LETTER

Stride Static Chunking Algorithm for Deduplication System

Young-Woong KO[†a], ***Member***, **Ho-Min JUNG**[†], ***Nonmember***, **Wan-Yeon LEE**[††], ***Member***, **Min-Ja KIM**[†††], ***Nonmember***, ***and*** **Chuck YOO**[†††b], ***Member***

SUMMARY In this paper, we propose a stride static chunking deduplication algorithm using a hybrid approach that exploits the advantages of static chunking and byte-shift chunking algorithm. The key contribution of our approach is to reduce the computation time and enhance deduplication performance. We assume that duplicated data blocks are generally gathered into groups; thus, if we find one duplicated data block using byte-shift, then we can find subsequent data blocks with the static chunking approach. Experimental results show that stride static chunking algorithm gives significant benefits over static chunking, byte-shift chunking and variable-length chunking algorithm, particularly for reducing processing time and storage space.

key words: *static chunking, stride, deduplication, byte-shift*

1. Introduction

In the face of exponentially growing data volumes, redundant data elimination techniques have assumed critical significance in the design of modern storage systems. For instance, in the Linux ftp mirror storage server, duplication data blocks exceed 50%. Data deduplication is a way to reduce storage space by eliminating redundant data to ensure that only a single instance of the data is stored in a storage medium. If implemented properly, data deduplication can reduce the substantial demand for storage space, thus reducing the cost of the storage system.

Data deduplication schemes use a chunking mechanism to divide a file into blocks. Well-known chunking algorithms include fixed-size, variable-size, and byte-shift chunking (Rabin fingerprinting) [1]. Fixed-size chunking is also referred to as static chunking, because each chunk has a static length. Static chunking allows files to be divided into a number of fixed-sized blocks, before applying hash functions to create a hash key of the blocks. A well-known static chunking is Venti [2], which is a network storage system that uses a 160-bit hash key. Venti enforces a write-once policy so other data blocks cannot have the same address. The main limitation of static chunking is the *boundary shift problem*. For example, 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. Variable-length chunking [3] 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 chunking approach. Variable-length chunking is also referred to as content-defined chunking or content-based chunking, because the size of each chunk is determined by the content value. A well-known variable-length chunking algorithm is LBFS [4], which is a network file system designed for low bandwidth networks. LBFS exploits the similarities between files or versions of the same file in order to save the bandwidth. Finally, byte-shift chunking is a rolling hash scheme where the input is hashed in a window that moves through the input. The Rabin hash module reads through the input and lets the window slide over the input data, while it recalculates the fingerprint each time after advancing by a byte. Byte-shift chunking recalculated the hash value each time it advanced by a byte and compared it with the information on the server, so the overheads were very high.

In this paper, we present the stride static chunking; a compromise solution for searching duplicated data blocks in a file that exploits the static chunking and byte-shift chunking. Static chunking is a fast algorithm for detecting duplicated blocks but its performance is not acceptable. On the other hand, byte-shift chunking can detect all of the duplicated blocks, but with high overhead. To enhance deduplication performance, we introduce the stride concept into the chunking algorithm. We assume that duplicated data blocks are generally gathered into groups; thus, if we find one duplicated data block using byte-shift chunking, then we can find subsequent data chunks with the static chunking approach.

2. Design Principle of the Proposed System

2.1 System Architecture

In this work, we implemented a deduplication server using a source-based approach [5]. In the source-based approach, the data deduplication process is performed in the client side and the client sends only non-duplicated files or blocks to the deduplication server. With this approach, we can save

Manuscript received November 19, 2012.
Manuscript revised February 27, 2013.

[†]The authors are with the Department of Computer Engineering, Hallym University, Korea.
[††]The author is with the Department of Computer Science, Dongduk Womens University, Seoul, Korea.
[†††]The authors are with the Department of Computer Science and Engineering, Korea University, Seoul, Korea.
a) E-mail: yuko@hallym.ac.kr
b) E-mail: hxy@korea.ac.kr (Corresponding author)

DOI: 10.1587/transinf.E96.D.1544

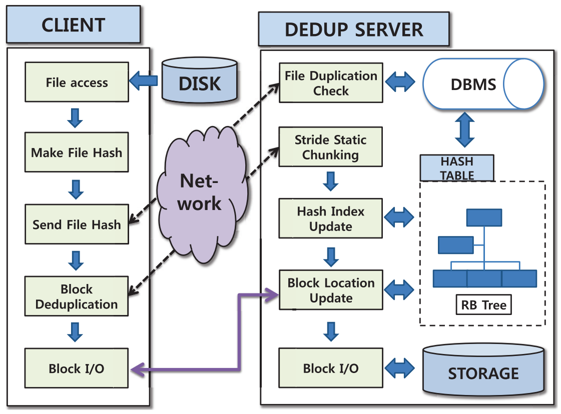


Fig. 1 System architecture overview.

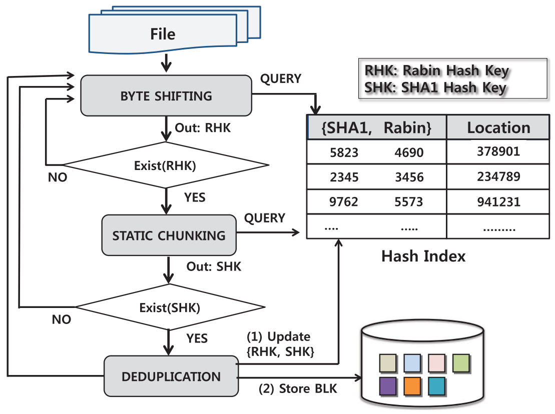


Fig. 2 The conceptual diagram of stride static chunking.

network bandwidth by reducing the number of duplicated data blocks.

As can be seen in Fig. 1, we first apply whole file chunking to eliminate file level duplication. In this work, the client calculates the SHA1 hash key and sends it to the deduplication server. If there is an identical hash key in the hash index, it means that we have a duplicated file on the server. The file hash generation module checks duplicated files by comparing the file hash and then processes block-level deduplication work. In the file deduplication stage, the hash key of each file is sent to the server. Duplicated files are checked at the server through comparison with existing file hashes on the DBMS. The server sends a hash list of duplicated files to the client. With this approach, we can prevent duplicated files from being transferred to the server. In the data deduplication module, block-level data deduplication is processed. The system divides the data stream into blocks with a chunking function. We then obtain each data block key using hash function. In our work, we adopt a static chunking method because of its simplicity and ease of implementation. The chunking size of a data block varies from 4 Kbytes to several megabytes. In our work, we fixed the chunking size from 4 Kbytes to 16 Kbytes in order to enhance the data deduplication performance. By choosing a small chunking block, we can increase the possibility of finding duplicated blocks.

2.2 Stride Static Chunking Algorithm

In this work, we integrate the byte-shift chunking approach with static chunking. We assume that duplicated data have spatial locality; therefore, if we can find one duplicated chunk using byte-shift chunking, then we can find subsequent duplicated chunks around that position using the static chunking approach. Figure 2 shows the conceptual diagram of the proposed system.

To implement the stride static chunking, we adapted two hash keys for each chunk: SHA1 hash key for static chunking and the Rabin hash key for byte-shift chunking. The Rabin hash key is used for light-weight duplication checking in byte-shift approach. However, the Rabin hash key has high probability of hash collision, so consequently we have to use another hash key (SHA1) to provide robustness against hash collision. The deduplication processing steps are as follows:

- The system calculates the Rabin hash key for a data block and compares it with hash indexes in the deduplication server. If an identical hash key does not exist, then the algorithm skips the fixed size of stride region and continues duplication checking using byte-shift chunking.
- When the Rabin hash key of the chunk is found in the hash index, and then the system compares the SHA1 hash key of that chunk with the SHA1 hash key in the hash index. If two SHA1 hash keys are identical, then we assume that the chunk is duplicated.
- In the deduplication step, one we found a duplicated chunk, then the system updates the SHA1 and the Rabin hash keys in the hash index and also updates the block location. If the chunk is not duplicated then the system stores the chunk to the storage server and inserts a new Rabin and SHA1 hash key tuple to the hash index.

Algorithm 1 presents the stride static chunking algorithm. The input data is the byte stream of a file. First, the system starts with byte-shift chunking with the *rabinfingerprint()* function, which calculates the hash key using the Rabin hash function. If *rabinlookup()* is TRUE, it means the same Rabin hash key exists in the deduplication server, then the system calculates the SHA1 hash key with the *SHA1digest()* function. If this hash key exists in the hash index, it means the chunk is really identical on the deduplication server. At this point, the chunking mode is changed to static chunking from byte-shift chunking. Static chunking works duplication check until there are no duplicated chunks. If static chunking reaches to non-duplicated region, then the chunking algorithm turns to byte-shift chunking.

3. Performance Evaluation

To perform comprehensive analysis on stride static chunk-

```
Input: DataStream
Output: HashList
begin
    strdieoffset ← 0
    offset ← 0
    length ← Length(DataStream)
    while offset < length do
        offset ← seek(DataStream, seek-cur)
        byte ← readbyte(DataStream)
        fingerprint ← rabinfingerprint(byte)
        if rabinlookup(fingerprint) = true then
            block ← read(DataStream, blocksize, offset)
            hash ← SHA1digest(block)
            if SHA1lookup(hash) = true then
                DedupHashList += (hash, fingerprint, offset)
                Perform Static Chunking
        else
            strideoffset += 1
            if strideoffset > blocksize × 2 then
                Perform Stride Static Chunking
```

Algorithm 1: Stride static chunking algorithm.

Table 1 Experiment data description.

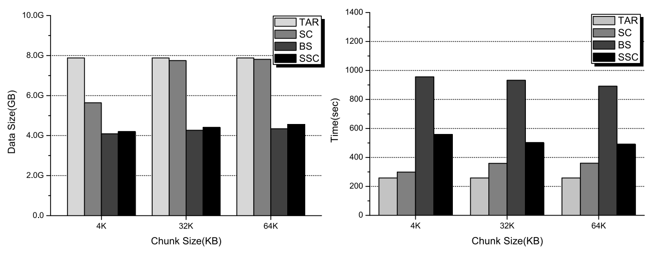
	Total (MB)	Description
Linux Distribution	7,952	Fedora Core
VMware Image Disk	18,154	CentOS
Media data	4,204	Drama Video

ing (SSC), we also implemented several algorithms for comparison purpose including static chunking (SC) and byte-shift chunking (BS). Table 1 summarizes the data sets used in this experiment. We have performed a series of sample experiments with different file types: Linux Distribution CD image, VMware image file with CentOS Linux system and multimedia video files. The chunk size is fixed as 4, 32 and 64 Kbyte.

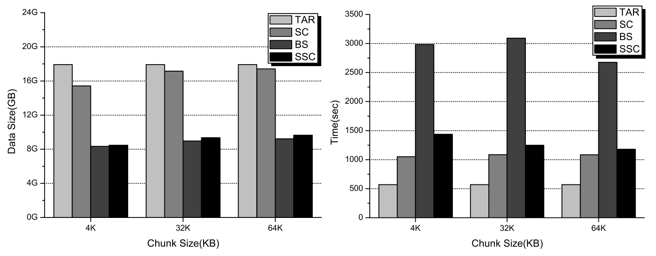
3.1 Performance Result: SSC vs. SC and BS

This experiment focused on evaluating the efficiency of the three algorithms: SC, BS, and SSC. The performance results for TAR are presented for comparison because TAR lacks a deduplication mechanism. In Fig. 3, we present the measurement results for the deduplication algorithms. The left figure shows the results of data compression, and the right figure shows the computation time. The x-axis and y-axis in the left figure denote the chunk size and data size (GB), respectively.

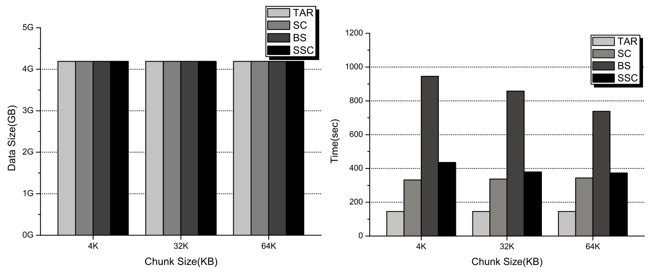
In SSC, the stride size is a key parameter for deduplication performance. If we increase the stride size then the deduplication system will have pros and cons: deduplication computation time will be decreased and data compression performance also will be degraded. Therefore, it is important to provide optimal stride size for SSC. In this work, we repeatedly test the performance of SSC and get heuristic value for the proposed system. In our system, we fixed stride size as 20 Kbyte for 4 Kbyte chunk, 80 KByte for 16 Kbyte chunk, and 320 Kbyte for 64 Kbyte chunk, respectively.



(a) Linux Kernel Source



(b) VMware Image Data



(c) Multimedia Data

Fig. 3 Deduplication performance result: SSC vs. SC and BS.

In this experiment, SSC was faster than BS in computation time. BS recalculated the hash value each time it advanced by a byte and compared it with the information on the server, so the overheads were very high. On the other hand, SSC only performed byte-shift chunking in the non-duplicated region and static chunking in the duplicated region. If SSC found a duplicated chunk in the data stream, it switched to the SC mode, so SSC could perform the deduplication process very rapidly. The data compression capability of SSC is superior to SC because the boundary shift problem of SC causes poor performance in data compression. Consequently, SSC can reduce data size more than SC and SSC can process data deduplication work faster than BS.

In terms of data compression, SSC and BS delivered the best results of all the algorithms. SSC compressed the Linux kernel data size from 8 to 4.2 GB (Fig 3 (a)) and the VMware data size from 18 to 8 GB (Fig 3 (b)). However, the compression performance results with SC were similar to FTP. As described above, SSC can find duplicated chunks rapidly in a small area using the BS approach. If there are no duplicated chunks, it increases the interval size in the data stream and initiates the BS approach to find duplicated chunks. If a duplicated chunk is found, the SC approach can find more duplicated chunks with low overheads. Therefore, SSC can find most of the duplicated chunks within a

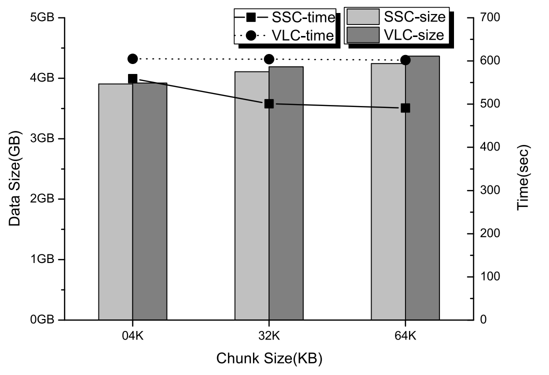


Fig. 4 Deduplication performance result: SSC vs. VLC.

data stream and its performance is comparable to BS. Although BS delivered the best compression performance, the computation time was twice that with SSC and three times that with SC.

For multimedia data (Fig. 3 (c)), there was no enhancement in terms of data compression because the multimedia data generally contained no duplicated data. Therefore, we consider that the deduplication algorithm is not suitable for multimedia data. We also tested the deduplication capacity with chunk sizes that varied from 4 Kb to 64 Kb. We found that the computation time decrease was inversely proportional to the chunk size and the difference was significant. For the Linux kernel data, the computation time with SSC was reduced from 550 s to 500 s, i.e., 10% less. For the VMware data, the difference was 250 s (from 1450 s to 1200 s). This experiment demonstrated that SSC delivered the best performance of the deduplication algorithms in terms of the data compression and deduplication computation times.

3.2 Performance Result: SSC vs. VLC

This experiment focuses on evaluation of the efficiency of two algorithms: SSC and VLC. In Fig. 4, we compare the performance of the two deduplication algorithm on Linux kernel source data. With respect to the computation time, SSC achieves fast deduplication performance compared to VLC by 5% (32 Kbyte chunk size) to 20% (64 Kbyte chunk size). In terms of data compression, SSC slightly outperforms VLC around 3%.

Although SSC is superior to VLC in terms of data compression performance and computation time, SSC requires more information for handling data deduplication. As shown in Fig. 2, SSC uses two hash keys: SHA1 hash key for static chunking and the Rabin hash key for byte-shift chunking. Therefore, SSC consumes slightly larger physical memory space than VLC approach. Consequently, we can conclude that in terms of data compression and deduplication computation time, SSC shows the best performance between deduplication algorithms.

4. Conclusion

In this paper, we introduced a novel deduplication algorithm that can be used as a way to store data efficiently in a storage system. The key idea of this paper is to exploit stride scheme that mixes static chunking and byte-shift chunking to accomplish enhanced deduplication capability. We assume that duplicated data blocks are generally gathered into groups, so if we can find one duplicated chunk, then we can find subsequent data chunks with static chunking approach. Stride static chunking only searches a fixed size of region for detecting duplicated chunk using byte-shift chunking. If there is no duplicated chunk, then it skips stride size of region and repeats byte-shift chunking. If a duplicated chunk is found, then it searches duplicated chunks with static chunking approach. The experimental results show that the proposed system can minimize storage space and reduces the computation time effectively.

Acknowledgment

This research was supported by Basic Science Research Program through the National Research Foundation of Korea (NRF) funded by the MEST (2012R1A1A2044694), (No. 2011-0029848), (2009-0076520).

References

[1] A.Z. Broder, "Some applications of rabin's fingerprinting method," Sequences II: Methods in Communications, Security, and Computer Science, pp.143–152, Springer-Verlag, 1993.

[2] S. Quinlan and S. Dorward, "Venti: A new approach to archival data storage," Proc. 1st USENIX Conference on File and Storage Technologies, FAST '02, Berkeley, CA, USA, 2002.

[3] P. Kulkarni, F. Douglis, J. LaVoie, and J. Tracey, "Redundancy elimination within large collections of files," Proc. Annual Conference on USENIX Annual Technical Conference, p.5, 2004.

[4] 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.

[5] H. Jung, W. Park, W. Lee, J. Lee, and Y. Ko, "Data deduplication system for supporting multi-mode," Intelligent Information and Database Systems, pp.78–87, 2011.