A Forest-structured Bloom Filter with Flash Memory

Guanlin Lu [‡], Biplob Debnath [†,1], David H.C. Du [‡]

[‡] University of Minnesota, Minneapolis, USA.

[†] EMC Corporation, Santa Clara, USA.

E-mail: lv@cs.umn.edu, biplob.debnath@emc.com, du@cs.umn.edu

***Abstract*—A Bloom Filter (BF) is a data structure based on probability to compactly represent/record a set of elements (keys). It has wide applications on efficiently identifying a key that has been seen before with minimum amount of recording space used. BF is heavily used in chunking based data de-duplication. Traditionally, a BF is implemented as in-RAM data structure; hence its size is limited by the available RAM space on the machine. For certain applications like data de-duplication that require a big BF beyond the size of available RAM space, it becomes necessary to store a BF into a secondary storage device. Since BF operations are inherently random in nature, magnetic disk provides worse performance for the random read and write operations. It will not be a good fit for storing the large BF. Flash memory based Solid State Drive (SSD) has been considered as an emerging storage device that has superior performance and can potentially replace disks as the preferred secondary storage devices. However, several special characteristics of flash memory make designing a flash memory based BF very challenging. In this paper, our goal is to design an efficient flash memory based BF that is fully aware of these physical characteristics. To this end, we propose a Forest-structured BF design (FBF). FBF uses a combination of RAM and flash memory to design a BF. BF is stored on the flash, while RAM helps to mitigate the impact of slow write performance of flash memory. In addition, in-flash BF is organized in a forest-like structure in order to improve the lookup performance. Our experimental results show that FBF design achieves 2 times faster processing speed with 50% less number of flash write operations when compared with the existing flash memory based BF designs.**

I. Introduction

A Bloom Filter (BF) is a bit vector that compactly represents a set of items (keys) and supports key query/insert operations. It can definitely tell if a key is not present, but it may not tell with guarantee that a key is indeed present. In other word, an answer given by a BF bears certain false positive rate. To keep this false positive rate low, traditional BF designs have to set Bloom Filter size a priori to be a few times larger than the maximum number of items represented.

Bloom Filters are heavily used in chunking based data de-duplication. Chunking based de-duplication is an efficient technique to eliminate data redundancy within both backup data and data stored in primary storage. Traditionally, chunking based de-duplication requires a chunk index that consists of each chunk's identifier (i.e., a SHA1 hashed value computed based on chunk's content) and its resided location in disk. This index is used to determine whether a chunk already exists and to retrieve a data chunk from disk. However, in many practical cases (e.g., hundreds TB of data to be de-duplicated), the chunk index size is too big to fit in RAM and disk-based index is too slow. To reduce the frequency of disk accesses, Zhu et al. [15] adopts an in-RAM Bloom Filter to identify new chunks. If a chunk is identified by BF as new, it is temporarily stored in an in-RAM container without querying disk-based chunk index. When in-RAM container becomes full, it is written to disk at once and the disk-based chunk index is updated. It has been demonstrated that by deploying an in-RAM Bloom Filter, a considerable amount of disk accesses for chunk index lookup could be avoided. On the other hand, this in-RAM BF consumes a significant amount of RAM space (e.g. 1GB size BF per billion unique chunks [15]). Furthermore, when the dataset size could not be determined in advance, BF size must be able to scale up to accommodate the growth of the data set.

Querying a BF may randomly access any bit position in a BF. It is well known that random disk accesses perform worse than serial disk accesses. Flash-memory based Solid State Drive (SSD) appears to be a good candidate for storing a large BF since flash memory access time is faster than disk access time. Moreover, unlike hard disk, random read operations are as fast as sequential read operations for a SSD. Nevertheless, flash memory exhibits several special characteristics: (1) Data can be read/write by pages (a page size is typically 4KB). However, data erase operation is based on blocks (a block size is 128KB). (2) A page write is slower than a page read and data cannot be updated unless it is erased first (in-place update problem). Therefore, it is important to reduce the number of writes. (3) Each cell allows a limited number of erase operations in flash memory life cycle. To lengthen its life cycle, a wear-leveling algorithm is performed. Our goal of this paper is to propose an efficient dynamic BF design with flash memory that has considered these unique characteristics.

The general idea of building a Bloom Filter with flash memory is to utilize a limited amount of RAM combined with a much larger flash memory space to form a Bloom Filter so that its capacity could go beyond the RAM size limitation. Since the BF is stored in flash memory, key query/insert operations may trigger flash read/write accesses respectively. Therefore, it is important to design a BF structure that considers flash memory characteristics and use RAM effectively to reduce the flash memory access time. For example, a key query

[1]Work done when the author was graduate student at the University of Minnesota

may require couple of flash read operations while several key insertions could be buffered temporarily in RAM and committed to flash later through one write operation. In this paper, we count the number of flash reads to response to a key query. Similarly, we measure the time overhead of a key insertion by the total number of flash writes. Our design goal is to minimize the overhead for both key query and insert so that the processing speed (i.e., the number of records processed per second) could be maximized.

Canim et al. [5] proposed a BF design with flash memory which pre-allocates a single large space on flash as BF. This BF is further partitioned into k sub-BFs. A sub-BF is a Bloom Filter of a fixed size. Correspondingly, the given RAM space is partitioned into k small fixed-size buffers: one for each sub-BF. To query a key e, it first decides which sub-BF to be checked and then all bit positions in that sub-BF related to e need to be read, which may scatter a few pages in flash. Each key inserted to a given sub-BF will be temporarily buffered at its corresponding buffer. The buffer will be written to flash memory when it is full. This will trigger one or multiple block erases and a few page writes. We denote this by **single-layer** BF design in this paper. although this design is efficient for key queries and the number of block erases and page writes will be improved by buffering, the big size gap between the RAM-based buffer and its corresponding sub-BF on flash may still cause a large number of flash writes and block erases for key insertions. On the other hand, in order to optimize the performance of key insertions, a naive extension of *the dynamic expanding Bloom Filter* design presented by Guo et al. can be done [9]. In this scheme an initial BF as the size of RAM is allocated in RAM first. When this BF reaches its capacity (i.e., certain number of bits is set to 1), the whole BF is written to flash memory and a new BF is restarted in RAM. The same procedure repeats when this substitute reaches its capacity again. Eventually it forms a chain of BFs with only the latest one being in RAM. We denote this design by **linear-chaining** design, which is optimal in terms of number of write operations because each BF is written to flash only once and never gets modified after that. However, the key querying performance would deteriorate very soon since the number of flash reads per query grows *linearly* with the number of BFs chained. See Section II for more details of both designs.

In this paper, we propose a Forest-structured BF (FBF) design that is efficient for both key queries and insertions and can dynamically adapt to the growth of data set. Initially when the dataset size is small and could be fit in RAM, our design allocates the root-layer of the forest in RAM and behaves identical to a traditional in-RAM BF; As the dataset size grows and go beyond the root-layer's capacity, our forest naturally expands by allocating a new layer of BFs in flash, as children of the BFs in the root-layer. The entire root-layer is then written to flash and the spared RAM space is switched into a buffer space for key insertions. The forest allocates a new layer in flash whenever its current lowest layer reaches its capacity.

FBF design sets the sub-BF size to be the underlying flash-page size (say, 4KB) thus given any key, one flash page access is guaranteed to fetch all its bit positions. This significantly reduces the flash accesses for key query/insert operation.

The proposed forest-structure targets at minimizing the size gap between the in-RAM buffer and the associated BF on flash: FBF organizes a number of sub-BFs at each layer of a forest and always inserts keys to the lowest layer. At any time, only sub-BFs at one layer of the forest will be buffered in RAM, giving each to-be-inserted sub-BF more buffering space.

In addition, the proposed FBF design, compared with linear-chaining one, achieves a much few flash reads for a given key query. Given the same overall BF size m, the required flash reads for a key query in a b-branching (each node has b children except those at the lowest layers) FBF is $\mathcal{O}\left(log_b m\right)$, whereas the required flash reads for a linear-chaining one is $\mathcal{O}\left(m\right)$.

We use several real-world de-duplication workloads to drive and evaluate our design. Our experimental results show that FBF design outperforms the single-layer BF on a broad range of in-RAM buffer sizes. Particularly, our design achieves 2 times faster processing speed with 50% less number of flash-write operations. We also evaluate our design with data backup workloads but will not present the results in this paper due to page limitation. The main contributions of this paper are summarized as follows:

- A novel forest-structured BF design, which combines the advantages of both linear-chaining and single-layer designs and yields a significantly higher processing speed, is proposed. Meanwhile, our BF design can handle dynamic workloads whose size could not be determined in advance.
- A proposed novel buffer management scheme that is particularly optimized for flash write overhead hence block erases too. It could reduce the number of flash write operations by $2-3$ times compared with two current schemes [5], [7] on various types of workloads.
- We conducted extensive experimental evaluations on a typical de-duplication workload.

The rest of the paper is organized as follows. Section II gives a brief overview of two existing Bloom Filter designs with flash memory. Section III describes the proposed FBF design. Section IV presents experimental results on a typical data deduplication workload. Section V gives an overview of related work and Section VI presents some conclusions.

II. Overview of Two Bloom Filter Designs with Flash Memory

This section briefly describes two existing BF designs with flash memory and points out potential improvements.

Single-layer BF design divides a single large BF on flash into many individual BFs called sub-BFs (say k sub-BFs).The available RAM space is partitioned into k smaller fixed-size buffering blocks (buffer compartments). Each buffer compartment is corresponding to a sub-BF. This buffer compartment is used to delay writes triggered by bit position updates on its corresponding sub-BF. To query a key e, it takes a two-step hashing procedure: the first hashing function on e decides which sub-BF should be checked. At second step to check the

key, multiple hash functions are applied to identify a set of bit positions in this sub-BF. If all bit positions are set, the key is considered to have been seen before. Note this requires checking the corresponding buffer compartment in RAM first, if not found there then the sub-BF needs to be accessed from flash and checked. Otherwise, this key needs to be inserted into this sub-BF for the future query. However, the update of sub-BF will be delayed by buffering the offsets of these bit positions into buffer compartment. The sub-BF will be updated only if its corresponding buffer compartment is full.

Since a sub-BF consists of many flash pages (note a mentioned sub-BF size is 2 MB [5] while a flash page size is typically 4KB), the number of pages in flash to be read and updated depends on the number of pages containing at least one updated bit in the filled up buffer compartment.

The linear-chaining BF design, as presented in Section I, is particularly optimized for key insertions. Each required key insertion will be done with the current BF in RAM. Therefore, no flash memory access is involved. The BF in RAM will be written to flash memory when the number of bits being set reaches to a limit that a given false positive rate can no longer be guaranteed (i.e., a BF's capacity is reached). However, to response to a key query, a number of BFs in the order of written to flash memory may need to be checked. If the checking of a BF is a success, the response to the key query is positive. Otherwise, all chained BFs have been checked without any success. In this case, a key insertion is required. Each in-flash BF checking accesses a number of flash pages that contains at least one hashed bit position in the BF.

The main advantage of single-layer design is that it is very efficient for a key query. For a required key insertion, it is first delayed by buffering in an equally partitioned buffer compartment that corresponding to the targeted sub-BF. Without buffering, each key insertion may trigger a few page updates (i.e., all the pages containing at least one bit position needed to be set). With page update delay through buffering, the total number page updates will be reduced. However, due to the randomness of hashing results and limited buffering space, a buffer compartment can be filled up soon and the required bit position updates are scattering over many flash pages. Some of the pages may contain the bit position updates from one or two insertions. Furthermore, single-layer design requires a priori knowledge of the maximum number of keys inserted, which is sometime impossible for some applications.

On the other hand, linear-chaining design supports dynamic growing data set and is optimized for key insert operation. For a key query, especially an unsuccessful one, all chained BFs have to be checked and each checking may involve multiple page reads from flash. In addition, its false positive rate is accumulated from all chained BFs. Therefore, it is bigger than that of a single layer BF design. Thus, we preclude it for further discussion in this paper.

III. Forest-structured BF on Flash

In this section, we present our proposed FBF design along with the new buffer space managing scheme.

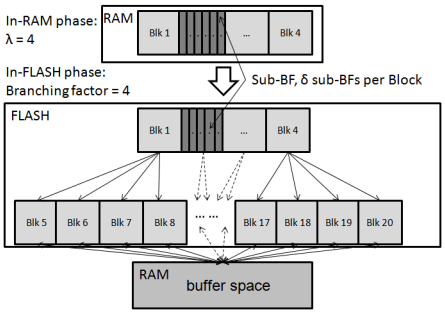

Figure 1. A Two-layer 4-branching FBF

```
FUNCTION in_flash_key_query(e)
if RAM_BUF_lookup(e) == TRUE:
    return FOUND;
else:
    layer = 1
    blkid = h0(e) % λ;
    pageid = h1(e) % δ;
    while layer < forest_height:
        subBF = bfload[blkid,pageid];
        if query(subBF,e) == TRUE:
            return FOUND;
        else:
            cpos=h1(e) ≫
            (LENGTH − layer_parent · ⌊log2 b⌋) % b;
            blkid = blkid * b + ROOT_NBLK;
            blkid = blkid − cpos;//to-be-checked blk
            layer += 1;
    return NOT FOUND
```

Figure 2. Pseudocode for key query routing under top-down traverse order.

A. Overview of FBF Design

FBF design partitions flash space into a collection of sub-BFs of flash-page sized and organizes them into a forest structure. Each sub-BF is an independent BF providing key query/insert operation. Within each layer, δ consecutive sub-BFs are packed into physical blocks in flash. The highest layer contains λ blocks. Each block has b children ($b \geqq 2$) except for the ones at the lowest layer. Correspondingly, in-RAM buffer space is partitioned according to those physical blocks at the lowest layer in flash space. Figure 1 presents a two-layer FBF, with each block of 1^{st} layer having 4 children at 2^{nd} layer.

Initially when the dataset size is small and could be fit in RAM, our design only allocates the highest layer (the root-layer) of the forest in RAM and behaves identical to a traditional in-RAM BF (in-RAM phase in Figure 1); As the dataset size grows and goes beyond the root-layer's capacity, the forest structure adds b children blocks to each block at the root-layer, forming a new layer of BFs in flash, to accommodate the dataset growth. Once the root-layer has been written into flash, the spared RAM space is switched into a buffer for key insertions (in-flash phase in Figure 1). The forest allocates a new layer in flash whenever the current lowest layer reaches its capacity.

To query a key e, it takes a three-step hashing procedure: (1) one hash function h_0 on e decides which block (*blkid*) to search for the key; (2) another independent hash function h_1 on e decides which sub-BF (*pageid*) to check for the key; (3) to check key e, multiple hash functions are applied to

identify a set of bit positions within this selected sub-BF. If all bit positions are set, the key is considered to have been seen before. Note steps (1) and (3) and identical to what presented in Section II, but step (2) guarantees all potentially checked bit positions are within one flash page. If e is not found at that block, FBF chooses one of the block's children to be searched, with the children offset determined by both *blkid* and *pageid* calculated in step (1) and (2) (5^{th}-to-last line, Figure 2). It is worth noting that since at any layer at most one flash page read is needed for the checking, the total number of flash accesses will be no more than the forest height for any key query. (Of course, if the key is found in the corresponding in-RAM buffer space ahead, then no flash access is needed) The key e is considered new if a sub-BF at the lowest layer is searched but the key e is yet to find. Then, FBF needs to insert e into the sub-BF for the future query. However, the update of sub-BF will be delayed by buffering the offsets of these bit positions into the RAM buffer space. The sub-BF will be updated, together with all other sub-BFs in the same physical block, only if the corresponding buffer space for its residing physical block is full.

Figure 2 sketches the key query procedure through the forest, with a top-down traverse order assumed. $\%$ stands for mod operation; $LENGTH$ stands for hash value length (e.g., 64); $forest\_height$ specifies the current height of the forest in terms of layers; $ROOT\_NBLK$ is the number of blocks initially allocated in root-layer; b is the branching factor. In the code, the 4^{th}-to-last line specifies the last child of the current blkid and the 3^{rd}-to-last line specifies the the blkid of the correct child it should go to in the while-loop.

Several characteristics of our FBF design worth some discussion: (1) aligning sub-BF to flash page size makes all to-be checked bit positions within the sub-BF to be read with just one flash access; (2) packing consecutive sub-BFs into physical blocks and partition the buffer space according to physical blocks eliminates random flash page writes and reduces associated block erase operations significantly. (3) Larger branching factor b will slow down the forest height growth rate, which is good for key query performance because as described above, the number of flash reads per key query is upper bounded by the forest height. However, as b becomes larger, more blocks needs to be buffered, which reduces the buffer space per block, resulting in more flash write operations; (4) FBF inserts new keys into the sub-BFs at lowest layer only, thus at anytime the in-RAM buffer space merely needs to serve sub-BFs at one layer of the forest instead of all layers. Therefore, our design manages to minimize the size gap between the in-RAM buffer and the associated BF on flash.

B. Buffer Space Managing Scheme

FBF minimizes flash write operations through the following Buffer space management design: (1) Instead of distributing insertions among all of blocks of the forest, FBF inserts new keys only into the lowest-layer blocks. This buffering strategy on average increases the caching space per buffered block by $1 + (b^{\alpha} - 1)/[b^{\alpha} \cdot (b-1)]$ times. (2) FBF proposes a set-list structure that stores updated bit positions within with a set and inserts all sets into a linked-list for fast insert/delete operation. Under this scheme the buffer space of each BF could grow to hold new updated bit positions until the entire buffer space is filled up. After that it selects a block with most bits buffered (the dirtiest block) to update. After all updated pages within the dirtiest block are written to flash the buffer space is reclaimed immediately for the use for other buffered blocks.

Even with the same single-layer BF design, it could be shown that this buffer space managing scheme could further reduce the total flush write operations by 50% on the same workload, comparing with its original scheme.

IV. Experimental Evaluation

In this section we evaluate our FBF design with a moderate-end SSD from OCZ Technology, using a real data deduplication workload. Among the evaluation, we measure the processing speed in terms of **ops/sec** (equivalent to the term *number of records processed per second* used in previous sections) which measures the number of key query/insertion operations accomplished per second. Accomplishing a key query/insert operation means for the key query either the BF answers found and returns or the BF answers not-found and a key insertion is done consequently. We also realize that some high-end SSD device like Fusion IO available on market. Nevertheless, Fusion IO itself takes up hundreds to thousands of megabyte of RAM space as device cache [2]. For example, As shown in the user-guide for 80GB ioXtreme, for 4KB recommended filesystem block size, the drive consumes 800 MB of RAM, which somehow invalidates our purpose of using Bloom Filter with flash memory to save RAM space. Also, such a large device cache sitting in the middle between the flash memory and the RAM-based buffering cache would buffer tons of data blocks, making it very difficult to justify the benefit of our own buffering cache design.

A. Description of Hardware Platform and Software Implementation

We implement the naive linear-chaining design, single-layer BF design as well as our proposed FBF design, sitting between application traces and storage device, with Python. The experiments are carried out on Linux build 2.6.32 SMP x86_64 machine. The machine has two 2.0 GHz cores, with 1 GB RAM. The SSD model is OCZ Agility SATA II [1], with the capacity of 120 GB.

B. Description of the Data De-duplication Workloads

A real workload (*vx-full*) is generated by content-defined chunking algorithm [13] which is commonly used in data de-duplication to produce data chunks. Within the workload, each record is a chunk-id, represented by a SHA1 [8] hash value of 160-bit length. A chunk-id is used to globally identify a data chunk in the storage system.

The *vx-full* (containing $164,766,619$ numbers of records, out of which 54.7% records are unique) consisting of chunk-ids derived from a networked primary file system shared by a

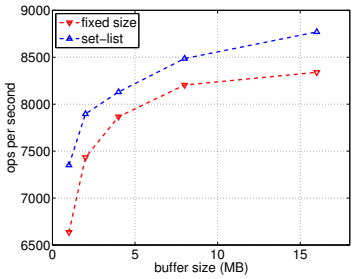

Figure 3. Processing speed vs. buffer size on *vx-9m* under single-layer design

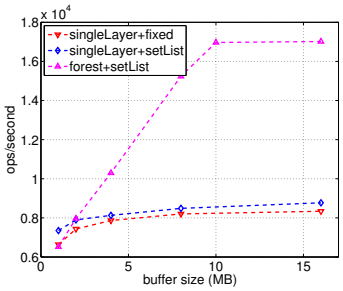

Figure 4. Processing rate vs. buffer size on *vx-9m* under two BF designs

group of software engineers, is a typical deduplication workload for primary file systems. It is considered an insertion-intensive workload because the percentage of unique chunk-ids determines the key insertion ratio through the process.

We plan to investigate our design and single-layer BF design with this workload. We also obtain some subsets from the *vx-full* workloads and present their notations as follows: *vx-9m, vx-20m*, and *vx-25m* contain the first 9, 20, and 25 million records of *vx-full* respectively. The number of unique records contained in *vx-9m*, *vx-20m* and *vx-25m* are $5,628,873$, $11,328,914$, and $14,163,022$ correspondingly. Also, the ratios of unique chunk-ids are 62.5%, 56.6% and 56.7%, which does not deviate much from 54.7%, the ratio of *vx-full*. In addition, we also have verified the recency-querying pattern does exist in *vx* workload, but again we omit verification detail in this paper due to page limitation. Both of these features indicate the reprentativeness of a subset of *vx-full*. In fact, further experimental results show that the results obtained from a smaller workload such as *vx-9m* are indeed consistent with the results obtained from *vx-full*, validating the representativeness.

C. Evaluation of Buffer Space managing Schemes

Table I
COMPARTMENT VS. SET-LIST SCHEMES ON *vx-20m*

buffer schemes	fixed-size compartment	set-list
number of flash writes	$2,024$	$1,053$
ops/sec	$8,405$	$8,657$

Table I compares the original buffer managing scheme [5] with our set-list structured one on *vx-20m*. **For fairness, both buffer managing schemes are tested with the single-layer structure** [5]. The in-flash layer contains 22 blocks of 1MB length, with 4MB in-RAM buffer space. As shown in the Table I, with higher processing speed ($8,657$ vs. $8,405$), our scheme requires roughly 50% number of flash write operations taken by the original buffer space managing scheme.

To figure out the impact of buffer size on processing speed for both cache space managing schemes, we increasingly double the buffer size from 1MB up to 16MB in 5 runs on *vx-9m* for both schemes and present the results in Figure 3. Each run is configured with 20 1MB blocks. Two results could be seen from this figure. First, both processing speeds increase as more buffer space is used. Second, set-list scheme uniformly outperforms fixed-size compartment scheme for all sizes, by up to 11% higher processing speed in terms of ops/sec at buffer size 1MB. Further examining the number of flash write operations shows a uniformly 50% reduction for our set-list buffer managing scheme.

D. Evaluation on Processing Speed over BF Designs

This section compares processing speeds of single-layer BF design and FBF design with *vx* workload. We run two sets of experiments with buffer sizes ranging from 1MB to 16MB on *vx-9m*. Figure 4 plots the processing speed vs. buffering cache size for forest-structure and single-layer BF designs on *vx-9m*. The 3 curves with downward-pointing triangle markers, with diamond markers, and with upward-pointing triangle markers represent the results of the single-layer BF design with fixed-size compartment buffer space managing scheme, results of single-layer BF design with set-list scheme and the results of our forest-structured BF design with set-list scheme respectively. Because the single-layer design does not support dynamic growth, for fairness, we fix block size to 1MB through all runs and configure each run with the same BF size of 20MB.

Several important conclusions could be drawn from Figure 4: (1) FBF design outperforms original single-layer design uniformly for all buffer sizes presented; (2) although the processing speed increases as larger buffer size becomes available for all designs, the slope of processing speed of the FBF design is much steeper than that of the single-layer one, showing that a much larger processing speed gain could be attained by FBF design when both designs are given larger RAM space; (3) if the allowed buffer size is less than 2MB, then the modified single-layer design (the one with set-list buffer space managing scheme) is better; otherwise, we should choose FBF design; (3) the slope for the FBF design curve flattens after the used RAM space bigger than 10MB. This phenomenon can be explained by the fact that the total amount of records could be processed entirely in RAM when the buffer size goes beyond 10MB. Hence further increasing buffer size would not improve the processing speed.

It is very interesting to point out that, by the boosting effect of the in-RAM phase, a FBF design which allows expansion could even outperform the single-layer design that could only handle static workloads (workloads have the maximum number of unique keys pre-determined) significantly (e.g. up

to 2 times faster) with the same amount of RAM space consumed!

We also compare the results of both single-layer design and FBF design on *vx-full*. With 40MB buffer size and 1MB block size used for both designs, the FBF design achieves 12105 ops/sec while single-layer design achieves 9390 ops/sec. This 30% higher speed attained by FBF design is consistent with the results we obtained from $vx - 9m$, provided that the total amount of records in *vx-full* would take up 160MB RAM space to memorize if traditional in-RAM BF was used, which is 4 times larger than in-RAM buffer size used in our design.

V. Related Works

The Bloom Filter structure was firstly proposed by B. H. Bloom as a compact representation of a static set with certain probability of false positives to serve set membership queries [4]. Since then, Bloom Filters are widely used in database applications [14] and are drawing increasing attractions from networking community recently [6], [11]. Data De-duplication has become another popular application area for Bloom Filters. Zhu et al. [15] utilizes a Bloom Filter to minimize chunk look-up latency; Navendu et al. [10] adopts Bloom Filters as a feature set of a data chunk; Lu et al. [12] takes a group of Bloom Filters to select out data chunks with more redundancy by filtering out low redundant ones.

Our FBF Bloom Filters design is inspired by [3], [9], each of which demonstrates the importance of representing dynamic growing set, and proposes a solution with RAM-based dynamic Bloom Filters. Nevertheless, two issues are not addressed with their design: (1) the linear look-up on each allocated BF causes the number of false positive errors significant higher than the calculated result; (2) how to design a Bloom Filter when its size exceeds the capacity of RAM as the dynamic set size grows.

Canim et al. [5] and Debnath et al. [7] propose similar designs to build a Bloom Filter with flash memory and uses a moderate amount of main memory space to buffer bit updates. In order to increase flash access locality, Canim et al. limits the size of each sub-BF on flash to 2MB and accesses the flash per sub-BF. Also, in order to amortize the cost of fetching 2MB from flash each time, it buffers queries into a request queue. Radically different from traditional Bloom Filters, this design does not guarantee that queries will be processed within a certain amount of time. In contrast to their work, our BF design does not buffer requests so as to immediately answer each query request in the order the request was received, in order to be useful for a data deduplication application. Furthermore, both designs presented in [5], [7] is only able to tackle static workloads, while our design targets at dynamic cases when the data set size could not be determined in advance.

VI. Conclusions

In this paper, we have proposed a forest-structured Bloom Filter design. It combines limited RAM space with a much larger flash memory space to form a compound BF. Our design splits a single large BF into multiple layers within a forest. Our design supports dynamic growing set and the querying overhead grows logarithmically as the BF size grows. Our experimental results show that the FBF design achieves 2 times faster processing speed with uniformly 50% less number of flash write operations compared with the state-of-the-art in-flash BF designs.

Acknowledgment

The authors would like to thank the anonymous reviewers for their valuable comments. This work was partially supported by grants from NSF (NSF Awards: 0960833 and 0934396)

References

[1] Ocz agility sata 2.5 ssd: http://www.ocztechnology.com.

[2] ioxtreme user guide for linux, version 3 for driver release 1.2.7. page 8, 12 2009.

[3] Paulo Sergio Almeida, Carlos Baquero, Nuno Preguica, and David Hutchison. Scalable bloom filters. *Information Processing Letters*, 101(6), 2007.

[4] Burton H. Bloom. Space/time trade-offs in hash coding with allowable errors. *Commun. ACM*, 13:422–426, July 1970.

[5] Mustafa Canim, George A. Mihalia, Bishwaranjan Bhattacharjee, Christian A. Lang, and Kenneth A. Ross. Buffered bloom filters on solid state storage. 2010.

[6] Francisco M. Cuenca-Acuna, Christopher Peery, Richard P. Martin, and Thu D. Nguyen. PlanetP: Using Gossiping to Build Content Addressable Peer-to-Peer Information Sharing Communities. In *IEEE International Symposium on High Performance Distributed Computing (HPDC)*, 2003.

[7] Biplob Debnath, Sudipta Sengupta, Jin Li, David J. Lilja, and David H.C. Du. Bloomflash: Bloom filter on flash-based storage. In *Proceedings of the 31th International Conference on Distributed Computing Systems*, ICDCS 2011, 2011.

[8] D. Eastlake, 3rd and P. Jones. Us secure hash algorithm 1 (sha1), 2001.

[9] Deke Guo, Honghui Chen, and Xueshan Luo. Theory and network applications of dynamic bloom filters. In *In Proceedings of the 25th Annual Joint Conference of the IEEE Computer and Communications Societies (INFOCOM)*, 2006.

[10] Navendu Jain, Mike Dahlin, and Renu Tewari. Taper: tiered approach for eliminating redundancy in replica synchronization. In *Proceedings of the 4th conference on USENIX Conference on File and Storage Technologies - Volume 4*, pages 21–21, Berkeley, CA, USA, 2005. USENIX Association.

[11] John Kubiatowicz, David Bindel, Yan Chen, Steven Czerwinski, Patrick Eaton, Dennis Geels, Ramakrishna Gummadi, Sean Rhea, Hakim Weatherspoon, Chris Wells, and Ben Zhao. Oceanstore: an architecture for global-scale persistent storage. *SIGOPS Oper. Syst. Rev.*, 34:190–201, November 2000.

[12] Guanlin Lu, Yu Jin, and David H. C. Du. Frequency based chunking for data de-duplication. In *Proceedings of the 2010 IEEE International Symposium on Modeling, Analysis and Simulation of Computer and Telecommunication Systems*, MASCOTS '10, pages 287–296, Washington, DC, USA, 2010. IEEE Computer Society.

[13] Athicha Muthitacharoen, Benjie Chen, and David Mazières. A low-bandwidth network file system. In *Proceedings of the eighteenth ACM symposium on Operating systems principles*, SOSP '01, pages 174–187, New York, NY, USA, 2001. ACM.

[14] Wei Wang, Haifeng Jiang, Hongjun Lu, and Jeffrey Xu Yu. Bloom histogram: path selectivity estimation for xml data with updates. In *Proceedings of the Thirtieth international conference on Very large data bases - Volume 30*, VLDB '04, pages 240–251. VLDB Endowment, 2004.

[15] Benjamin Zhu, Kai Li, and Patterson Hugo. Avoiding the disk bottleneck in the data domain deduplication file system. In *In Proceedings of the 6th USENIX Conference on File and Storage Technologies*, Berkeley, CA, USA, 2008. USENIX Association.