, 2021.
[35] "Yahoo webscope," https://webscope.sandbox.yahoo.com/, 2021.
[36] "Globus-toolkit," https://github.com/globus/globus-toolkit, 2021.
[37] "Minio," https://min.io/, 2021.
[38] C. Boettiger, "An introduction to docker for reproducible research," *ACM SIGOPS Operating Systems Review*, vol. 49, no. 1, pp. 71–79, 2015.
[39] "Amqp," https://en.wikipedia.org/wiki/Advanced_Message_Queuing_Protocol, 2021.
[40] B. F. Cooper, A. Silberstein, E. Tam, R. Ramakrishnan, and R. Sears, "Benchmarking cloud serving systems with ycsb," in *Proceedings of the ACM symposium on Cloud computing*, 2010, pp. 143–154.
[41] "Xsede-comet," https://portal.xsede.org/sdsc-comet, 2021.
[42] "Operating system and hardware information," http://oshi.github.io/oshi/, 2021.
[43] J. Li and S. Wu, "Real-time data prefetching algorithm based on sequential patternmining in cloud environment," in *Industrial Control and Electronics Engineering (ICICEE)*, 2012, pp. 1044–1048.
[44] P. Gu, Y. Zhu, H. Jiang, and J. Wang, "Nexus: a novel weighted-graph-based prefetching algorithm for metadata servers in petabyte-scale storage systems," in *Sixth IEEE International Symposium on Cluster Computing and the Grid (CCGRID'06)*, vol. 1. IEEE, 2006, pp. 8–pp.
[45] L. Lin, X. Li, H. Jiang, Y. Zhu, and L. Tian, "Amp: An affinity-based metadata prefetching scheme in large-scale distributed storage systems," in *Proceedings of CCGRID*, 2008, pp. 459–466.
[46] P. Xia, D. Feng, H. Jiang, L. Tian, and F. Wang, "Farmer: a novel approach to file access correlation mining and evaluation reference model for optimizing peta-scale file system performance," in *Proceedings of HPDC*, 2008, pp. 185–196.
[47] P. F. Brown, P. V. Desouza, R. L. Mercer, V. J. D. Pietra, and J. C. Lai, "Class-based n-gram models of natural language," *Computational linguistics*, vol. 18, no. 4, pp. 467–479, 1992.
[48] A. Wolman, M. Voelker, N. Sharma, N. Cardwell, A. Karlin, and H. M. Levy, "On the scale and performance of cooperative web proxy caching," in *Proceedings of the seventeenth ACM symposium on Operating systems principles*, 1999, pp. 16–31.
[49] A. Sadeghi, G. Wang, and G. B. Giannakis, "Deep reinforcement learning for adaptive caching in hierarchical content delivery networks," *IEEE Transactions on Cognitive Communications and Networking*, vol. 5, no. 4, pp. 1024–1033, 2019.
[50] X. Jia, D. Li, H. Du, and J. Cao, "On optimal replication of data object at hierarchical and transparent web proxies," *IEEE Transactions on Parallel and Distributed Systems*, vol. 16, no. 8, pp. 673–685, 2005.
[51] T. X. Tran, A. Hajisami, and D. Pompili, "Cooperative hierarchical caching in 5g cloud radio access networks," *IEEE Network*, vol. 31, no. 4, pp. 35–41, 2017.
[52] I. Chih-Lin, J. Huang, R. Duan, C. Cui, J. Jiang, and L. Li, "Recent progress on c-ran centralization and cloudification," *IEEE Access*, vol. 2, pp. 1030–1039, 2014.

Bing Zhang is a research programmer at the National Center for Supercomputing Applications (NCSA). He received a BE degree in Computer Science from JiLin University, the MS degree from University at Buffalo, SUNY, and a Ph.D. in Computer Science and Engineering from University at Buffalo, SUNY, 2019. His research interests include High-performance network, network and protocol optimization, Distributed systems, computer networks, and Cloud systems.

Tevfik Kosar Tevfik Kosar is a Professor in the Department of Computer Science and Engineering at the State University of New York at Buffalo. He has received his Ph.D. in Computer Science from the University of Wisconsin-Madison in 2005. His main research interests include data-intensive distributed computing, big-data analytics, data-center networking and I/O optimization, performance and energy efficiency in IoT and Edge computing systems. Some of the awards received by Dr. Kosar include NSF CAREER Award, IBM Research Award, Google Research Award, LSU Rainmaker Award, LSU Flagship Faculty Award, Baton Rouge Business Report's Top 40 Under 40 Award, 1012 Corridor's Young Scientist Award, UB Senior Faculty Research Award, and IEEE Region-I Technological Innovation Award. Dr. Kosar is a Senior Member of the IEEE.