International Journal of Security and Its Applications
Vol.7, No.5 (2013), pp.415-424
http://dx.doi.org/10.14257/ijsia.2013.7.5.38

Byte-index Chunking Algorithm for Data Deduplication System

Ider Lkhagvasuren[1], Jung Min So[1], Jeong Gun Lee[1],
Chuck Yoo[2] and Young Woong Ko[1]

[1]*Dept. of Computer Engineering, Hallym University Chuncheon, Korea*
[2]*Dept. of Computer Science and Engineering, Korea University, Seoul, Korea*
[1]*{Ider555, jso, jeonggun.lee, yuko}@hallym.ac.kr, hxy@korea.ac.kr*

Abstract

This paper presents an algorithm and structure for a deduplication method which can be efficiently used for eliminating identical data between files existing different machines with high rate and performing it within rapid time. The algorithm predicts identical parts between source and destination files very fast, and then assures the identical parts and transfers only those parts of blocks that proved to be unique region. The fundamental aspect of reaching faster and high scalability determining duplicate result is that data are expressed as fixed-size block chunks which are distributed to "Index-table" by chunk's both side boundary values. "Index-table" is a fixed sized table structure; chunk's boundary byte values are used as their cell row and column numbers. Experiment result shows that the proposed solution enhances data deduplication performance and reduces data storage capacity extensively.

Keywords: *Deduplication, Chunk, Index-table, Anchor Byte, Byte-index Table*

1. Introduction

With the explosion of data such as text, audio, video, image, and the proliferation of the data centers, the regulatory back-up data is the pending issue to be resolved. More importantly, universal digital data has been increasing faster than the Moore's Law of doubling the transistors on the integrated circuit in every two years. Deduplication is one of the main solutions to prevent accretion of duplicate data and bring cost saving in data centers. Deduplication technologies can further reduce the required storage capacity and can be used into various file systems. Data deduplication is technique for effectively reducing the storage requirement of backup feasible. Furthermore Data deduplication is a way to reduce storage space by eliminating data to ensure that only single instance of data is stored in storage medium. Data deduplication technique has also drawn attraction as a means of dealing with large data and is regarded as an enabling technology. Furthermore chunking based deduplication is one of the most effective, identical regions of data with references to data already stored on disk. Typically content-defined chunking [1, 2] and Fixed-sized chunking [3] are main deduplication schemes in among the chunking based data deduplication approaches.

The primary goal of this paper is to show a novel data deduplication system that provides simple fast and high rate data duplication ratio result. In other word, the proposed system will get higher deduplication ratio than well-known static chunking and also performance speed is much faster than speed of content-defined chunking approach. The key idea is to find the data blocks which are very high percentage probability duplication rate from the file in very fast time. In our approach, we adapt a table structure called "Index table" which is used for

[1] Corresponding author : yuko@hallym.ac.kr

ISSN: 1738-9976 IJSIA

detecting identical data blocks during data deduplication process. This two dimensional matrix structured table is basically reference to the file chunks in a server. The rest of this paper is organized as follows. In Section 2, we describe related works about deduplication system. In Section 3, we explain the design principle of proposed Byte-index Chunking system and implementation details. In Section 4, we show performance evaluation result of the proposed system and we conclude and discuss future research plan.

2. Related Works

There are several different data deduplication algorithms including static chunking, content-defined chunking, whole-file chunking and delta encoding. Static chunking is the fastest algorithm among the others for detecting duplicated blocks but the performance is not acceptable. On the other hand, byte shifting can detect all of the duplicated blocks with high overhead. The main limitation of static chunking is “boundary shift problem”. For example when adding a new data to a file, all subsequent blocks in the file will be rewritten and are likely to be considered as different from those in the original file. Therefore, it's difficult to find duplicated blocks in the file, which makes deduplication performance degrade. One of the well-known static chunking schemes is Venti [3]. Venti is a network storage system using static chunking, where 160-bit SHA1 hash key is used as the address of the data. This enforces a write-once policy since no other data block can be found with the same address. The addresses of multiple writes of the same data are identical, so duplicate data is easily identified and the data block is stored only once.

Content-defined chunking data deduplication provides a user with opportunity of best density of storage. However, content-defined data deduplication approach can achieve high deduplication ratio, but it spends too much time to perform deduplication process in comparison with the other data deduplication approaches. In content-defined chunking, each block size is partitioned by anchoring based on their data patterns. This scheme can prevent the data shifting problem of the static chunking approach. One of the well-known content-defined chunking systems is LBFS [2] that is a network file system designed for low bandwidth networks. LBFS exploits similarities between files or versions of the same file to save bandwidth. It avoids sending data over the network when the same data can already be found in the server’s file system or the client’s cache. Using this technique, LBFS achieves up to two orders of magnitude reduction in bandwidth utilization on common workloads, compared to traditional network file systems. Delta encoding[4] stores data in the form of differences between sequential data. Lots of backup system adopts this scheme in order to give their users previous versions of the same file from previous backups. This reduces associated costs in the amount of data that has to be stored as differing versions.

DEDE [5, 6] is a decentralized deduplication system designed for SAN clustered file systems that supports a virtualization environment via a shared storage substrate. Each host maintains a write-log that contains the hashes of the blocks it has written. Periodically, each host queries and updates a shared index for the hashes in its own write-log to identify and reclaim storage for duplicate blocks. Unlike inline deduplication systems, the deduplication process is done out-of-band so as to minimize its impact on file system performance. In [7], they propose a data deduplication system using file modification pattern. This approach can detect how file is modified and what types of deduplication is best for data deduplication.

3. Design and Implementation of Byte-index chunking

We implemented source-based deduplication network file system using Byte-index based chunking approach. In source-based approach, data deduplication process is performed in the

client side and the client sends only non-duplicated files or blocks to deduplication server. The client performs file data deduplication process by sending file hash key to server. The server checks file hash key from file hash index on DBMS. If there is no matching file hash key in the server, the client starts block-level deduplication. The client divides a file into several blocks and calculates hashes of the each block. The list of hash keys is delivered to the server and the server checks duplicated blocks by comparing the hash key with hash keys in the server. The server makes a non-duplicated block list and sends it to the client. Finally, the client sends the non-duplicated data blocks to the server.

The proposed system transfers "Index-table" (size: 256*256) between server and client to determine blocks that have high probability duplication rate. "Index-table" is a 256x256 sized table structure; all the chunk's information (chunk number) of file are distributed this table as both side boundary byte values of chunk are used as express row and column number in the Index-table. The proposed system accesses file just same as NFS. Let's suppose that once file needs to be synchronized in the client side. Then client sends file identification information to the server. When server receives the information from the client, the server filters the file chunks from DBMS using file identification. If there is nothing comes out from the server DBMS using file identification, it means file is considered to be not duplicated. Therefore the client transfers all the data blocks of the file to the server and the server executes the chunking process to store file meta-data information to the DBMS.

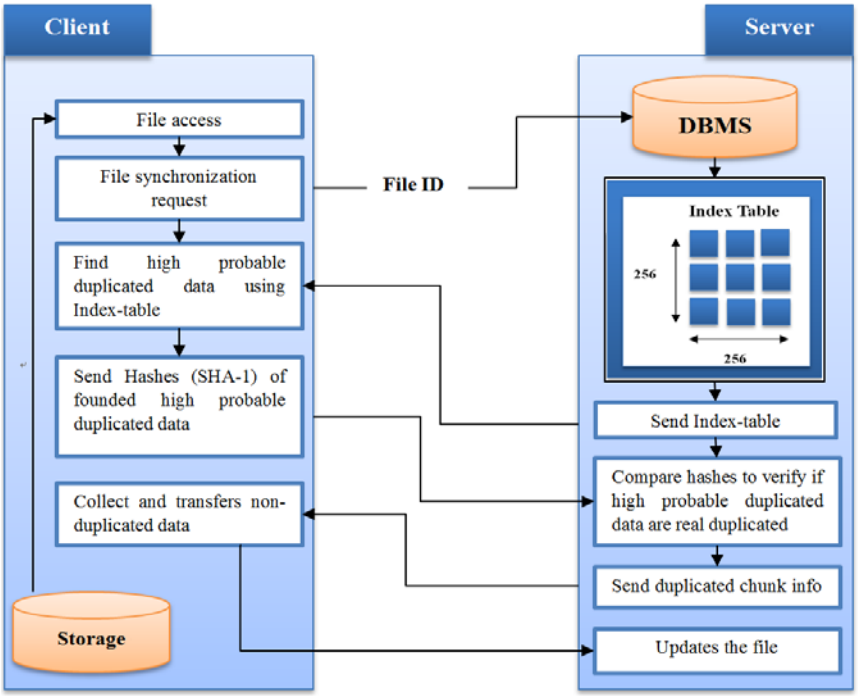


Figure 1. The Proposed System Architecture Overview: Byte Index based Approach

If file information has already saved to the DBMS, server accumulates them and produces Index-table of file. After that server sends produced Index-table to the client. Using this Index-table, client retrieves the duplicated chunks with high probability in very fast time. The client calculates the SHA-1 hash values from this result, and then the client also sends back them to the server. The server confirms whether the chunk is actually duplicated or not by

comparing these hash values to the corresponding hash values from server DBMS. Finally the proposed system knows the non-duplicated region and duplicated region of a file and then accumulates all this information and transfers only non-duplicated region information to the client.

3.1. Byte-index Chunking Algorithm Concept

Firstly, we aim to find the chunks that are expected to be duplicated (highly probable duplicate chunk) by their byte values. If the chunk in modified file not only has the same length with the any adjacent chunks in the server, but also both these chunks store same bytes of values at the position where boundaries of each chunk in server's adjacent chunks, then we call this chunk in modified file as "Highly probable duplicate chunk" .

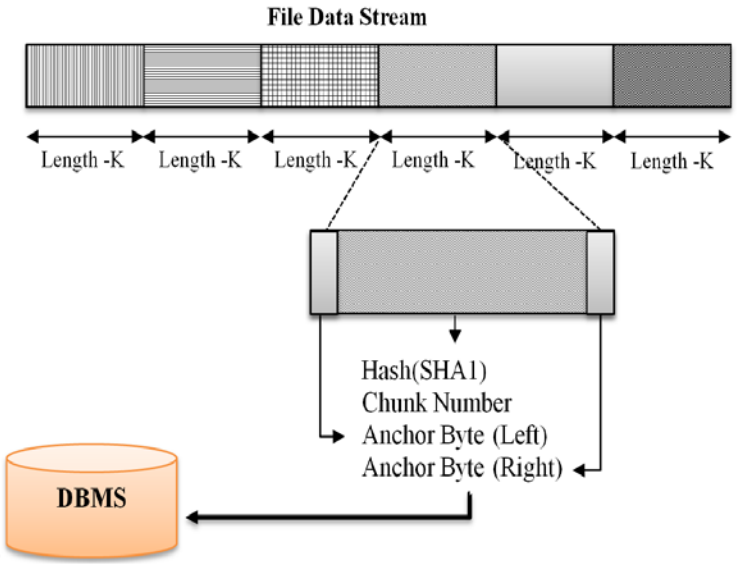


Figure 2. Overview of Chunking with Anchor Points

Figure 2 shows what the "highly probable duplicate chunk" is. Indeed, we can consider the "highly probable duplicate chunk" is a duplicate. Probability to be a duplicate is only one from 2564 (4,294,967,296) occasions and those two chunks have the same bytes in their specific four positions. Moreover, we can get each data block hash by applying hash functions such as SHA1, MD5 and SHA256. Nevertheless, "Duplicate high probability chunk" is possibly seen to be duplicated but we confirm whether they are duplicated or not by their SHA1 value after finding the "Duplicate high probability chunk". These processes are the backbone of our proposed system.

From the Figure 2, we can see in the proposed system with "Index-matrix". We use it for to find "Highly probable duplicate chunk" in a short time. "Index-matrix" is a special table, which was mentioned pervious. The table size is 256x256. Each cell contains index list of the chunks in the server. If some cells of "Index-matrix" table contain values, then these are the indexes of chunks in which two edge boundary bytes are equal to its cell horizontal and vertical position index. Thus, we can find any part of the data that might be duplicated through accessing "Index-matrix" to location at its boundary bytes value in a fast manner. Hence, we use this "Index-matrix" table to lookup "highly probable duplicate chunks".

For improving our search results to be more accurate, not only do we search a single chunk, but we also aim to seek adjacent double chunks for per offset in the modified file. Once we find the chunk that might be duplicated, we continue to confirm them exactly duplicated by comparing their SHA1 hash values. However, there may be a rare of occasion when a small of these chunks will not be proven to be duplicated. If this happens, then we augment the lookup process not only into the range of the proven chunk but also other highly probable duplicate chunks. We repeat this step until there is no highly probable duplicate chunk proven to a duplicate.

3.2. Predicting Duplicated Data with Look-up Process

In the client side when system receives "Index-table" from the server, the proposed system now aim to find part of chunks as hypothesis to be duplicated from the file in very fast time. Which means, system no need to calculate any hash to perform, no need to read bytes from duplicated chunks and system only use "Index-table" while system is finding high probable to duplicated chunks. The finding process starts the reading file from beginning to the end.

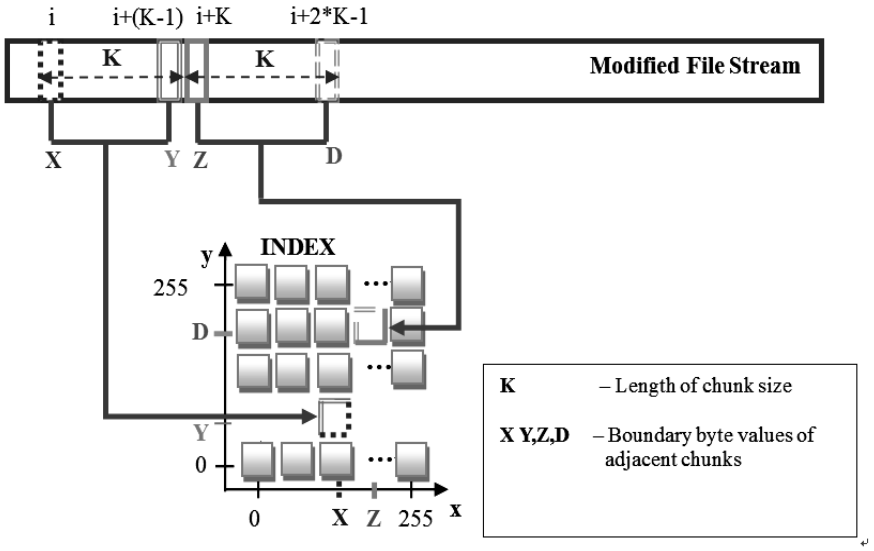


Figure 3. Lookup Chunks which are Might be Duplicated in Modified File

At the each location while system is reading bytes from file, system also concerns 3 more specifically byte location of the file. Let's suppose system is at the i-th location of file data stream reading, system also concerns the bytes at the (i+K-1)-th, (i+K)-th and (i+2*K-1)-th position when K is constant value of chunk size in the proposed system. First, system takes i-th and (i+K-1)-th position bytes, access cell of Index-table where cell location is i-th to the horizontal and (i+K-1)-th to the vertical and checks if cell contains any value. If cell doesn't include any value, system considers that byte at the i-th position couldn't be beginning of duplicated chunk and thus system keeps continue to finding process as shift one byte right. If cell contains any value, then system takes bytes at the (i+K)th and (i+2*K-1) –th offsets of the file, access the cell of Index-table as previous way and confirms if cell contains any value as previously. If that cell not includes any value, system skips the offset and shifts one byte right to continue finding process as same as previous step. If cell contains any value, then system also needs to be final check that value is exactly one over than founded value of previous cell. If that is not matched correctly system also consider the byte at the i-th position

couldn't be beginning of duplicated chunk and thus shifts one byte right to continue finding process. If comparing is matched which means the founded value is exactly one over than previous founded value system considers the parts data that from i-th to (i+2*K-1) -th offsets in the file to be "High probable to duplicated" chunk and shifting offset by 2*K to minimize preformation speed. Such wise as system scans the bytes from begin to end of the file and gets High-probable-duplicated chunks.

3.3. Verifying Predicting Chunks to be Duplicated

Although systems determine the chunks that might be duplicated (High-probable-duplicated), it is doubtful that the chunks are duplicated unless comparing their hash values to the original one's hash values. Therefore, system estimates the hashes of high-probable-chunks and sends them (with their chunk number) to the server for ascertaining the real duplication. The server receives the hash values and also filters the hash values of the corresponding chunk's hashes from DBMS using chunk numbers. Then system compares them as convenient to determine the duplicates. If some of the High-probable-duplicated chunks are cannot proven to be a duplicate (even though this kind of occasion has a very low likelihood), then system sends back non-match chunk information to the client. The client receives the non-duplicated chunk information from the server and performs the finding "High-probable-duplicated" process again and it is performed only within the chunk range which has proved to be duplicated. The finding process is carried out as the same as before. This cycle process continues until there no non-match chunks info comes from the server to the client.

4. Performance Evaluation

This section evaluates "Byte-Index Chunking System" with several experiments. First, we examine the behavior of proposed system's deduplication ratio result with comparing content-defined chunking, and fixed-size chunking approach. Next, we measure the performance time consumption utilization of system under several common workloads and compare it to the content-defined and fixed-size chunking approaches with deduplication result and with measured the performance time. In this work, we developed a deduplication storage system and evaluate the performance of the proposed algorithm.

The server and the client platform consist of 3 GHz Pentium 4 Processor, WD-1600JS hard disk, 100 Mbps network. The software is implemented on Linux kernel version 2.6.18 Fedora Core 9. To perform comprehensive analysis on similarity based deduplication algorithm, we implemented several deduplication algorithms for comparison purpose including fixed-length chunking, and content-defined chunking.

Table1 Amount of Modified Data Version Files of given Older Version

Number	Data Size	New Data	Overlap(%)
1	1110 MB	135 MB	87
2	1110 MB	232 MB	79
3	1110 MB	343 MB	69
4	1110 MB	479 MB	57
5	1110 MB	580 MB	48
6	1110 MB	669 MB	39
7	1110 MB	790 MB	28
8	1110 MB	889 MB	19
9	1110 MB	1000 MB	9

We made experimental data set using for modifying a file in a random manner. In this experiment, we modified a data file using *lseek()* function in Linux system using randomly generated file offset and applied a patch to make test data file.

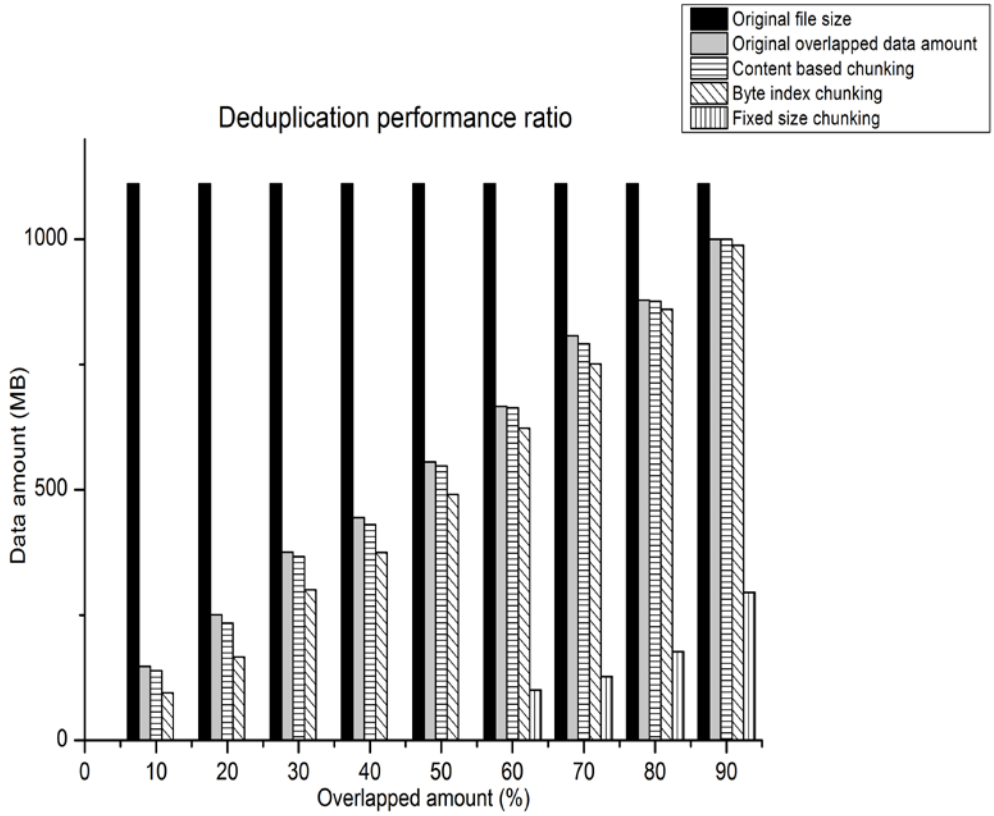


Figure 4. Deduplication Ratio Result of Chunking based Approaches. (By File Modification %)

In Figure 4, we performed data deduplication experiment on deduplication capability varying duplication rate. We examined each of to see how much duplication there is between files under a chunking approaches workload. From the deduplication graph in Figure 4, we can see each of content-defined chunking approach, fixed-sized chunking approach and byte-index chunking could find data redundancy of given percentage amount modification file between given original file. From the overlap of modified file in graph, content-defined chunking approach does the closest data deduplication ratio result to overlap of modified file than others. Almost, there is no difference between overlapping amount and content-defined chunking approach overlapping amount. Fixed-size chunking approach shows the lowest deduplication ratio result because of vulnerable to shifts inside data stream. For byte-index based chunking data deduplication approach shows a much better result than fixed chunking approach though it couldn't reach high as content-defined chunking approach.

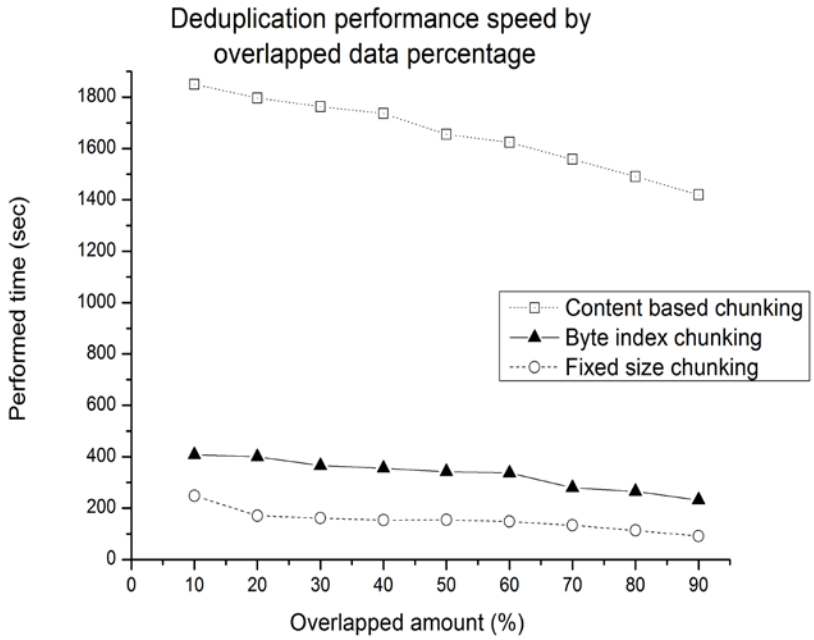


Figure 5. Deduplication Performing Time of Chunking based Approaches. (By FILE Modification Percentage)

We measured the deduplication speed varying given percentage modification amount, Figure 5. We can see content-defined chunking approach is performed in longest time. Byte-index based chunking approach is seen slightly slower than fixed chunking approach in the experiment. With this experiment result, we can conclude that byte-index chunking is very practical approach compared to several well-known data deduplication algorithms.

5. Conclusion

This paper presents an enhanced algorithm and structure for a deduplication method which can be efficiently used with high performance deduplication throughput and capacity in rapid time. The proposed deduplication system provides key point to predict highly probable to identical data within fast time. Lookup process shifting only single byte or twice chunk size depending if there is probable chunk existed at the offset position or not. We have found that using Byte-index chunking is superior to traditional data deduplication. Experiments under common operations, Byte-index approach can get separate amount as content-defined approach can do in data deduplication and performance time consume over an order of less than it, tendency to similar with the Fixed-sized chunking. Experiment result shows the proposed scheme achieves very high data deduplication capability, roughly same result with content-defined chunking approach. Fixed-length chunking shows worst performance result among data deduplication algorithms. Several issues remain open. First, our work has limitations on supporting simple data file which has redundant data blocks with spatial locality; therefore, if the file has several modifications then overall performance will be degrade. For future work, we plan to build a massive deduplication system with huge number of files. In this case, handling file similarity information needs more elaborated scheme.

Acknowledgements

This research was supported by Hallym University Research Fund, 2012(HRF-201209-024) and this work was supported by the National Research Foundation of Korea(NRF) grant funded by the Korea government(MEST) (No. 2011-0029848).

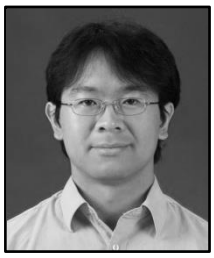
References

[1] K. Eshghi and H. Tang, "A framework for analyzing and improving content-based chunking algorithms", Hewlett-Packard Labs Technical Report TR. 30, **(2005)**.
[2] A. Muthitacharoen, B. Chen and D. Mazieres, "A low-bandwidth network file system", ACM SIGOPS Operating Systems Review, vol. 35, no. 5, **(2001)**, pp. 174-187.
[3] S. Quinlan and S. Dorward, "Venti: a new approach to archival storage", Proceedings of the FAST 2002 Conference on File and Storage Technologies, **(2002)**.
[4] M. Ajtai, R. Burns, R. Fagin, D. D. E. Long and L. Stockmeyer, "Compactly encoding unstructured inputs with differential compression", Journal of the Association for Computing Machinery, vol. 49, no. 3, **(2002)**, pp. 318-367.
[5] F. Douglis and A. Iyengar, "Application-specific delta-encoding via resemblance detection", Proceedings of the USENIX Annual Technical Conference, **(2003)**, pp. 1-23.
[6] P. Kulkarni, F. Douglis, J. LaVoie and J. Tracey, "Redundancy elimination within large collections of files", Proceedings of the annual conference on USENIX Annual Technical Conference, USENIX Association, **(2004)**.
[7] H. M. Jung, S. Y. Park, J. G. Lee and Y. W. Ko, "Efficient Data Deduplication System Considering File Modification Pattern", International Journal of Security and Its Applications, vol. 6, no. 2, **(2012)**.

Authors

Ider Lkhagvasuren graduated from Mongolian University of Science and Technology in 2007. He also graduated Department of Computer Engineering, Hallym University with master's degree in 2013.He is currently an working as researcher in Operating System Laboratory, Department of Computer Engineering, Hallym University. His research interests include Data deduplication and Cloud system.

Jungmin So received the B.S. degree in computer engineering from Seoul National University in 2001, and Ph.D. degree in Computer Science from University of Illinois at Urbana-Champaign in 2006. He is currently an assistant professor in Department of Computer Engineering, Hallym University. His research interests include wireless networking and mobile computing.



Jeong-Gun Lee received the B.S. degree in computer engineering from Hallym University in 1996, and M.S. and Ph.D. degree from Gwangju Institute of Science and Technology (GIST), Korea, in 1998 and 2005. He is currently an assistant professor in the Computer Engineering department at Hallym University.

Chuck Yoo received the B.S. degree in electronics engineering from Seoul National University, Seoul, Korea and the M.S. and Ph.D. in computer science in University of Michigan. From 1990 to 1995, he worked as researcher in Sun Microsystems Lab. He is now a Professor in College of Information and Communications, Korea University, Seoul, Korea. His research interests include Operating System, Virtualization and multimedia streaming.

Young Woong Ko received both a M.S. and Ph.D. in computer science from Korea University, Seoul, Korea, in 1999 and 2003, respectively. He is now a professor in Department of Computer engineering, Hallym University, Korea. His research interests include operating system, embedded system and multimedia system.