A Lookahead Read Cache: Improving Read Performance of Deduplication Storage for Backup Applications

Dongchul Park*, Young Jin Nam*† and David H.C. Du*
*University of Minnesota–Twin Cities, Minneapolis, MN 55455, USA
†Daegu University, Gyeongsan, Gyeongbuk, KOREA 712-714
Email: {park, youngjin, du}@cs.umn.edu

***Abstract*—Data deduplication (for short, dedupe) is a special data compression technique and has been widely adopted especially in backup storage systems with the primary aims of backup time saving as well as storage saving. Thus, most of the traditional dedupe research has focused more on the write performance improvement during the dedupe process while very little effort has been made at read performance. However, the read performance in dedupe backup storage is also a crucial issue when it comes to the storage recovery from a system crash. In this paper, we newly design a read cache in dedupe storage for a backup application to improve read performance by taking advantage of its special characteristic: the read sequence is the same as the write sequence. Thus, for better cache utilization, we can evict the data containers with smallest future references from the cache by looking ahead their future references in a moving window. Moreover, To achieve better read cache performance, our design maintains a small log buffer to judiciously maintain future access data chunks. Our experiments with real world workloads demonstrates that our proposed read cache scheme makes a big contribution to read performance improvement.**

***Keywords*-deduplication, dedupe, cache, read performance.**

I. Introduction and Motivations

Digital data explosion empowers data deduplication (for short, dedupe) to have been in the spotlight and over 80% of companies are drawing their attention to dedupe technologies [1]. Data dedupe is a specialized technique to eliminate duplicated data so that it retains only one unique data on storage and replaces redundant data with a pointer to the unique data afterwards. These days, dedupe technologies have been widely deployed particularly in secondary storage systems for data backup or archive due to considerable cost (i.e., time as well as space) saving. Thus, major concerns have been mostly related to write performance improvement thereby efficiently detecting and removing as many duplicates as possible with the help of efficient data chunking, index optimization/caching, compression, and data container design [2], [3], [4], [5], [6]. On the other hand, its read performance has not attracted considerable attention to researchers because read operations are rarely invoked in such dedupe storge systems. However, when it comes to system recovery from a crash, it has a significantly different story. Long term digital preservation (LTDP) communities were recently very emphatic on the importance of read performance in dedupe storage [7], [8]. Moreover, some primary storage systems have started to equip the dedupe technologies [9]. Although this read performance as well as write performance is also a crucial factor in dedupe storage, very little effort has been made at this issue. Typical data chunk (generally, a few KB) read processes in secondary dedupe storage are as follows: first, the dedupe storage system identifies the data container ID retaining the corresponding data chunks to be read. Then, it looks up the container (generally, 2 or 4MB) in the read cache. Once hitting the cache, it reads the chunks from the cache. Otherwise, it fetches one whole container from the underlying storage and then it can read the corresponding data chunks in the container. However, these read processes result in low cache utilization because even though there exist spatial locality in the data container, only partial data chunks in the containers are mostly accessed [1]. Furthermore, the higher dedupe rates, the higher data fragmentation rates. This can lower spatial locality so that it worsens cache utilization. Our key idea, in this paper, lies in exploiting future access patterns of data chunks. In general, read sequences are identical to write sequences in the dedupe storage for backup. Inspired by this special characteristic inherent in such an application, our read cache design can take advantage of future read access patterns during dedupe processes, but general cache algorithms such as LRU do not consider this special feature in dedupe mechanisms.

Based on these observations, we propose a lookahead read cache design in dedupe storage for a backup application. In this paper, we make the following main contributions:

- ***Exploiting Future Accesses:*** We maintain access information for future read references during dedupe (i.e., write) processes. Thus, our proposed design evicts a victim with a smallest future reference count from the read cache.
- ***Design Extension with a Log Buffer:*** We assign a portion of a read cache space into a log buffer which can effectively maintain future access chunks on the basis of our hot data identification scheme.
- ***Extensive Dataset Analysis:*** Our proposed design is fundamentally inspired by our diverse real dataset analyses.

Since our proposed design is a read cache scheme, unlike a selective duplication/deduplication approach that allows partial data duplication to improve read performance while hurting its write performance, our design not only does not hurt write performance at all, but also can be applied to other dedupe systems.

The remainder of this paper is organized as follows. Section II explains the design and operations of our proposed cache scheme. Section III provides a variety of our experimental results and analyses. Section IV discusses related work addressing especially a dedupe read performance issue. Finally, Section V discusses our future work.

II. Lookahead Read Cache Design

A. Rationale and Assumptions

In general, as more duplicates are eliminated from incoming data stream, the read performance stands in marked contrast to its good write performance due to the higher

Table I: Various dedupe gain ratio (DGR) in successive versions of each backup dataset (Unit:%). DGR represents the ratio of a data saving size to an original data size.

	ver-1	ver-2	ver-3	ver-4	ver-5	avg. DGR
ds-1	99.9	3.5	6.9	5.6	31.2	29
ds-2	100	28	24.7	14.9	20.6	37
ds-3	99.6	95.2	97.7	97.3	96.6	97
ds-4	90.5	55.4	63.6	20.8	20.6	50
ds-5	84.1	3.3	2.5	11.9	2.6	20
ds-6	54.4	22.4	–	–	–	38

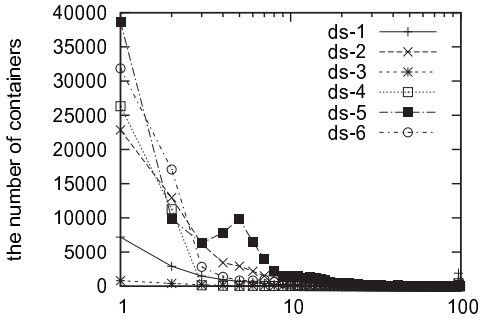


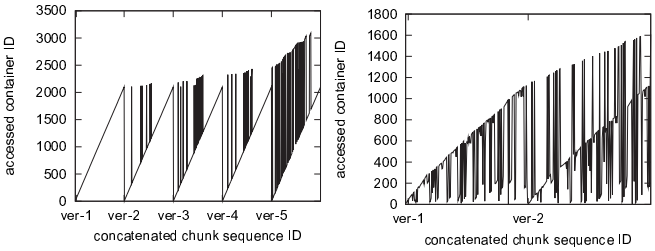
Figure 1: Distributions of the number of accessed container for six real backup datasets. X-axis represents the percentage of accessed chunks in a container.

likelihood of the shared data fragmentation [1]. This is the fundamental challenging issue of the tradeoff between read performance and write performance in dedupe storage. To address this read performance issue, we propose a novel read cache design leveraging future access information. In dedupe storage for backup or archive, the read access sequence is highly likely to be identical to its write sequence. Based on this key oservation, our proposed scheme records write access metadata information for the future read access during each dedupe process, which enables our lookahead cache to exploit future read references. We assume that each data chunk size is variable and a data container retaining many (generally, 200-300) data chunks is a basic unit for reads.

B. Dataset Analysis

We made an extensive analysis of six real backup datasets and also observed that a considerable portion of data is, in general, duplicated for each version of backup datasets (Table I). This implies the dedupe read cache can be poorly utilized in general cache design because only a small number of data chunks in a data container are accessed. Figure 1 exhibits the distributions of the number of accessed data container with respect to the percentage of accessed chunks in the shared container for each real backup dataset. That is, we observed how many data chunks in shared containers are accessed when duplicate data chunks are requested to read. When we reach a percentage of five (note: logarithmic scale for X-axis), it can accommodate most of the number of data container. This implies most of the data containers are only accessed of less than 5% of data chunks in each container. Therefore, we adopt 5% as our initial hot threshold value.

Figure 2(a) and (b) show container access patterns of the dedupe backup datasets. The X-axis represents a chunk sequence ID of successive versions of each backup dataset. The Y-axis represents accessed container IDs storing chunks



(a) *ds-1* (b) *ds-6*

Figure 2: Container access patterns of a typical dedupe.

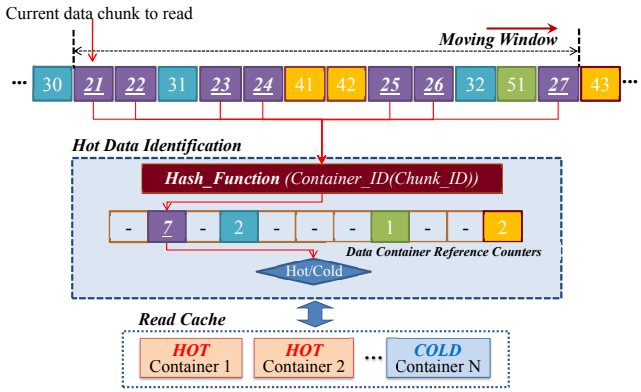


Figure 3: Overall architecture of our base read cache scheme. We assume each data chunk with same color belongs to the same data container.

indexed by the chunk sequence ID in the X-axis and the container ID starts from zero. For example, if a dataset includes many unique chunks, the accessed container ID increases linearly like the first version (ver-1) of the dataset *ds-1*: there are no fluctuations within the ver-1 section in Figure 2(a), whereas if a dataset contains many duplicates, the chart fluctuates due to fragmented accesses of previous containers. Four datasets (*ds-1* through *ds-4*) show similar data access patterns: most of the data chunks are unique in the initial backup version (ver-1) and the duplicate chunks increase for each successive backup dataset (Figure 2(a)). In fact, this is a typical characteristic of most backup datasets (due to space limit, we show only one figure (ds-1)). On the other hand, the other two datasets (*ds-5* and *ds-6*) exhibit different access patterns: there are many duplicates even in the initial version of backup datasets. Thus, we can see many vertical lines even in version 1 as well as successive versions of backup datasets (Figure 2(b)).

C. Architectural Overview

Figure 3 shows an overview of our proposed base read cache design for dedupe storage. The proposed scheme consists of three key components: a future sliding window, hot data identification scheme, and the read cache. The sliding window can be thought of as a lookahead window to see future chunk read accesses. The data chunk in its tail position is always considered a current data chunk to read and after reading the current chunk, the window takes a slide toward the future by one for each time. Based on this sliding window scheme, the hot or cold decision is made. Our hot

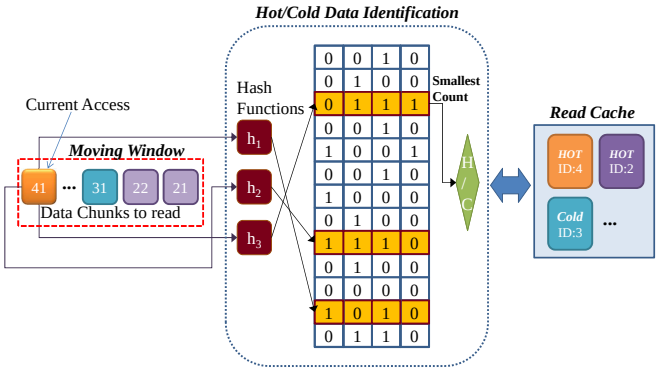


Figure 4: Architecture of our hot data identification scheme.

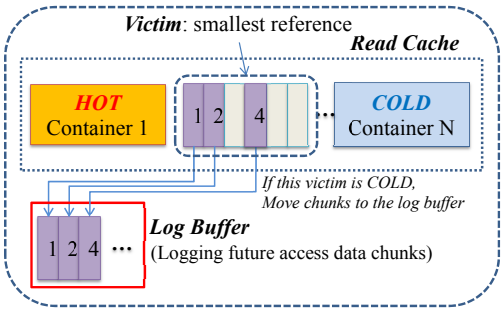


Figure 5: Architecture of our extended read cache design.

data identification scheme maintains data container reference counters. When any data chunk comes into the head position in the window, its reference counter of the corresponding data container is incremented by 1 (note: a data container is a basic unit of the chunk read). On the contrary, if any data chunk is evicted from the tail position in the window, the reference counter is decreased by 1. If the total container reference count of a current data chunk is over than a predefined threshold value (aforementioned 5%) within the window, the corresponding container is classified into a hot container, otherwise, a cold container. This hot threshold value was empirically derived in our workload analysis. More detailed architecture of this hot data identification scheme is described in the following subsection II-D. Lastly, our read cache stores the accessed containers and does not adopt the existing cache algorithms such as LRU or ARC. Instead, our read cache selects the least effective (i.e., smallest reference count) container as a victim. We will extend this read cache design by fitting a small log buffer into this base design to improve performance further.

D. Hot Data Identification Scheme

Hot data identification plays an important role in our cache design. Since it is invoked at each time a data chunk is accessed, it must achieve low computational overheads and small memory consumption [10]. To meet these requirements, we adopt counting bloom filter and multiple hash functions as displayed in Figure 4. That is, the aforementioned container reference counters are implemented by the counting bloom filter. This hot data identification scheme works as follows: whenever a data chunk comes in the sliding window to the head position, the chunk ID is converted into its container ID and then the container ID is fed to hash functions (we adopt multiple hash functions to reduce a false identification rate). Each hash value corresponds to their bit positions in the bloom filter. Finally, each reference counter is increased by 1 respectively. Similarly, outgoing data chunk from the window decreases each counter by 1 accordingly. Based on these basic operations, for identification of a current access chunk, its container ID is fed to hash functions and our scheme, then, checks its corresponding reference counters in each bit position of the bloom filter. Due to the likelihood of a hash collision, the scheme always chooses a smallest reference count. If it is greater than a predefined threshold, the data container is classified as hot, otherwise, cold.

E. Base Design

Our proposed read cache is mainly designed for the secondary dedupe storage for the backup or archive and makes an attempt to exploit its aforementioned special characteristic. In addition, its basic cache policy is managed by our hot data identification scheme. We initially assign a small (8MB=2MB container×4) read cache since we may consider multiple streams/processes environments. Overall working process is described as follows: whenever any data chunk is accessed for read, its container ID is first identified and then the container is ready to be stored in the cache. If the read cache is full of data containers, we need to choose a victim. Our scheme selects a container with a smallest future reference count as a victim. However, all data containers in the cache may turn to a different classification over time because the window continues to move forward. Thus, we need to update their reference counts accordingly, which will cause ignorable overheads because there are only a few numbers (basically, 4) of containers in the cache. Even though this algorithm is simple, it considerably improves dedupe read performance and outperforms the widely adopted cache algorithm, LRU. Unlike our extended design, our base design utilizes our hot data identification scheme mainly for its efficient reference count management, not for hot data identification.

F. Design Extension

This extended read cache design is inspired by our observations on dataset analysis. As shown in Figure 5, we assign a part of a read cache space into a log buffer to exploit future access information. That is, the key idea of this extended design is to maintain a small buffer to log future access chunks before a container is evicted from the read cache in accordance with the hot data identification. This log buffer is managed by a circular queue. Before eviction, our extended design attempts to classify the victim container into hot or cold by using our hot data identification scheme. If the victim container is identified as cold (that is, a small number of chunks in the container will be accessed in the near future), its remaining future access chunks within the window is stored into the log buffer. We do not store already accessed data chunks. Since we employ 5% as our hot threshold value, the maximum amount of data chunks to be logged is at most 5% of the total data chunks (typically 10-20 chunks) in the victim container. However, in most cases, the amount of logging chunks will be less than the threshold because the window moves forward over time and only the remaining future access chunks in the window are stored to

the log buffer. On the other hand, if a victim is identified as hot, we just evict the victim container without any logging because we can still achieve high container utilization with this hot container in the near future access. The rationale of this policy is that if we store many remaining future access chunks (in the hot container) into the log buffer, they also lead to the unnecessary eviction of many logged chunks in the buffer.

III. Performance Evaluation

A. Experimental Setup

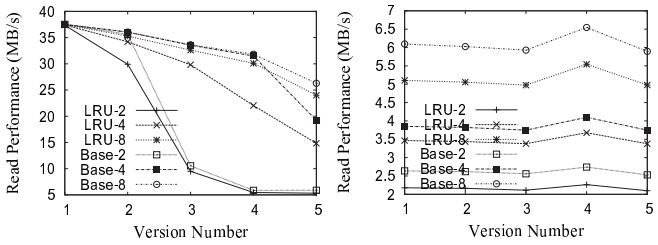
We implement our dedupe storage simulator on the basis of the DiskSim simulator [11]. The underlying storage includes 9 individual disks (IBM DNES-309170W), each of which provides storage capacity of 17,916,240 blocks (8.54GB). It is configured as RAID0 to provide better performance and enough storage space to accommodate all the chunks in the backup datasets. The stripe unit size is set to 32KB. We base our chunk indexing (hash index table) on google-sparsehash [12]. For read and write performance measurements, we ignore an elapsed time to execute the typical dedupe or our scheme because it is much smaller than storage I/O time. For the dedupe, we use a fixed sized container of 2MB for both and each container read (write) accesses 64 stripe units, where each individual disk serves about 7–8 stripe unit reads (writes).

We employ six backup datasets (Table I) traced in real data backup environments. Each dataset consists of 2 or 5 successive versions of (full) backup data streams. The *ds-1*, *ds-2*, and *ds-3* datasets were obtained from all Exchange Server data. The *ds-4* contains system data for a revision control system. The *ds-5* includes data from the /var directory in the same machine. The *ds-6* contains data from home directories with several users. For our experimental purpose, all the datasets except the last one (*ds-6*) were truncated to be 20GB in size and to have about 1,900K chunks in total. Each dataset contains chunked data streams by using variable-length chunking with an average chunk size of 8KB.

B. Experimental Results

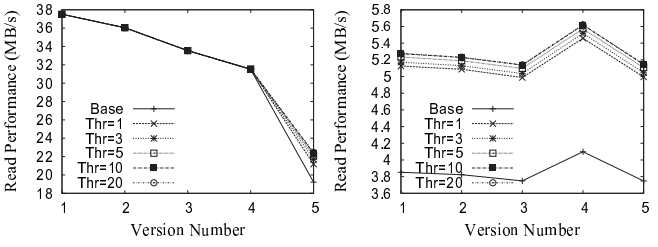
Figure 6 shows the read performance for LRU and our base design with various cache sizes. Note the the cache size of 2, 4, and 8 means the number of a data container (2MB). Thus, the cache size of two corresponds to 4MB cache. Moreover, due to the space limitation, we present only two datasets (*ds-1* and *ds-5*) because, based on our extensive dataset analysis, four datasets (*ds-1*, *ds-2*, *ds-3*, and *ds-4*) exhibit very similar data patterns and show almost identical result patterns. Similarly, the other two datasets (*ds-5* and *ds-6*) also exhibit similar patterns. Therefore, we just choose one of each on behalf of the other(s) afterwards. As plotted in the Figure 6, our base design shows better read performance than LRU for all cache sizes only with the change of a cache replacement algorithm because our algorithm exploits future access information. Our proposed base scheme improves the dedupe read performance by an average of 14.5% and 16.8% respectively.

Figure 7, 8, 9, and 10 depict the performance improvement with our extended cache design with various configurations. First of all, in Figure 7, we explore both the impact of a hot threshold value and performance improvement of our extended design. Hot data identification plays an important role in making decision about logging future access data



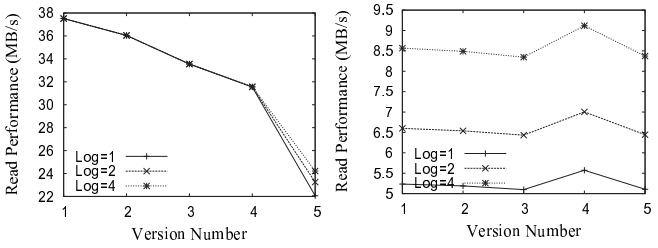
(a) *ds-1* (b) *ds-5*

Figure 6: LRU vs. Our Base Design



(a) *ds-1* (b) *ds-5*

Figure 7: Base vs. Extension with Diverse Hot Threshold

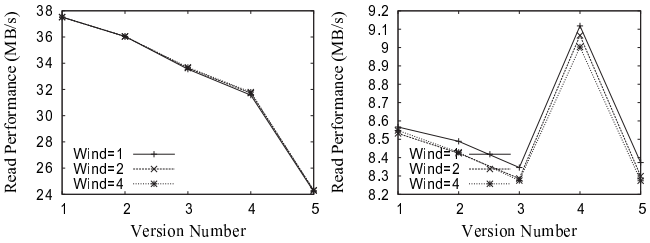


(a) *ds-1* (b) *ds-5*

Figure 8: Impact of a Log Buffer Size.

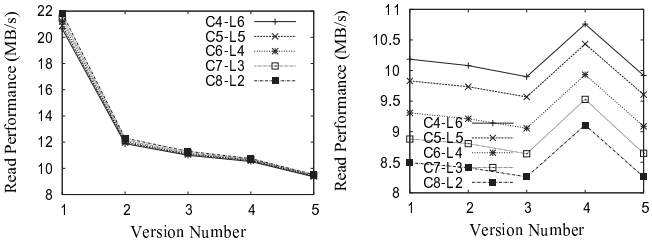
chunks in a container into the log buffer. The hot threshold of 5% means that if the percentage of total data chunk access in a data continer is over than 5%, the container is identified as hot, otherwise, cold. Only if it is identified as cold, the remaining future access chunks will be logged in the small buffer. Thus, intiutively, the higher a hot threshold, the better read performance because the more number of future access chunks will be able to be stored in the buffer. However, our design with the higer threshold than 3 or 5 does not lead to higher performance gain. In addition, we observe that our extended design (with a log buffer) does not considerably achieve performance improvement in *ds-1*, *ds-2*, *ds-3*, and *ds-4* (2.3% improvement on average). However, in both *ds-5* and *ds-6*, it signifiantly improves the read performance by an average of 63.1%. These results stem from the workload characteristics of the datasets and we have already addressed this in II-B. Moreover, Figure 8 supports the experimental results in Figure 7 and shows very similar performance patterns because all performance gains of our extended design are fundamentally originated from the log buffer. We also observed that our extended design outperforms LRU by an average of 64.3% in *ds-5* and *ds-6*.

Our sliding window size is another factor to be discussed since it implies how much we can look ahead in the near



(a) *ds-1* (b) *ds-5*

Figure 9: Impact of a Window Size.



(a) *ds-3* (b) *ds-5*

Figure 10: Impact of Diverse Cache and Log Buffer Sizes. In here, a total space (cache plus log buffer) is fixed. C and L stand for a cache size and a log buffer size respectively.

future. Interestingly, its impact is almost ignorable in *ds-1* through *ds-4* and a bigger window even worsens the performance in *ds-5* and *ds-6* (Figure 9). This is because the farther future accesses in a big window can contaminate a cache space, which can lead to low cache utilization in our cache policies. Lastly, we explore the impacts of various cache and log buffer sizes assuming the same total cache space. We vary both a cache size and a log buffer size. As shown in Figure 10, we can observe that a read cache size is a more important factor in *ds-1* through *ds-4*. Thus, assigning a more space into a read cache than into the log buffer will a better cache design in those types of datasets (we choose *ds-3* instead of *ds-1* in Figure 10 (a) since it shows clearer results). On the other hand, a larger log buffer has a more impact on the read performance in both *ds-5* and *ds-6*.

IV. Related Work

Zhu *et al.* in [5] put an empasis on importance of read performance in dedupe storage, in particular, for data recovery. They also addressed that read performance substantially decreased during dedupe process. Koller *et al.* [13] proposed a selective duplication scheme (named I/O Deduplication) to increase the read/write performance by reducing a disk head movement. They suggested a content-based cache design for both read and write requests as a part of their dedupe scheme, but it was implemented in a separate cache area from a virtual file system cache layer and just adopted the existing cache replacement algorithms such as LRU or ARC. Nam et al. in [14] introduced an indicator for the degraded read performance named chunk fragmentation level (CFL) and observed a strong correlation between the CFL value and the read performance under backup datasets. Recently, Srinivasan *et al.* [1] proposed primary inline dedupe system design (named iDedup). iDedup tried to exploit both spatial locality by selectively deduplicating primary data and temporal locality by maintaining dedupe metadata completely in memory, not on disk. However, it did not address the file system's read cache design.

V. Future Work

This work has assumed that the basic read unit in secondary dedupe storage is a data container retaining hundreds of small data chunks. As our future work, we will remove such assumption so that each data chunk can be directly read from the underlying dedupe storage. In addition, we are considering a variable container size and an adaptive scheme to workloads by dynamically changing the hot threshold. Furthermore, we are designing a novel dedupe system which can provide guaranteed read performance while assuring enough write performance in dedupe storage. Ultimately, we plan to combine both designs for our complete dedupe system to achieve both good read performance and write performance.

References

[1] K. Srinivasan, T. Bisson, G. Goodson, and K. Voruganti, "iDedup: Latency-aware, inline data deduplicationfor primary storage," in *Proceedings of the 10th USENIX Conference on File and Storage Technologies (FAST)*, February 2012.

[2] W. Dong, F. Douglis, K. Li, H. Patterson, S. Reddy, and P. Shilane, "Tradeoffs in scalable data routing for deduplication clusters," in *Proceedings of the 9th USENIX File and Storate Technologies (FAST)*, 2011.

[3] C. Dubnicki, L. Gryz, L. Heldt, M. Kaczmarczyk, W. Kilian, P. Strzelczak, J. Szczepkowski, C. Ungureanu, and M. Welnicki, "HYDRAstor: A scalable seconary storage," in *Proceedings of the 7th USENIX File and Storate Technologies (FAST)*, 2009.

[4] B. Debnath, S. Sengupta, and J. Li, "ChunkStash: Speeding up inline storage deduplication using flash memory," in *Proceedings of USENIX Annual Technical Conference (ATC)*, 2010.

[5] B. Zhu, K. Li, and H. Patterson, "Avoiding the disk bottlenext in the data domain deduplication file system," in *Proceedings of the 6th USENIX File and Storage Technologies (FAST)*, 2008.

[6] D. Meyer and W. Bolosky, "A study of practical deduplication," in *Proceedings of the 9th USENIX File and Storate Technologies (FAST)*, 2011.

[7] "LTDP Reference Model," *http://www.ltdprm.org*.

[8] "SNIA Digital Preservation and Capacity Optimization," *http://snia.org/forums/dpco*.

[9] D. Kay and M. Maybee, "Aspects of deduplication," *SNIA Spring 2010 Tutorials*, 2010.

[10] D. Park and D. Du, "Hot Data Identification for Flash-based Storage Systems using Multiple Bloom Filters," in *Proceedings of the 27th IEEE Symposium on Mass Storage Systems and Technologies (MSST)*, May 2011.

[11] "DiskSim v.4.0," *http://www.pdl.cmu.edu/DiskSim/*.

[12] "Sparsehash," *http://code.google.com/p/sparsehash/*.

[13] R. Koller and R. Raju, "I/O Deduplication: Utilizing content similarity to improve I/O performance," in *Proceedings of the 8th USENIX File and Storage Technologies (FAST)*, 2010.

[14] Y. Nam, G. Lu, N. Park, W. Xiao, and D. Du, "Chunk Fragmentation Level: An effective indicator for read performance degradation in deduplication storage," in *Proceedings of IEEE International Symposium on Advances on High Performance Computing and Networking (AHPCN)*, 2011.