Neptune: Efficient Remote Communication Services for Cloud Backups

Yu Hua
Wuhan National Lab for Optoelectronics, School of Computer
Huazhong University of Science and Technology
Wuhan, China
csyhua@hust.edu.cn

Xue Liu
School of Computer Science
McGill University
Montreal, Quebec, Canada
xueliu@cs.mcgill.ca

Dan Feng
WNLO, School of Computer
Huazhong Univ. of Sci. and Tech.
Wuhan, China
dfeng@hust.edu.cn

Abstract—**In order to efficiently achieve fault tolerance in cloud computing, large-scale data centers generally leverage remote backups to improve system reliability. Due to long-distance and expensive network transmission, the backups incur heavy communication overheads and potential errors. To address this important problem, we propose an efficient remote communication service, called Neptune. Neptune efficiently transmits massive data between long-distance data centers via a cost-effective filtration scheme. The filtration in Neptune is interpreted as eliminating redundancy and compressing similarity of files, which are generally studied independently in existing work. In order to bridge the gap between them, Neptune leverages chunk-level deduplication to eliminates duplicate files, and approximate delta compression to compresses similar files. Moreover, in order to reduce the complexity and overheads, Neptune uses a locality-aware hashing to group similar files and proposes shortcut delta chains for fast remote recovery. We have really implemented Neptune. We examine the Neptune performance by using real-world traces of LANL, HP, MSN and Google. Compared with state-of-the-art work, experimental results demonstrate the efficiency and efficacy of Neptune.**

I. Introduction

Cloud computing is typically housed in data centers that consume a great deal of energy. To reduce energy consumption, data centers are often constructed in close proximity of energy sources and near locations where cooling is more natural and cheaper, such as rivers and oceans (e.g., for hydraulic power and water cooling). Such locations, however, can also be disaster prone. Many cloud computing applications require high degree of reliability and availability. Data centers and their backups are hence built in geographically dispersed manner, in order to prevent failures from disasters, such as earthquakes, tsunami and hurricanes. The unpredictable occurrence of disasters may destroy the entire datasets stored in a data center, e.g., as a result of severe network outages during super storm Sandy [1]. Therefore, large-scale cloud networks rely on regular remote backups to protect against the disasters. In general, long-distance network connectivity is expensive and/or bandwidth-constrained, making remote backups for massive data very costly in terms of both network bandwidth and backup time.

An intuitive and direct solution is to detect data redundancy in the backup data stream to reduce the amount of data actually transmitted. It is worth noting that, from a systems implementation perspective, it is important to distinguish between the *managed* and the *unmanaged* redundancy. The former is purposely leveraged by the system to support and improve availability, reliability and load balance. The latter, however, is system-unaware and invisible to the system. Due to its consumption of substantial system resources, the unmanaged redundancy becomes a potential performance bottleneck in cloud systems. The cost effectiveness and efficiency of remote backup, we argue, lie in the significant improvements on the effective backup throughput, which can be achieved by transmitting data difference, called delta. Hence, it is important to understand the properties of the massive data in the backup streams.

According to an International Data Corporation (IDC) study, the amount of information created and replicated is more than 1.8 Zettabytes (1.8 trillion Gigabytes) in 2011, while the amount of digital data produced will exceed 40 Zettabytes during the next eight years [2]. In a foreseeable future, this already staggering volume of data is projected to increase at an annual rate of more than 60%, much faster than the expected growth of network capacity. Moreover, IDC analysis also exhibits that nearly 75% of our digital world has a copy, i.e., only 25% is unique. Moreover, inexpensive storage and more powerful processors have resulted in a proliferation of data that needs to be reliably backed up. Network resource limitations make it increasingly difficult to backup a distributed file system on a nightly or even weekly basis.

Since the network bandwidth across cloud systems is often a performance-limiting factor, existing systems leverage data reduction techniques to reduce the unmanaged redundancy and improve the effective throughput. The most commonly used techniques include chunk-level deduplication and delta compression, whose goal is to prevent redundant data from being transferred. Deduplication schemes split files into multiple chunks (say, generally 8KB size), where a hash signature, called a fingerprint, uniquely identifies each chunk. By checking their fingerprints, duplicate chunks can be removed, while avoiding a byte-by-byte comparison and replacing identical data regions with references. Moreover, the delta compression compresses similar regions by calculating their differences [3], [4]. The rationale of delta compression comes from the fact that both sender and receiver contain a reference file that is similar to the transmitted file. Hence, we only need to transmit the difference (or delta) between the two files, which requires a significantly smaller number of bits. For a system backup, both sender and receiver generally possess a reference file that is similar to the transmitted file. Therefore, transmitting only the difference (or delta) requires a smaller number of bits and significantly improves the effective throughput.

We use the effective throughput to measure the perfor-

978-1-4799-3360-0/14/$31.00 © 2014 IEEE

mance of the long-distance backup schemes. It is the throughput for transmitting the non-redundant backup information. For example, if 10Mb data are transmitted in one second, the throughput is 10Mb/s. However, if 9Mb of them is redundant, the effective throughput is only 1Mb/s. Our real implementations demonstrate that the effective throughput increases from 2Mb/s to 286.27Mb/s via efficient data filtration. The data filtration in Neptune needs to compute a sketch of each non-duplicate chunk as a similarity measure. Sketches has the property that if two chunks have the same sketch they are likely near-duplicates. These can be used during backups to identify similar chunks. Moreover, instead of using a full index mapping sketches to chunks, Neptune uses a cache with sketches from a previous stream to obtain compression performance improvements to a full sketch index. For a remote backup, identical chunks are deduplicated, and non-duplicate chunks are delta compressed relative to similar chunks that already reside at the remote servers. We then compress the remaining bytes and transfer across the WAN to the destination. Specifically, this paper has the following contributions.

Comprehensive Filtration. Neptune offers comprehensive filtration between the source and the destination of a remote backup. In the source, Neptune eliminates duplicate files and compresses similar files. The remote transmission leverages a similarity detection technique to obtain significant bandwidth savings. *Neptune goes far beyond the simple combination of system-level deduplication and application-level similarity detection.* While the former can deduplicate exact-matching chunk-level data well, it fails to deal with files from the application's viewpoint, since the low-level chunks can not explicitly express the properties of application-level data. The latter only concentrates on the files themselves from the application's viewpoint, failing to capture and leverage the system-level characteristics, such as metadata and access patterns. Neptune, in fact, bridges the gap between them and delivers high performance.

Cost-effective Remote Backups. During remote backups, Neptune alleviates computation and space overheads. First, to reduce the scope of processing data, Neptune leverages semantic-aware groups by using Locality-Sensitive Hashing (LSH) [5], [6] that has a complexity of $O(1)$ and light space overhead. In order to improve the efficiency of delta compression, Neptune slightly looses the selection of base fingerprint by using top-k, rather than only one similar fingerprint in conventional approaches. The top-k approximate delta compression can identify more chunks to be delta compressed, thus significantly reducing the entire network overheads. Moreover, in order to support efficient remote recovery, we propose a shortcut scheme for delta chains. The shortcut scheme allows any given version to be restored by accessing at most two files from the version chain. Neptune hence avoids extra computation latency on the intermediate deltas and supports fast recovery.

Prototype Implementation and Real-world Evaluations. We have implemented all components of the Neptune architecture. We built a prototype to compute fingerprints and features, which are stored together in caching units, called storage containers. Moreover, we have implemented Neptune in multiple servers. We used 8KB chunk size and 4.5MB containers holding chunks, fingerprints and features. We examine the performance of Neptune by using multiple real-world datasets, including Los Alamos National Laboratory (LANL), HP, MSN and Google. We also compare Neptune with state-of-the-art work, including EndRE [7], Cluster-Based Deduplication (CBD) [8] and Stream-Informed Delta Compression (SIDC) [4].

The rest of this paper is organized as follows. Sections II presents the Neptune design. Section III presents the implementation details. We present the experiment setup and evaluation results in Section IV. Section V presents the related work. We conclude our paper in Section VI.

II. The Neptune Design

In this section, we present the Neptune design in terms of research backgrounds and practical operations.

A. Resemblance Measure

Resemblance measure is used to evaluate whether two files are approximate. In general, besides some slight modifications, say formatting, minor corrections, etc, these files have the same content. The quantitative resemblance is a number between 0 and 1. When the value is close to 1, the files are possibly the same. High resemblance value represents near-duplicate or approximate measure.

Definition 1: The resemblance $r(A,B)$ of two files, A and B, is defined as $r(A,B) = \frac{|S_A \bigcap S_B|}{|S_A \bigcup S_B|}$, where S_A and S_B respectively represent the fingerprint sets of files A and B. The hashing functions are chosen uniformly and randomly from a min-wise independent family of permutations.

To accurately represent similarity of files, we leverage a small and fixed-size sketch for each file. Sketches allow us to group a collection of m files into the sets of closely resembling files. Specifically, a sketch consists of a collection of the fingerprints of files. To compute the fingerprints, we leverage Rabin fingerprints [9] due to its ease of use and computation efficiency. Rabin fingerprints are based on polynomial arithmetic and can be constructed in any length. It is important to choose the length of the fingerprints so that the probability of collisions is sufficiently low. In practice 64 bits Rabin fingerprints are sufficient for most real-world applications.

Consider two files, A and B, that have resemblance ε. If ε is close to 1, the sketches S_A and S_B will be approximately pairwise equal. In order to identify duplicates, we divide every sketch into k groups and each group contains s elements. The probability that all the elements of a group are pair-wise equal is ε^s. The probability that two sketches have r or more equal groups is $P_{k,s,r} = \sum_{r \leq i \leq k} \binom{k}{i} \varepsilon^{s \cdot i}(1-\varepsilon^s)^{k-i}$.

Each file needs to maintain these k fingerprints that are called features. The probability that two features are equal is $\varepsilon^s + f$, where f is the collision probability. The initial value of threshold resemblance is ε_0.

In order to delta compress chunks, we need to identify a similar chunk that has already been transmitted and maintained in the destination servers. A resemblance sketch can help identify the features of a chunk. These features that will not change maintain the salient property, even if we add some small variations in the data. Moreover, to compute the features, we use a rolling hash function of 32-byte windows over all overlapping small regions of data. We choose the maximal hash value as the feature. By using multiple different hash functions, Neptune generates multiple features. Chunks that have one or more identical features are possible to be very similar [10].

In practice, to generate multiple independent features, we use the Rabin fingerprint over rolling windows w of chunk C and compare the fingerprints. We then permute the Rabin fingerprint to generate multiple values with randomly generated coprime multiplier with 32-byte windows.

We select the Rabin fingerprint as $feature_i$. In general, if the maximal values are not changed, we can achieve a resemblance match. We hence group multiple features together to build a "super-feature". The super-feature value serves as a representation of the underlying feature values. It has the salient property that if two chunks have an identical super-feature, all the underlying features will match well. The super-features help identify the similar chunks.

In order to obtain a suitable tradeoff between the number of features and the super-feature's quality, we performed a large number of experiments on the used datasets, including LANL, HP, MSN and Google. We use the variable numbers of features per super-feature and the super-features per sketch. We observe that increasing the number of features per super-feature will increase the accuracy of matches, while unfortunately decreasing the number of the identified matches. On the other hand, if we increase the number of super-features, the number of matches increases, however causing the increase of the indexing overheads. We typically identify that four features per super-feature can obtain the suitable tradeoff that exhibits good resemblance matches.

A resemblance lookup is executed in an index representing the corresponding super-features of previously processed chunks. We leverage each super-feature as a query request. Moreover, we consider the chunks are better if they can match on more super-features. This scheme is helpful to our approximate delta compression in the remote backups.

B. Practical Operations and Workflow

In the Neptune remote backups, the source needs to send the files to the destination, while consuming the minimum bandwidth via data filtration. The data filtration consists of deduplication and approximate delta compression. For the destination, it needs to reduce the space overhead and aggregate correlated files into groups to deliver high performance of local deduplication. The similar base chunk is important to the data filtration, which the source uses to encode and the destination uses to decode.

The workflow in Neptune consists of the procedures in both source and destination servers in the remote backups. Figure 1 shows the details of Neptune's workflow. When a local deduplication-enabled system receives files, these files are divided into content-defined chunks. A hash value is calculated over each chunk to represent it as a fingerprint. The fingerprint is then compared with an index of fingerprints of previously stored chunks. If the fingerprint does not exist in the index, it is new and should be stored. Otherwise, only a reference to the previous chunk is maintained in a file's metadata.

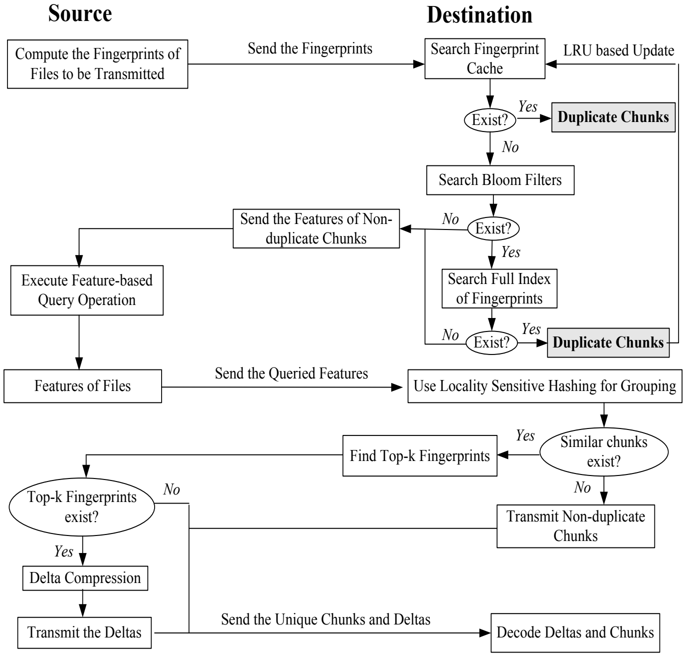
Source
Destination
Compute the Fingerprints of Files to be Transmitted
Send the Fingerprints
Search Fingerprint Cache
LRU based Update
Exist?
Yes
Duplicate Chunks
No
Search Bloom Filters
Send the Features of Non-duplicate Chunks
No
Exist?
Yes
Execute Feature-based Query Operation
Search Full Index of Fingerprints
No
Exist?
Yes
Duplicate Chunks
Features of Files
Send the Queried Features
Use Locality Sensitive Hashing for Grouping
Yes
Find Top-k Fingerprints
Similar chunks exist?
Top-k Fingerprints exist?
No
No
Transmit Non-duplicate Chunks
Yes
Delta Compression
Transmit the Deltas
Send the Unique Chunks and Deltas
Decode Deltas and Chunks

Fig. 1. The workflow of Neptune.

For the network transmission, the destination maintains the chunks and their fingerprints, which are loaded into a fingerprint cache. When a chunk from a source is requested for the transmission, its fingerprint is compared with the cache in the destination. If a miss occurs, a Bloom filter [11] is checked to determine whether the fingerprint possibly exists in the on-disk index. If so, the index is checked and the corresponding container's list of fingerprints is loaded into the cache. When an eviction occurs, based on a Least Recently Used (LRU) policy, all fingerprints from a container are evicted as a group based on the locality-sensitive hash computation. The destination further notifies the source to send the features of non-duplicate files.

The resemblance search in the groups can find the approximate top-k fingerprints that are most similar to the queried features. If the queried group contains the similar chunks, their fingerprints can be further sent to the source server to determine if they also exist in the source. Otherwise, the original chunks need to be sent to the destination server. When the source server contains the top-k similarity chunks, these similar chunks are selected as the base fingerprints. We execute the delta compression on these base fingerprints and the chunks to be transmitted. Finally, the required chunks and deltas in the source are sent to the destination servers.

To perform delta encoding, we use Xdelta [3] that is efficient to optimize the compression of highly similar data regions. We initialize the encoding by iterating through the base chunks, computing hash values at subsampled positions, and storing the hash and offsets. We then process the target chunks by computing their hash values at rolling window positions. In order to identify the matches against the base chunks, we retrieve the hash values in the index. If there is a match, we compare the bytes in the base and target chunks forward and backward from the starting position to create the longest match. Otherwise, if the bytes fail to match, we insert the target bytes into the output buffer, while adding this region to the hash index.

III. Implementation Details

In this section, we present the implementation details including correlation-aware grouping, data filtration and delta chain management.

A. Correlation-aware Grouping

In order to narrow the scope of processing data and alleviate the overheads, we leverage locality sensitive hashing (LSH) [5], [12] to map similar files into the same hash buckets with a high probability. The LSH function families have the locality-aware property. The files that are close to one another collide with a higher probability than files that are far apart. We define S to be the domain of files and $||*||$ to be the distance metric between two files. Let $\mathbb{H}$ be a family of hash functions and we can choose a function g from $\mathbb{H}$ uniformly at random. For any two points p and q, we study the probability that $g(p) = g(q)$.

Definition 2: LSH function family $\mathbb{H}$, is called (R, cR, P_1, P_2) sensitive for distance function $||*||$ if for any $p, q \in S$

- If $||p,q|| \leq R$ then $Pr_{\mathbb{H}}[g(p) = g(q)] \geq P_1$,
- If $||p,q|| > cR$ then $Pr_{\mathbb{H}}[g(p) = g(q)] \leq P_2$.

To allow resemblance identification, we choose $c > 1$ and $P_1 > P_2$. In practice, we need to widen the gap between P_1 and P_2 by using multiple hash functions. Distance functions $||*||$ correspond to different LSH families of l_s norms based on an s-stable distribution to allow each hash function $g_{a,b} : R^d \rightarrow Z$ to map a d-dimensional vector v onto a set of integers. The hash function in $\mathbb{H}$ can be defined as $g_{a,b}(v) = \lfloor \frac{a \cdot v + b}{\omega} \rfloor$, where a is a d-dimensional random vector with chosen entries following an s-stable distribution, b is a real number chosen uniformly from the range $[0, \omega)$, and ω is a constant.

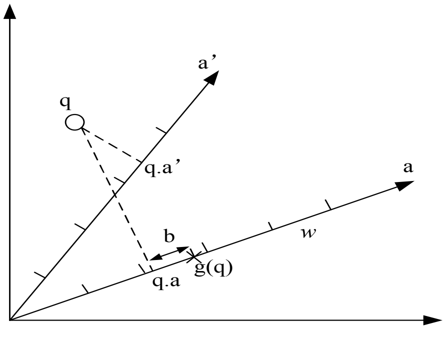

Fig. 2. An example of the LSH scheme hashing approximate points into the same bucket in hash tables.

Figure 2 shows an example to illustrate how the LSH scheme works in terms of measured distance and geometry results of hash functions. Specifically, LSH can determine the similarity between two points by examining the distance between them in a metric space. There exist two vectors a and a'. Given a vector a and query point q, $q \cdot a$ is their dot product. We uniformly choose b from the interval $[0, \omega)$. We can observe that $q \cdot a$ is the projection of point q onto vector a. From it, we get $g(q)$ with a shifted distance b. Since the vector a line is divided into intervals with length ω, each interval corresponds to a position sequence number of point q. In such a transformation, proximate points have a high probability of being located into the same interval.

Each file representation consists of multi-dimensional vectors, which are the inputs of LSH grouping. LSH computes their hashed values and locates them in the buckets of hash tables. Since LSH is locality-aware, similar vectors can be placed into the same or adjacent buckets with a high probability. We then select them from the hash buckets to constitute the correlation-aware groups and support resemblance retrieval.

Based on the LSH grouping techniques, we can execute the deduplication operations in the local servers. In order to further reduce the network bandwidth for data transmission, we leverage delta based policy for efficient compression.

B. Data Filtration

Data filtration in Neptune consists of chunk-level deduplication and approximate delta compression.

First, the deduplication has become a key component in the backup systems. By efficiently eliminating duplicate data, this technique can effectively improve the system efficiency. Specifically, the deduplication divides a data stream into variable-sized chunks and replaces duplicate chunks with pointers to their previously stored copies. A deduplication system identifies each chunk by its hash fingerprint. A fingerprint index is used to map fingerprints of the stored chunks to their physical addresses. In practice, due to the variable sizes of chunks, a deduplication system manages data at a larger unit, called container. A container is fixed-sized and becomes the basic unit of read and write operations. For a backup, Neptune aggregate the chunks into the containers to maintain the locality of the data stream. Moreover, for a restore, Neptune uses the container for efficient prefetching via updating algorithms, say Least Recently Used (LRU), to evict a container from the restore cache.

Second, the delta compression is designed as a faster and more efficient way to decrease the overhead of network transmission. It leverages the similarities between files and can create significantly small compressed files. Hence, in order to improve network transmission, transmitting only the difference (or delta) between two files requires a smaller number of bits. Delta compression uses a compressor that accepts two inputs. One is the current file to be compressed. The other is a reference source file. The delta compression operation needs to locate and copy the differences between the current and source files. A delta comes from compressing those differences. On the other hand, the decompressor leverages the delta and source files as the input to create an exact copy of the compressed file.

C. Delta Compression

In the remote backups, delta compression has the salient feature of bandwidth savings and efficient communications.

Definition 3: **Delta Compression.** We have two files f_{new}, f_{old}, and a client C and a server S connected by a communication link. C has a copy of f_{new} and S has a copy of f_{old}. The design goal is to compute a file f_δ, i.e., $f_{old} + f_{new} \rightarrow f_\delta$. In the meantime, S can reconstruct f_{new} from f_{old} and f_δ, i.e., $f_{old} + f_\delta \rightarrow f_{new}$. f_δ is called as a delta of f_{new} and f_{old}.

A differencing algorithm is able to identify the changes between two versions of the same file. A delta file (Δ) is the encoding of the output of the differencing algorithm. In order to create a delta file, Neptune uses as the input two versions of a file, including a reference file and a version file to be encoded. The output is a delta file that represents the modifications made between versions.

We represent the uncompressed ith version of a file by V_i. The difference between two versions V_i and V_j is indicated by $\Delta_{(V_i,V_j)}$. With respect to V_i, the file $\Delta_{(V_i,V_j)}$ differentially compresses and encodes V_j. We can then restore V_j by the inverse differencing operation on V_i and $\Delta_{(V_i,V_j)}$. We indicate the differencing operation by $\delta(V_i,V_j) \rightarrow \Delta_{(V_i,V_j)}$ and the inverse differencing (reconstruction) operation by $\delta^{-1}(\Delta_{(V_i,V_j)}, V_i) \rightarrow V_j$. V_i is created by modification of V_{i-1}.

A delta chain consists of a sequence of versions of the same file that continues $V_1, V_2, \ldots, V_{i-1}, V_i, V_{i+1}, \ldots$. In order to store this chain as a series of deltas, for two adjacent versions V_i and V_{i+1}, traditional approaches store the difference between these two files, $\Delta_{V_i,V_{i+1}}$. The "delta chain" is $V_1, \Delta_{(V_1,V_2)}, \ldots, \Delta_{(V_{i-1},V_i)}, \Delta_{(V_i,V_{i+1})}, \ldots$.

In order to reconstruct a version V_i, we need to apply the inverse differencing algorithm recursively for all intermediate versions through i. The related operations in fact generate a recurrence. V_i represents the contents of the ith version of a file and R_i is the recurrent file version. When $R_1 = V_1$, we can rebuild V_i and obtain

$$V_i = \delta^{-1}(\Delta_{(V_{i-1},V_i),R_{i-1}}). \tag{1}$$

The restore time for a version includes the time to restore all of intermediate versions. In general, the time grows linearly in the number of intermediate versions. For a multi-version system, the restore consumes too many system resources and incurs long restore delays, especially for the most remote version.

In order to improve the restore performance and reduce the operation delays, we propose a shortcut scheme for multi-version delta chains. The shortcut scheme leverages a minimum number of files for reconstruction. This delta chain consists of the modified forward deltas and an occasional whole file, $V_1, \Delta_{(V_1,V_2)}, \Delta_{(V_1,V_3)}, \ldots, \Delta_{(V_1,V_{i-1})}, V_i, \Delta_{(V_i,V_{i+1})}, \ldots$.

The shortcut chain has the benefit of allowing any given version to be reconstructed by accessing at most two files from the version chain. When a client executes the delta compression, the files transmitted to the server can be stored directly without additional manipulation.

Besides adjacent versions, Neptune needs to offer efficient and scalable delta compression for two non-adjacent versions, say, versions V_i and V_j. We use $|\Delta_{(V_i,V_j)}|$ to represent the size of $\Delta_{(V_i,V_j)}$. If $j-i$ increases, the size will increase, thus leading to potential decrease of the compression quality. Neptune hence uses an occasional whole file to limit the maximum version distance.

In general, two adjacent versions, V_i and V_{i+1}, have $\alpha|V_i|$ modified fractions between them. The parameter α represents the compression quality between adjacent versions. Neptune makes use of a differencing algorithm to create a delta file, $\Delta_{(V_i,V_{i+1})}$. The larger the size $\alpha|V_i|$ is, the better the compression performance is. The version compression is given by

$$V_i = 1 - \frac{|\Delta_{(V_i,V_{i+1})}|}{|V_{i+1}|}. \tag{2}$$

IV. Performance Evaluation

In this section, we present the experimental results of Neptune in terms of multiple performance metrics.

A. Experiments Setup

We have implemented Neptune via multiple servers. Each server has a 8-core CPU, a 32GB RAM, a 500GB 7200RPM hard disk and Gigabit network interface card. The Neptune prototype implementation required approximately 6000 lines of C code in a Linux environment.

Currently, since no large-scale data center traffic traces are publicly available, we generate patterns along the lines of traffic distributions in published work and open system traces to emulate typical data center workloads. To examine the system performance, we leverage 4 datasets, i.e., Los Alamos National Laboratory (LANL) dataset [13], HP file system trace [14], MSN trace [15] and Google clusters [16]. We describe the characteristics of real-world datasets. The measured traces are listed as follows.

First, Los Alamos National Laboratory (LANL) provides multiple datasets [13]. These datasets exhibit the properties of files' attributes. The entire dataset is about 19GB and consists of roughly 112 million lines of archive data and roughly 9 million lines of home/project space data. The attributes of the dataset contain unique ID, file sizes (in bytes), creation time, modification time, block sizes (in bytes) and the paths to files.

Second, HP file system provides a 10-day 500GB trace [14] and this dataset shows the accesses from 236 users. The dataset contains multiple operations, such as READ, WRITE, LOOKUP, OPEN, and CLOSE, on the accessed files with file names and device numbers.

Third, the MSN trace [15] includes the metadata information in a 6-hour period. The entire dataset has been divided into 10-minute intervals. This trace contains 1.25 million files and records 3.3 million "READ" and 1.17 million "WRITE" operations. The queried objects are the files that contain multi-dimensional attributes, including access time, the amounts of READ, the amounts of WRITE, operational sequence IDs and file size within an examined interval.

Fourth, Google provides anonymized log data from the clusters [16]. This dataset consists of a collected trace in a 7-hour period. The workload in the trace contains a set of tasks and each task runs on a single machine. Tasks consume memory and one or more cores (in fractional units). The trace contains 3,535,029 observations, 9218 unique jobs and

176,580 unique tasks. Each task belongs to a single job that has multiple tasks.

We randomly allocate the available data segments or snapshots among 128 servers in a round-robin way. Moreover, a client leverages the Neptune design for deduplication and approximate delta compression. A client captures the system operations from these traces and then delivers the requests to servers. Both clients and servers use multiple threads to exchange messages and data via TCP/IP.

Neptune can support top-k approximate delta compression and deliver high system performance. The k value determines the number of transmitted fingerprints from destination to source servers to identify similar fingerprints. In general, the larger the k value is, the more base fingerprints can be found, which unfortunately incurs the longer latency due to computation-intensive indexing. In order to select suitable k values, we attempt to examine the tradeoff between obtained bandwidth savings and execution latency. The used metric is the normalized rate that is the value of the saved bandwidth divided by the indexing latency. Specifically, we count the bandwidth against different k values and compute their normalized values between the minimum and maximum values. In the similar way, we compute the normalized latency values of executing top-k indexing. Figure 3 shows the normalized values when using different k values. We observe that the selected k values are respectively 4, 4, 5, and 6 for four used sets. These k values can obtain suitable tradeoff between bandwidth savings and indexing latency.

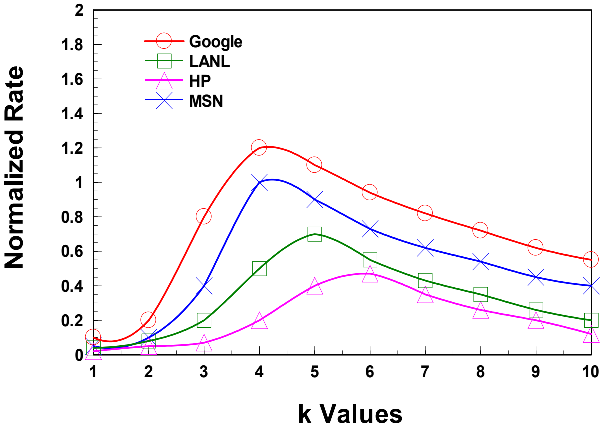

Fig. 3. Selection of suitable k values.

To the best of our knowledge, there are no existing approaches that can support both local deduplication and delta compression for remote communications. In order to carry out fair and meaningful comparisons, we respectively compare Neptune with these two aspects. For deduplication, we compare Neptune with state-of-the-art deduplication schemes, including EndRE [7] and Cluster-Based Deduplication (CBD) [8]. For delta compression, we compare Neptune with Stream-Informed Delta Compression (SIDC) [4] that is a salient feature of backup replication in the backup recovery systems. Moreover, due to no open source codes, we choose to re-implement EndRE [7], CBD [8] and SIDC [4]. Specifically, we implemented EndRE's end-host based redundancy elimination service, which includes adaptive algorithm (i.e., SampleByte) for fast fingerprinting and its optimized data structure for reducing cache memory overhead. We also implemented the components of CBD, including fingerprint cache, containers and super-chunk based data routing scheme in the deduplication clusters. SIDC was implemented, including its Bloom filter, fingerprint index, and containers, to load the stored sketches into a stream-informed cache.

All approaches for comparisons use the same running environments to examine their performance under different metrics. Specifically, the deduplication metrics include duplicate elimination, throughput and RAM usage. The duplicate elimination is defined as the percentage of duplicate data eliminated. The throughput is the rate at which the features of the streams are processed. The RAM usage comes from recording the space overhead of the grouping lookups. Moreover, the compression metrics include the effective throughput, multi-level normalized delta compression and delta overheads.

B. *Results and Analysis*

1) Deduplication Performance: We first present the experimental results in terms of local deduplication.

Filtration Elimination. Figure 4 shows the percentage of files duplicate elimination. Since EndRE and CBD carry out the exact-matching deduplication and work for fully identical files, they can remove on average 22.6% duplicate files. They, in the meantime, fail to detect and remove the files with slightly different files. Neptune eliminates the most duplicates (i.e., 92.7%). The reason is that Neptune not only identifies exact-matching chunks but also detects the similar chunks to obtain the larger percentage of the filtrated data.

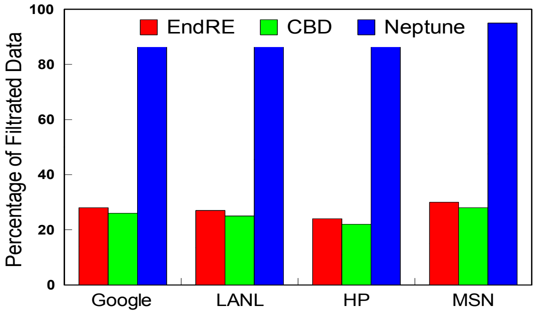

Fig. 4. Percentage of the filtrated data in multiple traces.

Deduplication Throughput. Figure 5 shows the results in terms of deduplication throughput. Specifically, Neptune achieves an average throughput of about 3.25GB/s, higher than 1.85GB/s in EndRE and 1.22GB/s in CBD. The substantial throughput improvements of Neptune attribute to the LSH based grouping scheme that can significantly narrow the scope of processing data, while alleviating the overheads and improving the deduplication throughput. Although EndRE optimizes data structures to reduce memory overhead, its fingerprinting hash table in practice consumes substantial space that is much larger than the limited memory size. EndRE hence have to frequently access to the not cached fingerprints in hard disks, which meanwhile tends to adversely decrease the throughput.

Space Overhead. We compare the space overhead as shown in Table I. The space overhead that is normalized to that of EndRE in terms of RAM usage. Specifically, EndRE has the highest RAM usage among the mentioned approaches

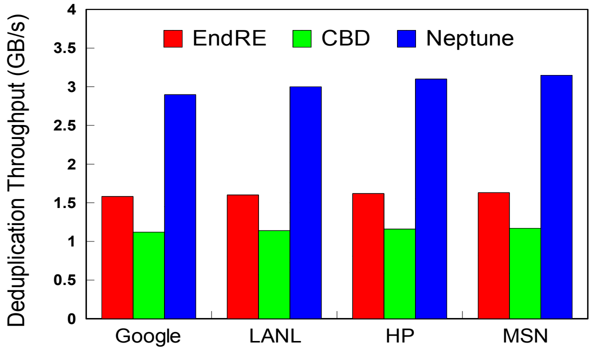

Fig. 5. Deduplication throughput.

since it carries out the exact deduplication that demands a large hash table in the memory to maintain the index of fingerprints. Although EndRE uses SampleByte algorithm to store compact signatures, it still requires at least 32 bytes for each new chunk, thus leading to a very large hash table for millions of fingerprints. Moreover, CBD needs to maintain the information for stateful routing that consumes much space overhead. Neptune leverages space-efficient features to significantly reduce the space overhead.

TABLE I. RAM USAGE NORMALIZED TO ENDRE.

	EndRE	CBD	Neptune
Google	1	0.56	0.15
LANL	1	0.82	0.19
HP	1	0.73	0.14
MSN	1	0.61	0.18

Since Neptune demands significantly less RAM space than other approaches, it can also improve the deduplication throughput by caching more index information into RAM. In fact, the Neptune system can be dynamically configured according to the users' requirements. For example, the throughput and duplicate elimination rates can be configured by tuning the proper system parameters (e.g., the number of indexed files in the cache, the similarity degree and the number of features, etc.). Therefore, from the system evaluation, Neptune is shown to offer the efficient and scalable deduplication performance, thus obtaining higher throughput and duplicate elimination at much lower RAM overhead.

Time Overhead. We examine the time overhead in completing the file deduplication and the experimental results are shown in Figure 6. EndRE leverages sample based fingerprinting algorithm to accelerate the deduplication. Neptune obtains the smallest time overhead due to the usage of the LSH computation to implement the fast and accurate detection of duplicates.

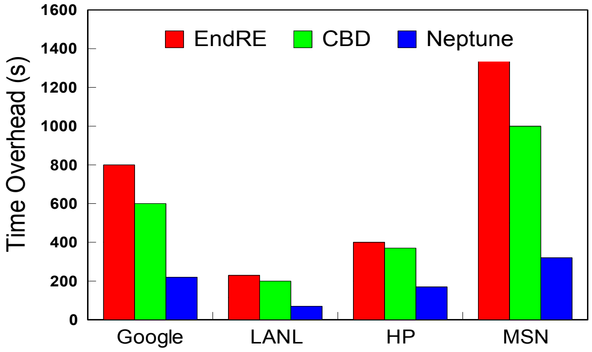

Fig. 6. Time overhead in the file deduplication.

2) Delta Compression in Remote Backups: We mainly report the experimental results from the real backups by examining the multi-level delta compression, effective throughput and delta overheads.

Multi-level Delta Compression. Our design goal is to reduce the amounts of the transmitted data by using the multi-level compression, as well as deduplication. Specifically, we examine the performance of chunk deduplication (only reduce the amounts of transmitted data and maintain the original data in local systems), standard delta compression and Neptune. The experimental results are shown in terms of the normalized compression in the cumulative form.

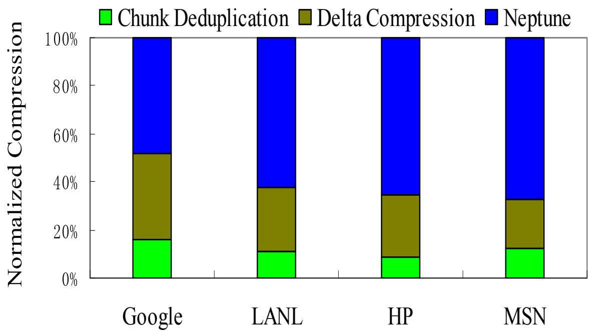

Fig. 7. Multi-level delta compression.

Figure 7 shows the compared compression results for the deduplication and compression options. The lowest region of each vertical bar is the amount of compression achieved by chunk deduplication, about 15.7% normalized compression. The next sets of colored regions demonstrate how much extra compression is achieved by using standard delta compression and Neptune. In all cases, using deduplication adds further compression, and by using standard delta compression, the normalized ratios are improved, on average 18.2%. Beyond existing compression ratios, Neptune obtains the better performance since it supports both deduplication and top-k approximate delta compression. Neptune can hence identify more data to be compressed.

Effective Network Throughput. We perform numerous remote communications experiments to measure effective network throughput between two remote cities. Their distance is more than 1200km via 2Mb/s network link. Figure 8 shows the results of the MSN dataset that has the largest size. The throughput runs at 2Mb/s and is measured every 20 minutes. We observe that compared with full replication (i.e., the baseline), the average effective throughputs in Neptune and SIDC are 286.27Mb/s and 57.26Mb/s respectively, much larger than the baseline, 1.57Mb/s. The main reason is that both Neptune and SIDC leverage the delta compression that physically transfers much less data across the network. Furthermore, SIDC replies on the closest fingerprint to determine the similarity. Performing the delta compression on a single base chunk limits the utilization and causes the decrease of communication performance. The reason is that SIDC often fails to identify similar chunks to be compressed. Unlike it, Neptune obtains better performance by finding more chunks that can be delta compressed.

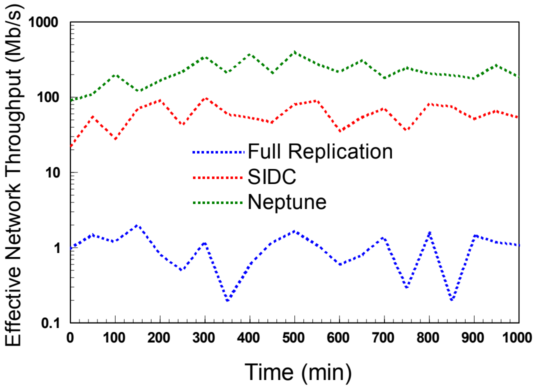

Fig. 8. Effective throughput in remote communications.

Delta Computation Overheads. Neptune leverages the delta compression to improve the effective throughput, which meantime incurs extra computation and disk I/O overheads. We examine these overheads in local servers and remote backup servers. First, the storage capacity overheads for maintaining fingerprints are relatively small. Each chunk stored in a container (after deduplication) has the fingerprints. The fingerprints are added to the metadata section of the storage container, which is less than 40 bytes. The disk I/O overhead is modest (around 4.5%). Furthermore, since the main overhead comes from the computation cost, we examine the real CPU utilization in both source and destination servers, as shown in Figure 9.

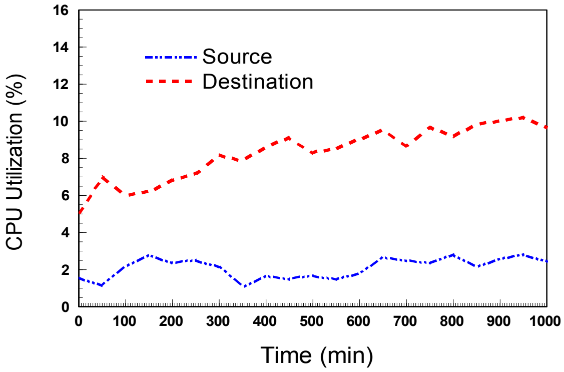

Fig. 9. Delta computation overheads.

We measure the CPU utilization over every 2 minute period after the initial seeding phase. In the source servers, the CPU utilization is around 2.47% to mainly identify top-k fingerprints and compute the deltas to be transmitted. We also observe that in the destination server, the CPU utilization demonstrates increasing trend, from 4.7% to 9.8%. The main reason is that the CPU overhead, i.e., indexing upon Bloom filters and fingerprint structure, almost scales linearly as the number of transmitted data. Overall, the entire CPU overheads are no more than 10%, which is acceptable in the servers, due to the increasing number of CPU cores. In essence, we trade computation and I/O resources for higher network throughput between remote network servers.

Recovery Overheads. The delta based recovery scheme can remotely restore the removed files with relatively small space and computation overheads. Compared with full backups, the proposed delta-based recovery scheme in Neptune can significantly reduce the system overheads. Note that since most encoding operations can be completed in an off-line manner, we mainly examine the decoding (i.e., recovery) time that is the main concern for users.

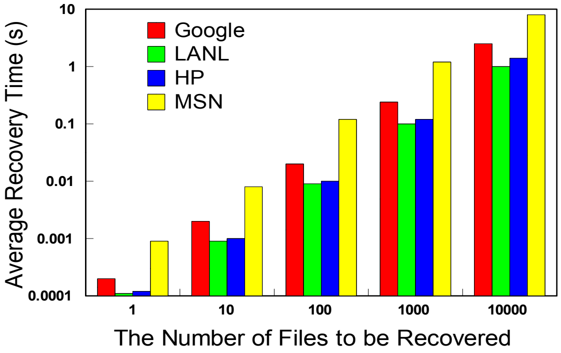

Fig. 10. Average recovery time.

Figure 10 shows the recovery time by examining the operation of decoding the deltas. We observe that the recovery time is approximately linear to the number of files to be recovered. Recovering 1,000 files requires about 1 second, which is generally acceptable to users. Neptune leverages the shortcut chains to allow any given version to be restored by accessing at most two files from the version chain. Neptune hence avoids extra computation latency on the intermediate deltas and supports fast recovery.

3) Neptune Assessment by Users: We argue that it is important and intuitive to evaluate new backup service by considering users' experiences after they use a new system design. Users' feedbacks often show some important aspects that may not be revealed by either simulations or implementations, and thus serve to complement the prototype-based evaluations.

We analyzed daily reports from cloud systems used by users during the second week of October 2012. Figure 11 shows how much time was saved by the users versus sending data without any compression. We reported the amounts of the transmitted data, network throughput and compression performance. We hence can calculate how long remote backups would take without compression. The median users would require 1025 hours to fully replicate their data (more than 6 weeks). With the aid of Neptune, the backups were reduced to 18.5 hours (saving over 1000 hours of network transfer time).

V. Related Work

Delta compression offers an efficient way to store or transmit data in the form of differences between sequential data rather than complete files. The features of delta compression are to reduce data redundancy and obtain substantial space savings. We need to select the most appropriate reference file(s) from a collection of files, if there is no obvious similar file. General delta compression tools are the copy-based algorithms based on the Lempel-Ziv approach [17]. These tools include vdelta and its newer variant vcdiff [18], the xdelta compressor used in XDFS [3], and the zdelta tool [19].

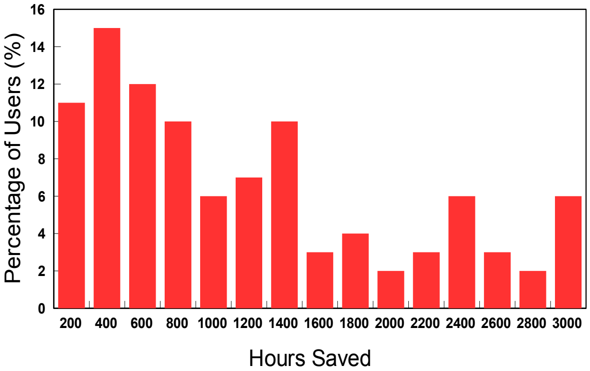

Fig. 11. Distribution of hours saved by users.

EndRE [7] uses an adaptive SampleByte algorithm for fingerprinting and an optimized data structure for reducing cache memory overhead. SIDC [4] proposes an architecture that uses stream-informed delta compression to already existing deduplication systems without the need of persistent indexes. Cluster-Based Deduplication (CBD) [8] examines the tradeoffs between stateless data routing approaches with low overhead and stateful approaches with high overhead but being able to avoid imbalances. NetStitcher [20] uses a network of storage nodes to aggregate not utilized bandwidth, and leverages a store-and-forward algorithm to schedule data transfers. Volley [21] performs automatic data placement across geographically distributed data centers, while reducing WAN bandwidth costs. Cimbiosys [22] offers a replication platform to allow each device to define its own content-based filtering criteria. By exploiting the skewness in the communication patterns, a tradeoff between improving fault tolerance and reducing bandwidth usage is obtained in [23].

Compared with the above schemes, Neptune shares the same design goal of reducing the amounts of the transmitted data between long-distance data centers. Unlike them, Neptune judiciously implements the deduplication in local servers and proposes a novel approximate delta compression to obtain significant bandwidth savings. Neptune also leverages shortcut delta chains to support fast remote recovery.

VI. Conclusion

This paper studied an efficient remote backup framework, called Neptune, which offers efficient cloud backup services. Neptune supports comprehensive data filtration via local deduplication and network delta compression. To improve the network transmission performance, Neptune uses approximate delta compression to identify more chunks to be compressed. The shortcut chains approach can further optimize the compression performance and alleviate the computation overhead in the recovery. Neptune has been implemented and thoroughly evaluated in a real remote backup system. Results demonstrate the benefits over state-of-the-art schemes including EndRE, CBD and SIDC.

Acknowledgements

This work was supported in part by National Natural Science Foundation of China (NSFC) under Grant 61173043, National Basic Research 973 Program of China under Grant 2011CB302301, NSFC under Grant 61025008, and NSERC Discovery Grant 341823, US National Science Foundation Award 1116606.

References

[1] J. Heidemann, L. Quan, and Y. Pradkin, "A Preliminary Analysis of Network Outages During Hurricane Sandy," *USC/ISI Technical Report ISI-TR-685*, November 2012.

[2] J. Gantz and D. Reinsel, "The Digital Universe in 2020: Big Data, Bigger Digital Shadows, and Biggest Growth in the Far East," *International Data Corporation (IDC) iView*, December 2012.

[3] J. MacDonald, "File system support for delta compression," *Master's thesis, EECS, University of California at Berkeley*, 2000.

[4] P. Shilane, M. Huang, G. Wallace, and W. Hsu, "WAN Optimized Replication of Backup Datasets Using Stream-Informed Delta Compression," *Proc. USENIX FAST*, 2012.

[5] P. Indyk and R. Motwani, "Approximate nearest neighbors: towards removing the curse of dimensionality," *Proc. STOC*, pp. 604–613, 1998.

[6] Y. Hua, B. Xiao, B. Veeravalli, and D. Feng, "Locality-Sensitive Bloom Filter for Approximate Membership Query," *IEEE Transactions on Computers*, no. 6, pp. 817–830, 2012.

[7] B. Aggarwal, A. Akella, A. Anand, A. Balachandran, P. Chitnis, C. Muthukrishnan, R. Ramjee, and G. Varghese, "EndRE: an end-system redundancy elimination service for enterprises," *Proc. NSDI*, 2010.

[8] W. Dong, F. Douglis, K. Li, H. Patterson, S. Reddy, and P. Shilane, "Tradeoffs in scalable data routing for deduplication clusters," *Proc. USENIX FAST*, 2011.

[9] M. Rabin, "Fingerprinting by random polynomials," *Technical Report TR-81-15, Aiken Laboratory, Harvard University*, 1981.

[10] A. Z. Broder, "On the resemblance and containment of documents," *Proc. Compression and Complexity of Sequences*, 1997.

[11] B. Bloom, "Space/time trade-offs in hash coding with allowable errors," *Communications of the ACM*, vol. 13, no. 7, pp. 422–426, 1970.

[12] Y. Hua, B. Xiao, D. Feng, and B. Yu, "Bounded LSH for Similarity Search in Peer-to-Peer File Systems," *Proc. ICPP*, pp. 644–651, 2008.

[13] Los Alamos National Lab (LANL) File System Data, *http://institute.lanl.gov/data/archive-data/*.

[14] E. Riedel, M. Kallahalla, and R. Swaminathan, "A framework for evaluating storage system security," *Proc. FAST*, pp. 15–30, 2002.

[15] S. Kavalanekar, B. Worthington, Q. Zhang, and V. Sharda, "Characterization of storage workload traces from production Windows servers," *Proc. IEEE International Symposium on Workload Characterization (IISWC)*, 2008.

[16] J. L. Hellerstein, "Google Cluster Data," *http://googleresearch.blogspot.com/2010/01/google-cluster-data.html*, Jan.2010.

[17] J. Ziv and A. Lempel, "A universal algorithm for sequential data compression," *Information Theory, IEEE Transactions on*, vol. 23, no. 3, pp. 337–343, 1977.

[18] J. J. Hunt, K.-P. Vo, and W. F. Tichy, "Delta algorithms: An empirical analysis," *ACM Transactions on Software Engineering and Methodology (TOSEM)*, vol. 7, no. 2, pp. 192–214, 1998.

[19] D. Trendafilov, N. Memon, and T. Suel, "zdelta: a simple delta compression tool," *Technical Report, CIS Department, Polytechnic University*, 2002.

[20] N. Laoutaris, M. Sirivianos, X. Yang, and P. Rodriguez, "Inter-datacenter bulk transfers with netstitcher," *Proc. SIGCOMM*, 2011.

[21] S. Agarwal, J. Dunagan, N. Jain, S. Saroiu, A. Wolman, and H. Bhogan, "Volley: Automated data placement for geo-distributed cloud services," *Proc. USENIX NSDI*, 2010.

[22] V. Ramasubramanian, T. Rodeheffer, D. Terry, M. Walraed-Sullivan, T. Wobber, C. Marshall, and A. Vahdat, "Cimbiosys: A platform for content-based partial replication," *Proc. USENIX NSDI*, 2009.

[23] P. Bodík, I. Menache, M. Chowdhury, P. Mani, D. Maltz, and I. Stoica, "Surviving failures in bandwidth-constrained datacenters," *Proc. SIGCOMM*, 2012.