Research Article

Energy Efficient Metadata Management for Cloud Storage System

Young Woong Ko, 1 Sun-Jeong Kim, 2 Jin Kim, 1 Eui-Jik Kim, 2 and Jung Min So 1

1 Department of Computer Engineering, Hallym University, 1 Hallymdaehak-gil, Chuncheon-si, Gangwon-do 200-702, Republic of Korea
2 Department of Convergence Software, Hallym University, 1 Hallymdaehak-gil, Chuncheon-si, Gangwon-do 200-702, Republic of Korea

Correspondence should be addressed to Jung Min So; jsol@hallym.ac.kr

Received 29 September 2014; Accepted 24 November 2014

Academic Editor: Damien Sauveron

Copyright © 2015 Young Woong Ko et al. This is an open access article distributed under the Creative Commons Attribution License, which permits unrestricted use, distribution, and reproduction in any medium, provided the original work is properly cited.

To effectively handle duplicate files, data deduplication schemes are widely used in many storage systems. Data deduplication algorithms reduce storage space by eliminating data to ensure that only single instance of data is stored in storage device. In this paper, we propose an energy efficient file synchronization scheme that provides hybrid data chunking using variable-length chunking (VLC) and fixed-length chunking (FLC). The main idea is to analyze similarities between old and new versions of data and decide which chunking method to apply in synchronizing the files. In particular, the proposed algorithm exploits the file similarity pattern for calculating the energy efficiency of chunking algorithms. We have developed an Android mobile application for file synchronization and measured energy consumption. The experiment results show that the proposed scheme helps save energy in synchronizing files, regardless of file types or amount of redundancies the files have.

1. Introduction

With the rapid penetration of smartphones, the necessity of synchronizing files between mobile clients and servers has significantly increased. Users synchronize their files through cloud storage systems such as Dropbox and Google Drive, so that they can access the updated files on any device. Many mobile cloud storage systems are in service, such as NDrive, Daum Cloud, UCloud, Dropbox, and iCloud. Among these systems, only Dropbox and iCloud synchronize files between client and server. Other systems treat the source file and modified file as two different files even if only a small portion of the file is changed, and thus the whole file has to be transferred through the network. As a result, transferring duplicated data blocks prevent implementing energy efficient data processing system. Energy efficient synchronization becomes a major issue when synchronizing files between a mobile client and its server, especially for large files such as mp3s, videos, and documents including high resolution images. Downloading or uploading these files through wireless interfaces such as Wi-Fi or 3G cause high energy consumption, reducing the usage time of smartphones.

When synchronizing files, it is obviously unnecessary to transmit data through the network if the data blocks already exist in both ends. It is quite possible that if a file is edited at the mobile client, most portions of the edited file remain the same at the server. In this case, it is a waste of energy to transmit the whole file to the server. Data deduplication techniques [1–4] such as FLC [2] and VLC [3] can be applied to discover redundancies in two different files and omit the redundant part when synchronizing the files. However, existing mobile storage cloud services do not go further than detecting file-level redundancies. In other words, they only detect if two files on the mobile client and the server are the same files or not.

FLC and VLC have their pros and cons, so which one is a better choice depends on the files. FLC allows files to be divided into a number of fixed-sized blocks, before applying hash functions to create a hash key of the blocks. The main limitation of FLC is the “boundary shift problem.” The main
problem of boundary shift is that inserting or deleting bytes in the middle of the data blocks will shift all the block boundaries following the modification point. This changes contents of the remaining blocks after the modification point. Thus, all subsequent blocks in the file will be rewritten and are likely to be considered different from those in the original file when adding new data to a file. Therefore, it is difficult to find duplicated blocks in the file, which degrades the deduplication performance. VLC is a more advanced approach that anchors variable-length segments based on their interior data patterns. This solves the boundary shift problem in the fixed-size block approach. VLC is also referred to as contents-defined chunking or content-based chunking, because the size of each chunk is determined by the content value. FLC has lower CPU usage and is faster but is possible to leave out redundancies which results in higher communication costs. VLC can always detect redundancies but has higher CPU usage and is slower compared to FLC.

In this paper, we apply the hybrid approach that supports multipolicy based on file similarity pattern. The key idea is to check similarity between a source file on the mobile client side and files on the server side with light-weight hash comparison algorithm. If we find a similar target file on the server, then we can select the best chunking algorithm by considering file pattern between the source file on the mobile client and the target file on the server. When selecting a chunking algorithm, the proposed system considers the energy efficiency between algorithms by predicting total energy consumption. In our work, by using multipolicy based approach, we can reduce overall data duplication capacity and increase the performance of the system. Therefore, our scheme is a kind of data deduplication method.

In this paper, we evaluate the energy consumption parameters for hash computation and data transferring on smartphones. The rest of the paper is organized as follows. In Section 2, we present the system design and implementation details. In Section 3, we report results from experiments. In Section 4, we discuss relevant related works. Finally, we conclude the paper with remarks on future work in Section 5.

2. System Architecture

Figure 1 shows the proposed system architecture. In the figure, send node is the mobile client and receive node is the server. The numbers indicate the order of synchronization procedures.

1. The sender calculates similarity hashes of the files it wants to synchronize and sends them to the receiver. (2) Based on the hash values, the receiver decides which strategy to use. If there is no file with high similarity, the receiver decides not to use deduplication. (3) If files with high similarities are found, the receiver analyzes redundancy pattern of the files and calculates expected energy consumption when using FLC and VLC. The receiver tells the sender what deduplication methods to use. The receiver also chunks files using the selected method. (4) The sender chunks the files according to the method chosen by the receiver and sends hash, offset, and size of the chunks to the receiver. The receiver finds the redundant chunks and copies the redundant chunks to the temporary file, in the proper offset. (5) The sender sends nonredundant chunks to the receiver. The receiver copies the received chunks to the temporary files to complete the synchronization.

2.1. File Similarity Evaluation Mechanism. In this work, we utilize the file similarity information that has two tuples, hash key and file offset information. With that information, we can easily find duplicated region on a file by comparing hash key between two files. If there is same hash key, we use corresponding file offset where we apply fixed-length chunking; otherwise, we apply variable-length chunking or skip data deduplication based on the ratio of representative hashes. So, total processing time of the proposed system is very short compared with variable-length chunking approach only.

The key idea of this paper is applying file similarity information to find duplicated points between two files. In this work, we have to decide how much duplicated data blocks exist between two files. As a fast and efficient file comparison mechanism, we exploit the representative hash list that is used for evaluating the degree of similarity between two files. We made representative hash list for a given file by searching and composing the maximum hash list.

As can be seen in Figure 1, Rabin hash function is used for computing a hash key for a block. The Rabin hash starts at each byte in the first byte of a file and over the block size of bytes to its right. If the Rabin computation at the first byte is completed, then we have to compute the Rabin hash at the second byte incrementally from the first hash value (fingerprint). Now since the hash value at the second byte is available, we use it to incrementally compute the hash value at the third byte and continue this process. We have to sort the Rabin hash value and choose only 10 maximum values as a representative hash.
2.2. Anchor-Based File Similarity Extraction. In this paper, we propose a method for calculating similarity hashes only on the anchor positions, for the sake of reducing computation overhead. An “anchor” is a byte pattern used as a delimiter in dividing a byte stream into chunks. Most systems that use variable-length chunking divide a byte stream based on anchors. For example, if the anchor is chosen to be "0x00," the probability will be 1 out of 256. If the anchor is "0x0000," the probability will be 1 out of 65536. If a byte pattern with 1/16384 probability is used as the anchor, it is expected that one anchor pattern appears in each block with size 16384 bytes.

If probability of anchor appearance is low, number of hashes created for each file will be smaller, which results in reduced computation time. On the other hand, if the probability is high, number of hashes will become large and will require more computation time. Previous similarity hashing schemes use Rabin fingerprinting scheme to hash continuous blocks, but anchor-based hashing does not need to hash continuous block and thus can use MD5 or SHA1 hash for fixed-sized byte stream. Figure 2 shows how we can generate representative hash list using anchor values. In this work, we first find anchors from the file stream and divide chunks between anchors. By calculating and sorting hash values, we choose top 10 or 20 hash values as representative hashes.

The proposed similarity hash extraction algorithm is shown in Algorithm 1. First, we set offset to be 0 and length to be the total length of the file. For each loop, a byte is read from the file stream. If both the current byte and the previous byte have remainder 1 when divided by 128, then an anchor is found. This will make the probability of anchor appearance 1 out of 16384. Once an anchor is found, a hash is generated for the block. If the generated hash value is smaller than MinHash, it is not saved in the HashArray. Otherwise, the hash value and the offset are saved in the Hasharray. Whenever a new hash value is inserted, the Hasharray is sorted so that the front of the array has the minimum hash value.

Algorithm 1: Pseudocode for extracting similarity hash values.

```
AnchorBaseExtractSimInfo()
Input: FileStream
Output: Hasharray
begin
  offset ← 0
  length ← Length(FileStream)
  chold ← −1
  while Read(chnew, FileStream) != EndOfFileStream do
    if (chnew MOD 128 == 1) AND (chold MOD 128 == 1) then
      block ← ReadBlock(blocksize, FileStream)
      hash ← Digest(block)
      if MinHash < hash then
        Hasharray[0].value ← hash
        Hasharray[0].offset ← offset
        quicksort(Hasharray)
        MinHash ← Hasharray[0].value
        chold ← −1
      else
        chold ← chnew
  return Hasharray
end
```

2.3. Analysis of Energy Consumption. Figure 3 shows energy consumption of hash calculation for different file sizes. File size was varied from 20 MB to 100 MB. For 20 MB file, 350 mJ of energy was consumed, whereas, for 100 MB file, 2048 mJ of energy was consumed. Intuitively, energy consumption is proportional to file size. Thus, when we estimate energy
consumption of computing file similarity, we can multiply file size (byte) to energy consumption per byte (J/byte), to get the result.

Figure 4 shows an energy consumption of FLC and VLC when synchronizing a file that has no redundancy. In the figure, “dedup” refers to energy consumed for conducting data deduplication and “transfer” refers to energy consumed for data transfer. Basically, energy consumption is linearly proportional to file size, and VLC shows significantly higher energy consumption for deduplication compared to FLC. Based on the measurements, we can see that file size is linearly proportional in energy consumption, so we can compute the energy consumption of synchronizing a file in the following way.

\[
\begin{align*}
(i) \ E_{\text{sim},f} &= P_f \times E_{\text{sim},\text{byte}}. \\
(ii) \ E_{\text{FLC},f} &= P_f \times E_{\text{FLC},\text{byte}}. \\
(iii) \ E_{\text{VLC},f} &= P_f \times E_{\text{VLC},\text{byte}}. 
\end{align*}
\]

In the equations, \(E_{\text{sim},f}\), \(E_{\text{FLC},f}\), and \(E_{\text{VLC},f}\) are the energy consumptions of checking similarity of a file, running FLC for the file, and running VLC for the file, respectively. Also \(E_{\text{sim},\text{byte}}\), \(E_{\text{FLC},\text{byte}}\), and \(E_{\text{VLC},\text{byte}}\) are the energy consumptions of checking similarity of a byte, running FLC for a byte, and running VLC for a byte, respectively. \(P_f\) is the size of the file in bytes.

Figure 5 shows the energy consumption of FLC and VLC for different redundancy ratio. While transfer cost is directly related to duplication ratio, cost of running FLC and VLC is not affected.

2.4. Energy Efficient File Synchronization Algorithm. The measurements in the previous section show that energy consumption of running FLC is lower than VLC. However, FLC may not find redundant blocks due to the “byte shift problem.” If new data is inserted in a file, FLC only detects redundancy until the new data block and fails to detect redundancy that exists behind the new data block. In Figure 6, a data block is inserted in the source file. The new data block shifts the boundaries of chunks behind the new block, causing FLC to fail in detecting redundant chunks. Although similarity of the two files is very high, FLC can only detect small portion of redundancy due to this problem. With VLC, performance of redundancy detection matches that of similarity check because VLC can detect all the redundant blocks based on anchor values.

Once we can estimate how much redundancy FLC can detect, we can choose which deduplication technique is better in terms of total energy consumption. Redundancy detection rate of FLC can be estimated using locations of similarity hash, as in Figure 7. In the figure, black dots are similarity hashes of the two files. Comparing the offsets of the hashes, we can conclude that FLC can detect redundancy up to the point where offsets of the similarity hashes are the same. Thus, we can estimate the energy consumption when using FLC.

In our approach, we can estimate the file similarity using representative hash list. The ratio of same hash values between the client hash list and the server hash list is file similarity values. For example, suppose that a file on the client and the server has 20 representative hash values. If the number of same hash values between the client and the server is 12, then it means that file similarity is roughly 60% between two files.

Algorithm 2 is the algorithm for selecting the file synchronization scheme. The inputs to the algorithm are ModifyHasharr, TargetHasharr, and ModifyFileLength. ModifyHasharr is the set of hash values of the modified file (that the client sends to the server), whereas TargetHasharr is the hash values of the reference file (previous version of the file). Initial status of each array is NULL that means there are no elements in it. ModifyFileLength is the size of the modified file. First, the algorithm finds the position where the first change has occurred. This will determine the energy consumption and transfer cost of FLC. Then, the algorithm calculates the total cost of FLC, VLC, and FTP (no data deduplication) to determine which strategy is the best. The algorithm returns the best strategy and its estimated energy consumption.

3. Performance Evaluation

We have implemented the algorithm and conducted experiments to test the feasibility of the proposed algorithm. For the server, we have used a PC with 3.0 GHz dual-core CPU clock and 1024 MB RAM, running Window 7. For the mobile client we have used Samsung Galaxy S phone running Android OS. Figure 8 shows the processing steps in GUI.

Figure 9 shows the energy consumption when synchronizing 500 MB of MP3 files, 700 MB of video files between client and server using Wi-Fi. In this figure, all files are new and the server does not have any redundant data. The energy consumption of FTP, FLC, VLC, and proposed algorithm is 610 J, 620 J, 820 J, and 623 J. It is obvious that FTP works the best in this case. The proposed algorithm consumes energy slightly higher than FTP and FLC.

Now we have saved all the mp3 files in the server and ran the experiments again. Figure 10 shows the result. The energy consumption of FTP, FLC, VLC, and the proposed algorithm is 610 J, 364 J, 566 J, and 368 J, respectively. FLC and VLC all
consume less energy compared to FTP, taking advantage of redundancy. In this case, FLC is the best choice because the same files are stored on the server, and thus there is no byte shift problem. The result shows that the proposed algorithm successfully chooses to use FLC.

Figure 11 shows the result of a case where a tar file containing Linux kernel source is synchronized. The client tries to synchronize the file “linux-3.1.7.tar,” while the server has “linux-3.1.6.tar,” which is the previous version. The similarity between the two files is 62%. VLC finds all redundancy and removes 62% of data, but FLC can only find less than 1% redundancy. Energy consumption for FTP, FLC, VLC, and proposed algorithm is 433 J, 443 J, 153 J, and 162 J, respectively. VLC consumes the least amount of energy, and the proposed algorithm pays a slightly higher cost due to similarity check. FLC consumes more energy than FTP, because the savings from transfer energy are less than the cost paid for redundancy detection.
Algorithm 2: Synchronization scheme selection algorithm.

```
CalcMinEnergy()
Input: ModiHasharr, TargetHasharr, ModifyFileLength
Output: Pattern
begin
    ChangePosition ← ModifyFileLength
    Similarity ← CompareSim(ModiHasharr, TargetHasharr)
    for i ← 0 to ModiHasharr.Length do
        isEqual ← false
        for j ← 0 to TargetHasharr.Length do
            if ModiHasharr[i].hash = TargetHasharr[j].hash then
                is Equal ← true
                shift ← ModiHasharr[i].offset − Target Hasharr[j].offset
                break
            if isEqual ← false OR shift != 0 then
                ChangePosition ← OriHasharr[i].offset
                break
        FLCEnergy ← ModifyFileLength * BytePerFLCEnergy
        RemainPortion ← ModifyFileLength − ChangePosition
        A ← FLCEnergy + RemainPortion * BytePerTransferEnergy
        VLCenergy ← ModifyFileLength * BytePerVLCenergy
        RemainPortion ← ModifyFileLength * Similarity
        B ← VLCenergy + RemainPortion * BytePerTransferEnergy
        C ← ModifyFileLength * BytePerTransferEnergy
        Pattern = <B ? (A < C ? FLC: FTP): (B < C ? VLC: FTP)
        MinEnergy = <B ? (A < C ? A: C): (B < C ? B : C)
        Return Pattern, MinEnergy
end
```

Figure 12 shows the result when multiple document files are synchronized. The total amount of data is 300 MB. Among the data, 100 MB consists of files that are totally redundant files, which means the same version of files is stored in the server. Other 100 MB consists of files that are stored in the server, but the client has modified the files slightly. The final 100 MB of data are new data, which the server does not have. The energy consumption is 150 J, 104 J, 101 J, and 74 J for FTP, FLC, VLC, and the proposed algorithm. This result shows the usefulness of the proposed algorithm. When there is a mixture of files to be synchronized, the proposed algorithm adaptively chooses which deduplication scheme to be used, thereby minimizing energy consumption.

In summary, the experiments reveal that, depending on the redundancy pattern of files, the best deduplication strategy varies. VLC requires greater energy cost for detecting redundancy, and FLC may fail to find redundant data when there is byte shift problem. This calls for adaptive selection of deduplication strategy, which is the goal of the proposed algorithm. The proposed algorithm requires energy in computing similarity hashes and communicating with the server, but this cost is small compared to the cost paid for choosing the wrong deduplication technique. The result of synchronizing a mixture of files shows the superiority of the proposed adaptive algorithm, compared to using a fixed deduplication strategy.
4. Related Works

There are file synchronization services that use data deduplication, such as rsync [5], llrfs [6], and tpsync [7]. Rsync uses rolling checksum algorithm to detect redundant data and only transfers new data, while llrfs uses CDC (content-defined chunking) and set reconciliation technique to reduce amount of deduplication and network traffic. Tpsync uses CDC to find redundancy in a coarse-grained manner and then uses rolling checksum to remove redundancy in small scale. However, these services are not adequate for synchronizing large files such as mp3 and video files.

Data deduplication techniques for synchronizing large files include fixed-length chunking and variable-length...
chunking. FLC is used in deduplication systems such as Venti [2] and shows the fastest deduplication speed among deduplication techniques. FLC divides a file into fixed-length blocks, calculates hashes of the blocks such as MD5 or SHA1, and compares hash values of the two files to find redundant blocks. However, FLC may fail to find redundant blocks due to the byte shift problem, when a new data block is inserted in the modified file. For VLC, LBFS [3] is a representative work. LBFS uses Rabin fingerprint to find anchor patterns and divides a file into blocks using the anchors. The hash values of the anchor blocks are computed using MD5 or SHA1 and compared against the previous version of the file to remove redundancy. Since blocks are defined by anchors, VLC does not suffer from byte shift problem. Defining blocks (chunks) using anchor patterns is called contents-defined chunking (CDC). The problem with CDC is that the size of the blocks may vary significantly, which results in increased transfer overhead. HP also proposed the TTTD (two-threshold, two-Divisor) algorithm [8] which reduces the variation of block size in CDC.

Efforts have been made to reduce cost of comparing hashes. In Lillibridge [9], chunks are aggregated into segments, and the chunk with the largest hash value becomes the “champion.” Hashes are compared in segment units using sparse indexing map. This reduces memory usage of indexing and disk lookup overhead. In SiLo [10], similar to Lillibridge, the chunks are grouped after running CDC. The similarity is compared between chunk groups, and deduplication is applied only on the groups with high similarity. HYDRAstor [11] implements global data deduplication. The system uses a file system called HydraFS [12], which uses bloom filters. DEDE [13] is a data deduplication system designed to support SAN cluster file system in virtualization environments. Each host maintains logs which contain hash values of blocks and periodically generate queries to share block hashes and remove unnecessary hashes. Similarity based deduplication system [14] uses file similarity measure and uses delta encoding to reduce redundancy between segments with high similarity. Since the system only compares similarity, only 4 GB of memory is used when storing 1 PB of data. But reliability is low since the system only compares 4 values for each 16 MB segment. Also, its running time is high because similarity hash must be computed for all segments.

5. Conclusion

This paper proposes an energy efficient scheme for synchronizing large files using data deduplication. The proposed algorithm selects which deduplication technique to use based on file pattern, in order to minimize the total energy consumption of synchronization process. Using offline measurements, per-byte energy consumption of each deduplication techniques is measured. When the client tries to synchronize files, it computes and sends the similarity hashes along with the size of the files, and the server selects which deduplication strategy to use based on estimated total energy consumption. We have implemented and tested the proposed algorithm using a mobile client and a server. The results show that the proposed algorithm successfully chooses the best deduplication technique regardless of redundancy patterns and thereby reduces energy consumption compared to the case where a fixed strategy is used. In the near future, we will extend this idea to mobile file systems and mobile storage cloud systems to reduce energy consumption of file synchronization in these environments.

Conflict of Interests

The authors declare that there is no conflict of interests regarding the publication of this paper.

Acknowledgments

This research was also supported by Hallym University Research Fund, 2014, (HRF-201407-016) and this research was
also supported by Basic Science Research Program through the NRF funded by the MEST (.2012R1A1A2044694).

References

[1] K. Srinivasan, T. Bisson, G. Goodson, and K. Voruganti, “iDedup: latency-aware, inline data deduplication for primary storage,” in Proceedings of the 10th USENIX conference on File and Storage Technologies (FAST’12), p. 24, USENIX Association, Berkeley, Calif, USA, 2012.

[2] S. Quinlan and S. Dorward, “Venti: a new approach to archival storage,” in Proceedings of the 1st USENIX Conference on File and Storage Technologies, p. 7, Berkeley, Calif, USA, 2002.

[3] A. Muthitacharoen, B. Chen, and D. Mazieres, “A low-bandwidth network file system,” ACM SIGOPS Operating Systems Review, vol. 35, no. 5, pp. 174–187, 2001.

[4] Y. Moon, H. Jung, C. Yoo, and Y. Ko, “Data deduplication using dynamic chunking algorithm,” in Computational Collective Intelligence. Technologies and Applications: 4th International Conference, ICCCI 2012, Ho Chi Minh City, Vietnam, November 28-30, 2012, Proceedings, Part II, N.-T. Nguyen, K. Hoang, and P. Jędrzejowicz, Eds., vol. 7654 of Lecture Notes in Computer Science, pp. 59–68, Springer, Berlin, Germany, 2012.

[5] A. Tridgell, Efficient algorithms for sorting and synchronization [Ph.D. thesis], The Australian National University, Canberra, Australia, 1999.

[6] H. Yan, U. Irmak, and T. Suel, “Algorithms for low-latency remote file synchronization,” in Proceedings of the IEEE 27th Conference on Computer Communications (INFOCOM ’08), IEEE, Phoenix, Ariz, USA, April 2008.

[7] D. Xu, Y. Sheng, D. Ju, J. Wu, and D. Wang, "High effective two-round remote file fast synchronization algorithm," Journal of Frontiers of Computer Science and Technology, vol. 5, no. 1, pp. 38–53, 2011.

[8] T. Moh and B. Chang, "A running time improvement for the two thresholds two divisors algorithm," in Proceedings of the 48th Annual Southeast Regional Conference (ACM SE ’10), pp. 69.1–69.6, ACM, Miss, USA, 2010.

[9] M. Lillibridge, K. Eshghi, D. Bhagwat, V. Deolalikar, G. Trezise, and P. Camble, "Sparse indexing: large scale, inline deduplication using sampling and locality," in Proceedings of the 7th Conference on File and Storage Technologies (FAST’09), pp. 111–123, USENIX Association, Berkeley, Calif, USA, 2009.

[10] W. Xia, H. Jiang, D. Feng, and Y. Hua, "Silio: a similarity-locality based near-exact deduplication scheme with low ram overhead and high throughput," in Proceedings of the USENIX Annual Technical Conference (USENIXATC ’11), pp. 26–28, USENIX Association, Berkeley, Calif, USA, 2011.

[11] C. Dubnicki, L. Gryz, L. Heldt et al., "Hydrastor: a scalable secondary storage," in Proceedings of the 7th Conference on File and Storage Technologies (FAST’09), pp. 197–210, USENIX Association, Berkeley, Calif, USA, 2009.

[12] C. Ungureanu, B. Atkin, A. Aranya et al., "HydraFS: a high-throughput file system for the hydrastor content-addressable storage system," in Proceedings of the 8th USENIX Conference on File and Storage Technologies (FAST’10), p. 17, USENIX Association, Berkeley, Calif, USA, 2010.

[13] A. T. Clements, I. Ahmad, M. Vilayannur, and J. Li, "Decentralized deduplication in san cluster file systems," in Proceedings of the Conference on USENIX Annual Technical Conference (USENIX ’09), pp. 8, USENIX Association, Berkeley, Calif, USA, 2009.

[14] L. Aronovich, R. Asher, E. Bachmat, H. Bitner, M. Hirsch, and S. T. Klein, “The design of a similarity based deduplication system,” in Proceedings of the SYSTOR: The Israeli Experimental Systems Conference (SYSTOR ’09), pp. 6.1–6.14, ACM, Haifa, Israel, May 2009.
Submit your manuscripts at http://www.hindawi.com