BURST: A Chunk-Based Data Deduplication System with Burst-Encoded Fingerprint Matching

Bo Peng
Shanghai Jiao Tong University
Shanghai, China
pengbo_michael@sjtu.edu.cn

Yingquan (Cody) Wu
SambaNova Systems
CA, USA
icodywu@gmail.com

Ziyang Zhang Jianguo Yao
Shanghai Jiao Tong University
Shanghai, China
{ziyang.zhang, jiango.yao}@sjtu.edu.cn

Xue Liu
McGill University
Montreal, Canada
xueliu@cs.mcgill.ca

***Abstract*—With the explosively-growing scale and heterogeneity of data, the legacy deduplication storage systems face more stern challenges in achieving high deduplication ratios and low overheads because these systems can remove redundancy successfully only if the fingerprint of an incoming data chunk perfectly matches the existing ones in the depository.**

In the paper, we present BURST, a novel online chunk-based data deduplication system, towards a more general scenario wherein an incoming chunk best matches one of the existing chunks by a short burst. BURST proposes a new scheme that creates head and tail fingerprints of each chunk and efficient delta burst deduplication designs to provide superior data reduction to the legacy deduplication systems and incur no CPU overhead. We implement and evaluate our open-sourced BURST in the practical storage platform, OpenZFS, whose full-blown production features render BURST ready to be deployed in real storage systems. The evaluations prove that BURST consistently achieves higher data reduction than the legacy approach in all datasets while achieving up to 2.06 $\times$ optimization. BURST can achieve higher data reduction as the legacy deduplication using a much larger (average) chunk size (4KB$\rightarrow$16KB), rendering higher performance and lower system cost.

***Index Terms*—file system, data deduplication, chunk-based storage, burst-encoded, fingerprint matching**

I. Introduction

As the global volume of data continues its exponential growth, with estimates projecting an increase from the current 64.2 to over 180 zettabytes by 2025 [1], the challenges associated with managing this surge in data have become a focal point for modern cloud infrastructures. The escalating data landscape is a primary contributor to the complexities of storage management in today's digital age. To address these challenges, various storage optimization technologies have been widely embraced, including compression, thin provisioning, space-efficient snapshot technology, and the latest advancements in deduplication technologies (e.g., [2]–[6]).

Deduplication, in particular, has been a central focus of research for an extended period. Numerous studies in the field have successfully demonstrated its efficacy in reducing storage space by eliminating duplicate data in disk storage I/O [3], [4], [7]–[10], while some deduplication techniques have proven effective in minimizing redundant data transmission in network environments [7], [11]–[14]. Moreover, recent innovations in deduplication techniques have extended their applications to enhance the longevity of flash mediums [15], [16].

The legacy deduplication systems are usually based on chunk-level data deduplication, which is highly performance-critical and deduplication-efficient to the chunk size choices and fingerprint computation designs. Typically, a chunk-level deduplication system splits the input data stream (e.g., backup files, program archives, virtual machine images) into multiple data chunks. Each chunk is uniquely identified and discerned through the application of a cryptographically-secure hash signature, commonly known as a fingerprint [7], [17], such as SHA-1 or SHA-2 [18]. The chunk size can be fixed like file blocks or variable-sized units determined by the data content. For example, fixed-size chunks (interchangeably, block) have been widely adopted in modern primary deduplication storage systems [3], [4], [9], where maintaining low latency is one of the most critical design goals. Moreover, most mainstream file systems generate blocks with a fixed length, often set at 4K-Bytes, with an upcoming transition to a 16K-Byte length. In contrast, the prevalent variable-length segmenting approach is through computing Rabin-fingerprint [19] for each sliding window of data bytes and to set chunk boundary when the associated Rabin-fingerprint meets certain criteria, e.g., a number of least significant bits are all zeros [20]. However, the existing variable-length segmenting methods usually compute certain metric over a consecutive number of bytes associated with each byte (e.g., [20]–[22]), which is computationally costly as the number of computed fingerprints is as large as the data length, for example, in Rabin-fingerprint segmenting. Therefore, how to achieve a promising deduplication ratio and low overhead for the more widely used fixed-length chunk systems remains a significant challenge in data deduplication research.

In this paper, we introduce a generalization of the traditional perfectly matched chunk deduplication to **imperfectly-matched deduplication**, where the delta between similar chunks can be characterized as a short burst. Typically, a delta burst is defined as the replacement of an interval of data (potentially NULL) with a new interval of data (also potentially NULL), and the replacement can be the deletion or insertion of an interval of data, or the substitution of an interval of data with a new interval of possibly different length. The motivation for this generalization stems from our observations of revision locality. Specifically, when revising an old version of a file, edits tend to occur in bursts, meaning that if one byte is modified, its neighboring bytes are likely to be modified as well. While a revised file may contain multiple bursts, we reasonably assume that each partitioned data chunk may contain up to one burst of revision. Our chunk-level burst hypothesis can also be viewed as a deduction from the extensively researched file-level sparse delta hypothesis in terms of delta encoding [23], [24]. It is essential to note that two burst-alike chunks may differ in their lengths, in contrast to scenarios involving identical chunks. Subsequently, we present BURST, a novel burst-encoded deduplication method. BURST efficiently identifies a delta burst from existing data chunks by creating two additional key-value tables: the head fingerprint and tail fingerprint, associated with the head and tail of the data chunk, respectively. Through extensive evaluations, we demonstrate that the proposed BURST method is particularly advantageous when dealing with large chunk sizes. In summary, this paper mainly makes the following contributions.

1) We investigate deduplication on imperfectly matched data chunks, and particularly characterize the delta as a single burst based on real data characteristics.
2) We present BURST, a novel chunk-based deduplication system with a burst-encoded fingerprint matching. BURST efficiently identifies a delta burst from existing data chunks through creating head fingerprint and tail fingerprint in associated with the head and tail of data chunk respectively. Notably, by selecting a small number of head and tail bytes as their respective fingerprints, this approach incurs no additional CPU overhead compared to the legacy method.
3) We implement the BURST system prototype [1] based on the open-source storage platform OpenZFS [25] . The wide popularity of OpenZFS along with its full-blown production feature set renders the build-in BURST ready to be deployed in the real storage systems.
4) Our evaluations reveal several key advantages of BURST over traditional deduplication methods. BURST consistently outperforms the legacy method in deduplication ratios across all datasets (up to 2.06 $\times$), with the larger dataset exhibiting a more pronounced gap. Moreover, BURST demonstrates superior data reduction with a 16KB chunk size, surpassing the performance of the legacy method with a 4KB chunk size, which not only enhances overall performance but also reduces system costs.

The rest of this paper is organized as follows. Section II provides a comprehensive exploration of the observations and characterizations of bursty delta at the chunk level. Section III details the design of the BURST deduplication framework. Section IV outlines the integration of the proposed BURST into the widely used open-source platform, OpenZFS. Section V presents the evaluation results derived from various datasets, showcasing the effectiveness of the BURST framework. Section VI includes the technical discussions and future works about BURST. Section VII conducts a survey of related works in the field. Finally, Section VIII concludes the paper.

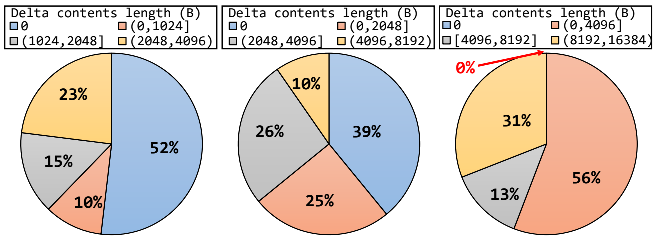
[1]Open-sourced at https://github.com/old-memories/zfs/tree/burst-dedup

II. Background and Motivations

Legacy deduplication systems grapple with a crucial trade-off between data reduction and performance when deciding between a fixed or variable average chunk size. It is evident that smaller chunks lead to a more effective recognition of duplicate chunks, resulting in a superior deduplication ratio. However, the utilization of smaller chunks in deduplication systems introduces challenges. Systems employing smaller chunks must contend with processing a higher number of chunks during each deduplication loop, thereby diminishing the data I/O performance of the storage system. Furthermore, the use of smaller chunks necessitates larger storage footprints for their metadata, as a reduced amount of total user bytes can be cached in a given amount of memory. This situation leads to an increased number of updates to the chunk index. It is important to acknowledge that any data structure scaling with the number of chunks imposes limitations on the overall capacity of the storage system when smaller chunks are employed. Considering the typical constraints of commodity servers with limited physical memory resources, the (average) chunk size becomes a significant factor influencing the cost of the system. In essence, while a smaller (average) chunk size may yield gains in data reduction, it concurrently incurs losses in terms of performance, capacity, and overall system cost.

The individual files are represented as backup images consisting of a large number of component files in backup systems, . Successive backups of the same file system rarely result in entirely identical files, as even a single addition, deletion, or modification of content can swiftly alter the entire image composition. To address this, content-based chunking has been employed to preserve identical chunks despite shifts in content. However, in legacy deduplication, the challenge arises when storing entire chunks for those cases where shifts or modifications occur, regardless of the magnitude of the change or delta. It is evident that in legacy deduplication, larger (average) chunk sizes lead to poorer data reduction, as a higher percentage of chunks are affected by these changes. Conversely, if an effective mechanism is in place to identify and store these deltas, the amount of deduplicated data remains unchanged even with larger chunk sizes.

To examine the delta characteristics within real-world datasets, we crawl HTML, JS, and CSS files from a news website using Httrack [26], creating an archival dataset as a typical representative of cloud computing workloads. Many HTML files within this dataset share similar structures as they are constructed from templates. The archival dataset was divided into 4KB, 8KB, or 16KB chunks. We conducted a comparative analysis by assessing the content of each chunk against the content of all other chunks to identify the closest match in terms of the shortest burst length. In our analysis, we consider the delta or difference between two chunks as a single burst. Therefore, our measurement focuses on burst length rather than Hamming distance (i.e., the number of different bytes). Figure 1 illustrates the distribution of burst lengths among similar chunks within the dataset, where '0' denotes



(a) 4KB chunk size (b) 8KB chunk size (c) 16KB chunk size

Fig. 1. The length distribution of the delta contents in the web archival.

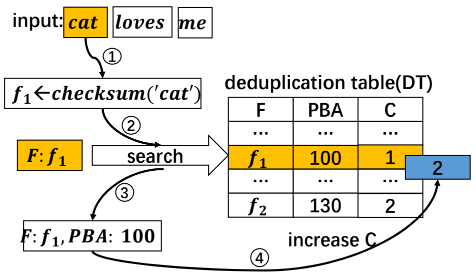


Fig. 2. An example of legacy deduplication process.

a burst length of zero, signifying a perfect match. The figure reveals three discernible trends as chunk size increases. (1) The percentage of perfectly deduplicated chunks decreases. When the chunk length is 16KB, there is no perfectly matched block. (2) The percentage of imperfectly deduplicated chunks increases. As the chunk length increases from 4KB to 16KB, the percentage of imperfectly-matched chunks increase from less than half to all. (3) The percentage of imperfectly-matched chunks with a burst length not exceeding 1/4 of the chunk length increases among all the imperfectly-matched chunks. For instance, with the chunk size as 16KB, **56%** of burst lengths are smaller than 4KB.

Hence, it is reasonable to define the delta between the two most similar chunks as a short burst. Building upon these insightful observations and characterizations, the next section endeavors to introduce an innovative burst-encoded chunk-based deduplication framework. This framework is designed to efficiently identify and store short bursts, resulting from minor changes, in incoming data chunks. The ultimate goal is to preserve data reduction benefits even when dealing with significantly larger chunk sizes.

III. BURST DESIGN

A. Terminologies for Deduplication System

As BURST is a novel design of burst encoded deduplication framework for storage and file systems, we first introduce the following terminologies to facilitate a concise description of BURST's write, read, and delete procedures, where an uppercase letter represents a vector, and a lowercase letter denotes a scalar.

Deduplication Ratio (dedup-ratio) is defined as the ratio of the data size of pre-deduplication over the size of post-deduplication. For example, if there are two identical 4KB chunks and another 4KB chunk with different content, then the deduplication results in dedup-ratio= $\frac{4KB*3}{4KB*2} = 1.5$.

Data Chunk (D) represents a chunked raw data vector, whereas a **Data Block** (B) is a data structure which includes a data chunk and its metadata, such as physical block address, size, modified time, fingerprint, etc.. In the sequel, "block size" is used interchangeably as "chunk size".

Logical Block Address (LBA), denoted by l, is provided by most storage systems for upper applications to reference each data block. Applications only need to address a specific LBA in order to read or write the corresponding data block. In Linux, filesystems utilizes a specific inference to an inode and it may refer to many blocks.

Physical Block Address (PBA), denoted by p, is used to address data chunks in storage devices. The read procedure requires a PBA to read one specific block while the write procedure returns a PBA addressing the data chunk allocated and stored in storage devices. A storage system maintains an LBA table which maps LBA to PBA. For conciseness, we shall omit this table in the subsequent algorithmic procedures.

Fingerprint (f) (empirically unique) identifies a data chunk which is usually a log number or string generated by checksum algorithms. There are two types of fingerprints, namely, weak and strong. When strong fingerprint (e.g., SHA-1, SHA-2) is applied, two blocks with identical fingerprints are safeguarded to be identical (noting the probability of data collision is way below storage failure tolerance). On the other hand, when weak fingerprint (e.g., CRC, or ECC parity) is applied, the probability of two different blocks with identical fingerprint can no longer be neglected. Therefore, two data chunks must be read and matched byte-wise if two weak fingerprints are matched (e.g. [16], [27]). In our context, we will focus on strong fingerprint for conciseness.

Reference Counter (c) records the deduplication times of a data block. When a block is successfully deduplicated, the counter increases by 1; When a deduplicated block is deleted, the counter decreases by 1. When it becomes 0, the corresponding data block is set to be erased from the device.

Deduplication Table (DT) refers to a key-value table containing multiple entries $dte \triangleq \{f, p, c\}$. We use dte_r to define a **DT** entry which is referenced and used to deduplicate a incoming data chunk.

We use Figure 2 with the interaction among the previous terminologies to illustrate the flow of a deduplication write procedure in the legacy data deduplication system, which usually utilizes a deduplication table (**DT**) which maps each unique fingerprint (f) to its PBA (p) in conjunction with a reference counter (c).

B. Burst-Encoded Deduplication Methodology

Since BURST is a novel *burst*-encoded chunk-based data deduplication framework, we first formally define the data *burst* and basic interaction between the data chunks and *bursts*.

A *burst* is denoted by β, as constituted by three elements,

$$\beta \triangleq \{[start,\ \ end);\ \ data\}, \tag{1}$$

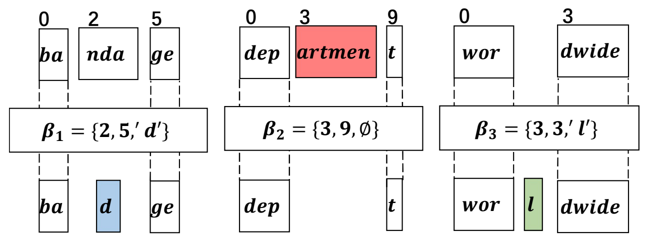


Fig. 3. Three types of bursts between burst-alike blocks.

where *start* and *end* respectively denotes the starting and ending index of the incurred burst revision respectively, and *data* denote the substitute burst data. Note that the ending index is exclusive.

We give a few examples to clarify the *burst* definition in Figure 3. $\beta_1 = \{2, 5, 'd'\}$ represents that an incoming chunk replaces the three bytes of reference at positions 2-4 with one bytes 'd' (at the reference chunk location 4); $\beta_2 = \{3, 9, \emptyset\}$ represents that an incoming chunk deletes six bytes at its reference chunk's indexes 3-8; $\beta_3 = \{3, 3, 'l'\}$ represents that an incoming chunk inserts one byte 'l' at reference chunk position 4.

We introduce a pre-length and a post-length concept when we want to discuss the length of a burst. The pre-length $len^-(\beta)$ is defined by

$$len^-(\beta) \triangleq \beta.end - \beta.start, \tag{2}$$

which represents the length of data affected by the new burst. and the post-length $len^+(\beta)$ is given by

$$len^+(\beta) \triangleq len(\beta.data), \tag{3}$$

which represents the actual length of the new incoming burst.

From Figure 1, we can find that over 50% of the burst has a post-length less than $1/4$ of the size of the original chunk B, that is

$$len^+(\beta) \leq \frac{B.size}{4}. \tag{4}$$

Specifically, $len^-(\beta) = 0$ indicates that the new burst is inserted right after $\beta.start$, and $len^+(\beta) = 0$ indicates that original data in the interval $[\beta.start, \beta.end)$ is deleted.

Based on the *burst* concept, we define that data chunk D_1 and D_2 are **burst-alike** if we can use a short burst β to update D_1 to D_2. Since the BURST deduplication system is designed towards a more general scenario wherein an incoming chunk best matches one of the existing chunks by a short *burst*. In order to get shorter and more accurate *bursts* between different data chunk in the chunk-based deduplication systems, we partition a chunk D into three sub-chunks,

$$D = [D_h, D_c, D_t], \tag{5}$$

where D_h, D_c, D_t refers to the head, center, tail sub-chunk, respectively. We set the lengths of D_h, D_t equal and prefixed for all chunks, denoted by τ. The following theorem lays out the foundation of burst detection.

Theorem 1: Let an incoming data chunk D be a bursty revision, given by β, of a reference chunk $\bar{D}$ such that the pre-burst length satisfies

$$len^-(\beta) \leq len(D) - 2\tau. \tag{6}$$

Then, either their head sub-chunks or tail sub-chunks are identical, i.e.,

$$D_h = \bar{D}_h, \quad \text{or } D_t = \bar{D}_t. \tag{7}$$

Proof: If the burst does not occur in the head sub-chunk, then apparently $D_h = \bar{D}_h$. Now assume the burst occurs in the head sub-chunk. Note the end burst index is bounded by $\tau - 1 + (len(D) - 2\tau) = len(D) - \tau - 1$, by assuming the longest burst starts at the last byte of head sub-chunk. Therefore, the burst does not overlap with the tail sub-chunk, yielding $D_t = \bar{D}_t$.

The above theorem indicates that a strong head fingerprint and tail fingerprint can be deployed to effectively identify burst-alike chunks in a large system, as extensively deployed in legacy deduplication systems. Accordingly, We compute two fingerprints respectively: head fingerprint: $f_h = checksum(D_h)$ and tail fingerprint: $f_t = checksum(D_t)$, through one checksum algorithm such as SHA-2.

The choice of the head (tail) sub-chunk length, τ, must take into account of the following constraints. (1) τ must be smaller than half of minimum chunk length; (2) τ must be no shorter than the length of strong fingerprint, for example, SHA-2; (3) τ is proportional to the computational cost of two fingerprints. Therefore, in a specific case where we choose SHA-2 as checksum algorithm and set the size of a fingerprint as 32B, we can set τ as 32B and let head (and tail) fingerprints be the data of head (and tail) sub-chunks to help eliminate extra CPU overhead over the legacy design.

We use **HeadDT** (**TailDT**) to define the head (tail) deduplication table. The entry in **HeadDT** (**TailDT**) is denoted by *head-dte* (*tail-dte*), contains a head (tail) fingerprint, address of its chunk's dt in **DT** and a counter(c). A block B is the main data structure for read and write requests which contains a data chunk ($data$, or D), a fingerprint (f), a PBA (p, maybe NULL), its size ($size$), type ($type$, such as '$burst$') and other attributes not related to deduplication. Mathematically,

$$B \triangleq \{D, f, p, size, type\}. \tag{8}$$

C. Deduplication FS Interfaces

1) BURST Write: Figure 4 shows an example that roughly illustrates the steps of detecting and writing a burst-alike block with respect to its reference block. For simplicity it only shows operations on the head sub-chunk. In our BURST design, the above operations are carried out on head and tail sub-chunks respectively. We describe each step of the write operations in detail. BURST first computes the head fingerprint f_h and search f_h in **HeadDT**. In this example, BURST gets one matched *head-dte* related to a unique block ("bandage") written to storage devices before. While writing this unique block, new entries for **DT**, **HeadDT** and **TailDT** are created and inserted, respectively. Algorithm 1 describes

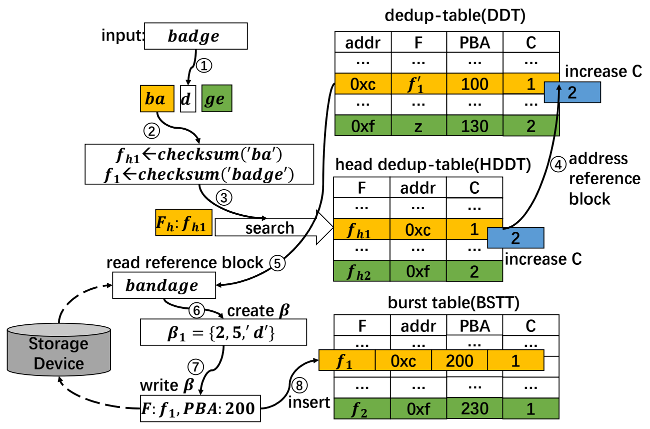


Fig. 4. Steps of writing a burst-alike block in BURST framework.

how to write a unique block. The $writeDev()$ function, whose input parameter is a data block (B) and return value is a PBA (p), serves as the write interface provided by storage devices.

Algorithm 1: Write a unique Block

```
1  void writeUniqueBlock($B$, **DT**, **HeadDT**, **TailDT**)
2  begin
3      $B.p \leftarrow$ writeDev(B) ;
       // write to physical devices
4      $dte \leftarrow \{B.f, B.p,\ 1\}$ ;
5      insertEntry ($\mathbf{DT}, dte.f, dte$) ;
6      $head\text{-}dte \leftarrow \{B.f_h, dte,\ 1\}$ ;
7      insertEntry($\mathbf{HeadDT}, head\text{-}dte.f, head\text{-}dte$) ;
8      $tail\text{-}dte \leftarrow \{B.f_t, dte, 1\}$ ;
9      insertEntry($\mathbf{TailDT}, tail\text{-}dte.f, tail\text{-}dte$) ;
10 void insertEntry(**T**, key, entry);
```

We can find that the two block have the same heads ("ba") in this example. We call this original unique block with "bandage" as the reference block of this burst-alike block "badge". A *head-dte* entry that contains address of the dte_r related to the reference block will be insert into **HeadDT**. Then, we increase the reference counter c of reference dt_r and *head-dte* to record this match. Besides, we mark the type of the burst-alike block as '*burst*' to distinguish it from unique blocks. Next we issue a read request to get the entire chunk addressed by PBA(p) in that dt_r. As soon as the read request completes, we create a burst ($\beta_1 = \{2, 5, 'd'\}$). Then we replace original data chunk ("badge") by $\beta_1('d')$ and issue a new write request to storage devices. Size of a β is in units of hundred bytes while the size of the original data chunk could be 4KB size so we achieve higher deduplication ratio. After the lower layer returns one PBA(p) addressing β_1, we create a new burst entry (bte) and insert it into a burst table (**BT**), which is a deduplication table data structure similar with **DT, HeadDT, and TailDT**. A bte includes B_1's fingerprint, PBA (p) returned before, address of dt_r and a counter (c).

We further discuss two interesting corner cases where meticulously handling them enables to achieve higher deduplication ratio and throughput.

The first case is to write a new block B_2 which has the same data content to a burst-alike block B_1 already written to storage devices. We prefer to perfectly deduplicate B_2 instead of BURST deduplicating this block. As the legacy deduplication method does, We additionally search key $f_2 = checksum(B_2)$ in the **BT** table while writing B_2. Then we find one **BT** table entry bte related to B_1. Therefore B_2 is perfectly deduplicated and we increase c of the corresponding bte by 1.

The second case is that two burst-alike blocks sharing the same reference block. It happens while writing multiple versions of a file and two new versions are forked from one original version. If we allow too many burst-alike blocks to share the same reference block, the deduplication ratio is lower but write latency is higher because more read requests for reference block are required. Therefore, we note that the number of burst-alike blocks which share the same reference block is related to a trade-off between deduplication ratio and I/O throughput. We design reference counters c_h, c_t (initialized to 1) for *head-dte* and *tail-dte*, respectively, and check whether it exceeds a maximum counter parameter (c_max). If it exceeds c_max, we stop deduplication and the burst-alike block is viewed as a unique-block. The parameter choices of c_max will be discussed in the evaluation section again.

Algorithm 2 describes the algorithmic procedure about how BURST write a burst-alike block. We use **XDT** (*x-dte*) refers to either **HeadDT** (*head-dte*) or **TailDT** (*tail-dte*) when writing a burst-alike block. The $ReadDev$ function, whose input parameter is a PBA (p) and return value is a block data structure (B), serves as the read interface provided by storage devices. Note that we do not create a dte for a burst-alike block as a unique block. If we do so, there could be nested relationships between blocks. This may lead to high write latency because three read requests have to be issued for a "three-depth" burst-alike block. Besides, it brings additional complexity for implementation.

Algorithm 2: Write A Burst-Alike Block

```
1  void writeAlikeBlock($B$, **XDT, BT**,$x\text{-}dte$)
2  begin
3      if $x\text{-}dte.c \leq c\_max$ then
4          writeBurst($B, x\text{-}dte, \mathbf{BT}$) ;
5      return
6  void writeBurst($B, x\text{-}dte, \mathbf{BT}$)
7  begin
8      $x\text{-}dte.c$++ ;
9      $x\text{-}dte.dt_r.cc$++ ;
10     $B.type \leftarrow$ '*burst*' ;
11     $B_r \leftarrow$ readDev($x\text{-}dte.dt_r.p$) ;
12     $\beta \leftarrow$ makeBurst($B.data, B_r.data$) ;
13     $B.data \leftarrow \beta$ ;
14     $B.p \leftarrow$ writeDev(B) ;
15     $bt \leftarrow \{B.f,\ B.p,\ x\text{-}dte.dt_r,\ 1\}$ ;
16     insertEntry($\mathbf{BT}, bte.f, bte$) ;
```

Herein we give an example to elaborate the second case. Assume we set lengths of head and tail(τ) to $\frac{len(D)}{4}$ and c_max

to 1. Block $B_r =' hhaaaaaa'$ is a unique block already written to storage devices. black We are now going to write block $B_1 =' hhbbbbbb'$. We find B_1 is burst-alike toward B_r after searching in **HeadDT** and there is $\beta = \{2, 8, ' bbbbbb'\}$. Therefore the reference counter $head - dte.c$ related to B_r increase by 1. Assume another write block is $B_2 =' hhcccccc'$. We find B_2 is burst-alike to B_r too, after searching **HeadDT**, but *head-dte.c* related to B_r exceeds c_max. So we give up creating another burst but directly write the original data chunk into storage devices. Accordingly, we create new entries *head-dte* and *tail-dte* referring to B_2 and reset their reference counters to 0 (note *head-dte* previously refers to B_r). If the next write block is $B_3 =' hhdddddd'$, it will find B_2 its reference block in **HeadDT**. Note that we do not lose the relation between B_r and B_1 because a *bte* related to B_1 keeps it. Finally, we use Algorithm 3 to introduce the write procedure of BURST deduplication system based on the previous burst and block write interfaces.

Algorithm 3: BURST Write

```
void burst_write(B)
begin
    $f \leftarrow$ Checksum($B.data, 0, B.size$);
    $f_h \leftarrow$ Checksum($B.data, 0, \tau$) ;
    $f_t \leftarrow$ Checksum($B.data, B.size - \tau, B.size$);
    if $dte \leftarrow$findEntry($\mathbf{DT}, f$) then
        updateEntry($B, dte$) ;
        return
    if $bte \leftarrow$ findEntry($\mathbf{BT}, f$) then
        updateEntry($B$, $bte$) ;
        return
    if head-dte $\leftarrow$ findEntry($\mathbf{HeadDT}$, $f_h$) then
        writeAlikeBlock($B, \mathbf{HeadDT}, \mathbf{BT}$, head-dte) ;
        return
    writeUniqueBlock($B, \mathbf{DT}, \mathbf{HeadDT}, \mathbf{TailDT}$) ;
void UpdateEntry($B$, $entry$)
begin
    $entry.c$++ ;
    $B.p \leftarrow entry.p$ ;
```

2) BURST Read: To read a block B, we first check its type, which may be either "burst" or "unique". If B is "burst", we search its key $f = checksum(B)$ in **BT** to obtain the matched bt. This bte contains address of dte related to B's reference block, B_r. Then we issue two read requests to storage devices, one to read B_r whose PBA is stored in the dte and other to read β whose PBA is stored in B. After two read requests complete, We reconstruct the original chunk of B by patching β on B_r. The complete read procedure is described in Algorithm 4.

3) BURST Delete: To delete a block B, we first check its type whether "*unique*" or "*burst*", as the read procedure does. For a burst-alike block, we search $f = checksum(B)$ in **BT** and decrease both counters of its bte and the reference block's dte_r by 1. Data chunk will be marked for deletion only when either one of the two counter becomes zero. The complete delete procedure is described in Algorithm 5.

Algorithm 4: BURST Read

```
void BURST_Read($B$)
begin
    if $B.type ==' burst'$ then
        $f \leftarrow B.f$ ;
        $bt \leftarrow$ find($\mathbf{BT}, f$) ;
        $B_r \leftarrow$ readDev($bt.dt_r.p$) ;
        $B \leftarrow$ readDev($B.p$) ;
        $\beta \leftarrow B.data$ ;
        $B.data \leftarrow$ makeChunk($B_r.data$, $\beta$) ;
    else
        $B \leftarrow ReadDev(B.p)$
```

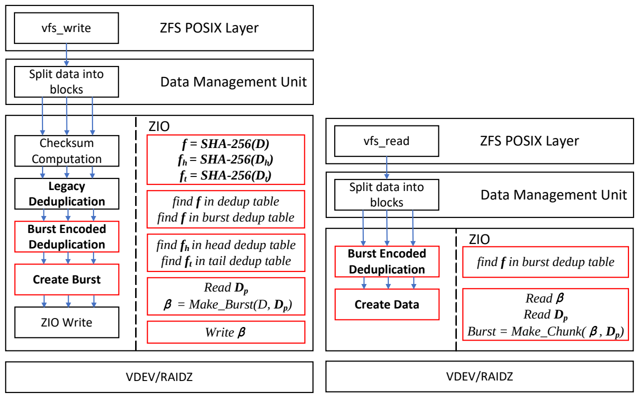
Algorithm 5: BURST-Delete

```
void BURST_Delete($B$)
begin
    $f \leftarrow B.f$ ;
    if $B.type ==$ "$burst$" then
        $bte \leftarrow$ findEntry($\mathbf{BT}$, $f$) ;
        $dte \leftarrow bte.dte_r$ ;
        $bte.c \leftarrow bte.c - 1$ ;
        $dte.c \leftarrow dte.c - 1$;
        if $bte.c == 0$ then
            $free(B.p)$ ;
        if $dte.c == 0$ then
            $free(dt.p)$ ;
    else
        $dt \leftarrow$ findEntry($\mathbf{DT}, f$) ;
        $dt.c - -$ ;
        if $dt.c == 0$ then
            $free(B.p)$ ;
```

IV. BURST Implementation

In this section, we demonstrate how to implement our proposed BURST deduplication framework on an open-source storage platform, OpenZFS [25]. It includes the functionality of both traditional file systems and volume manager. It gained tremendous popularity because of many advanced and desirable features, including: (1). protection against data corruption; (2). integrity checking for both data and metadata; (3). Continuous integrity verification and automatic "self-healing" repair; (4). support for high storage capacities — up to 2^{128} bytes; (5). space-saving with transparent compression using LZ4, GZIP or ZSTD; (6). hardware-accelerated native encryption; (7). efficient storage with snapshots and copy-on-write clones; (8). efficient local or remote replication.

ZFS supports multiple data chunk sizes such as 512B (minimum), 1KB, 4KB, 16KB and 128KB (maximum). Users



(a) Write procedure (b) Read procedure

Fig. 5. BURST implementation on OpenZFS.

can choose one size through the command line. This means block in ZFS are fixed-size. However, ZFS supports allocating blocks with different sizes. Therefore, while writing a burst-alike block, we are able to replace the original chunk by a smaller burst. For example, when writing an incoming 4KB block and a 120B burst is created, we roundup this 120B burst data to 512B by padding zeros and write this 512B burst instead of the original 4KB chunk.

ZFS is constituted with many subsystems. Herein we introduce some important subsystems from top to bottom. **ZFS POSIX Layer** implements some VFS (Virtual File System) interfaces. We focus on vfs_write, vfs_read and vfs_unlink which are three primary operations on data deduplication. **Data Management Unit (DMU)** transforms input data into blocks which is prepared to be processed concurrently. **ZIO** is the main layer in ZFS which computes the checksums and deduplicated blocks. **VDEV/RAIDZ** is the lowest layer in ZFS which allocates chunks and interacts storage drivers.

We proceed to describe our implementation of write, read and delete procedures on ZIO layer of ZFS. For simplicity, data is chunked into fixed-size blocks such as 4KB, 8KB or 16KB. Figure 5(a) shows the BURST write procedure in ZFS on writing a burst-alike block. After splitting input data into blocks in DMU layer, ZIO layer handles these blocks concurrently in five steps. These steps include computing checksums, searching in deduplication tables, creating the burst and writing burst to VDEV/RAIDZ layer.

To reduce the overhead of locking, we can use only one mutex implemented as $mutex\_lock$ in Linux kernel to lock the corresponding entries in the multiple deduplication tables (including **DT, HeadDT, TailDT and BT**) when operating read or write accesses on specific data chunks. The ZIO layer issues an asynchronous write request to VDEV/RAIDZ layer while writing a unique block. Then, the address PBA (p) of it is assigned in a callback function. Therefore, every incoming block referring to this block is queued before the write request completes. After the write request completes, we assign PBA (p) for every block queued and increase c of dt.

One corner case happens when issuing a read request to read the reference block for creating a new burst. If the reference block has not been written to storage devices, we choose to cancel this read request and skip deduplication. Although this happens rarely thus resulting in a negligible loss of deduplication, some tricky conditions(e.g. delete the reference block when creating a burst) is avoided and the worse-case latency is significantly shortened.

Figure 5(b) shows the BURST read procedure in ZFS while reading a burst-alike block. Steps in ZIO layer include searching in deduplication tables, reading burst-alike and reference blocks from VDEV/RAIDZ layer and creating the original chunk. The BURST delete procedure in ZFS decreases the reference count c by 1 in **BT** and **DT**. When c becomes zero, a unlink request is issued to VDEV/RAIDZ layer, indicating the chunk is marked free for future reallocation.

V. Evaluation

A. Experimental Setup

Experimental platform. We construct and implement our method based on OpenZFS [25] (version 0.8.5) according to the discussions in the Implementation Section shown before. Our implementation adds about 500 lines of C code based on the native OpenZFS which is compiled along with other components under the default compilation configuration of OpenZFS. In the following evaluations, to generate write and read requests for OpenZFS, we use the "cp" command to copy files of the datasets from a **tmpfs** directory to a **zfs** directory. In this process, the **tmpfs** is adopted instead of a disk filesystem(e.g. ext4) to avoid additional disk I/O latency. Also, the following evaluations are configured to perform on a single machine to simplify the complexity of the environment deployment. Table I shows our detailed information of the experimental platform.

TABLE I
Configuration of the experimental platform.

CPU	Intel Xeon E5-2680V2@2.80GHz
OS	Ubuntu 18.04 64bit(Linux 4.15)
Memory	64GB
Disk	Intel Optane 900P 480GB
Filesystem	OpenZFS 0.8.5

Configurations for deduplication. The legacy fixed-size chunk-based deduplication method is adopted as the baseline for evaluating BURST deduplication, which has already been implemented by OpenZFS. In legacy and BURST deduplication performance, block size is one relevant parameter, which is set to 4KB, 8KB or 16KB for evaluation. In BURST deduplication ,there are two more relevant parameters to set, that is, size of head & tail and c_max. We set head & tail length (τ) to a fixed value of 1/8 block and the actual value should be changed with block size. c_max is one parameter shown in our design, which is set to 1, 3 or 255 (big enough for our datasets) in the following evaluation.

Performance Metrics. I/O throughput, deduplication ratio and utilization of CPU and Memory are the selected performance metrics for the following evaluations. Throughput of read and write is defined as $\frac{Total\ Size}{Time}$, where $Total\ Size$ refers to the size of files in one dataset and

$Time$ is measured by the "system time" section of the "time" utility (not time command of bash), considering that our code runs in the kernel. Deduplication ratio is defined as $\frac{Total\ Size\ Before\ Deduplication}{Total\ Size\ After\ Deduplication}$. Since our implementation of OpenZFS has already included the computation of deduplication ratio, so this metric can be recorded directly from the command line. Also, CPU & Memory utilization is measured by Linux tools including "Top", "Perf" and "Slabtop" . We use "Slabtop" to record the memory cost because deduplication metadata is allocated by "kmem_cache_alloc" of Linux Slab Allocator.

Datasets. We choose five different real world datasets and randomly generate data deduplication workloads from these datasets. These selected datasets stand for different characteristics for the mainstream workload scenarios in cloud systems,and these workloads can help to demonstrate the performance and efficiency of our BURST system.

- **Version-Small**, which contains the source codes of GDB [28] 8.0 to 10.2 (12 versions totally).
- **Version-Large**, which contains the source codes of GCC [29] 7.1 to 8.5 (10 versions totally).
- **Version-Range**, which contains the source codes of Linux [30] 4.4 to 5.10 (nearly 10-year range).
- **Backup-VM**, which contains 5 backups of an Ubuntu 18.04 raw image [31].
- **Web-Archive**, which contains the CSS, JS and HTML files from a university website crawled by Httrack [26].

Figure 6 demonstrates the deduplication ratio and throughput performance of BURST against the legacy deduplication file systems when we generating data deduplication workloads with the different real-world datasets.

B. Deduplication Ratio

Figure 6(a) shows that BURST can achieve higher dedup-ratio than the baseline (original OpenZFS). However, the performance gap depends on datasets, for example, BURST dedup-ratio is marginally better than the baseline in Version-Small, Version-Large and Backup-VM test cases. This is mainly due to small dataset size. In Version-Small and Version-Large, there are only thousands of new lines of code in a new versions comparing to an older version, so burst-alike blocks are fewer than identical blocks; in Backup-VM, only a small portion of an image file may be changed because we perform the backup at a small time period interval. On the contrary, Figure 6(a) shows that BURST can perform much better than baseline especially with large block size. With 16KB block size, the baseline cannot deduplicate anything while BURST effectively reaches a dedup-ratio of **2.06** in the Web-Archive cases. The reason why BURST performs well with Web-Archive is that there are many similar HTML files with the size smaller than 4KB which share the same HTML headers and elements. These HTML headers and elements can be recognized by our method. However,these files will just be viewed as totally different chunks by legacy method. For Versions-Range, BURST can still recognize many similar C source code files even with a bigger block size, while the legacy one usually results in a lower deduplication ratio, Our results indicate that BURST is highly advantageous for large backup systems in which larger percentage of chunks are due to minor changes on exist data repository.

Different maximum reference count parameter (c_max). The larger c_max is, the higher deduplication ratio our method can achieve. Figure 6(a) shows that the deduplication ratio with the maximum value set to "255" is 1.9x higher than that with maximum value set to "1" in the Web-Archive case. This result is because of the fact that more burst-alike blocks will be deduplicated with the same reference block with a higher maximum value.

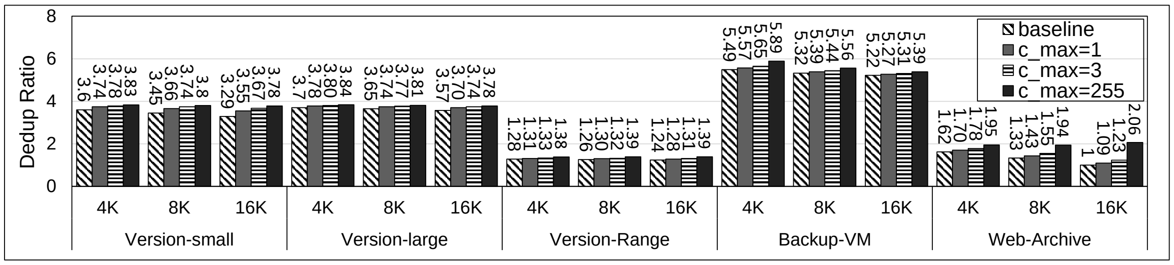
Different block sizes. In legacy deduplication, the dedup-ratio monotonically decreases when the block size increases. However, in the Versions-Range and Web-Archive cases, BURST dedup-ratio keeps flat with the increment of block size. Besides, for Version-Small and Version-Large, BURST dedup-ratio only decreases a bit with the increment of block size. This situation is caused by the fact that burst-alike blocks can still be deduplicated even when the block size becomes larger. Overall, BURST dedup-ratio appears rather insensitive to the block size. This is another key advantage over the legacy approach.

C. I/O throughput

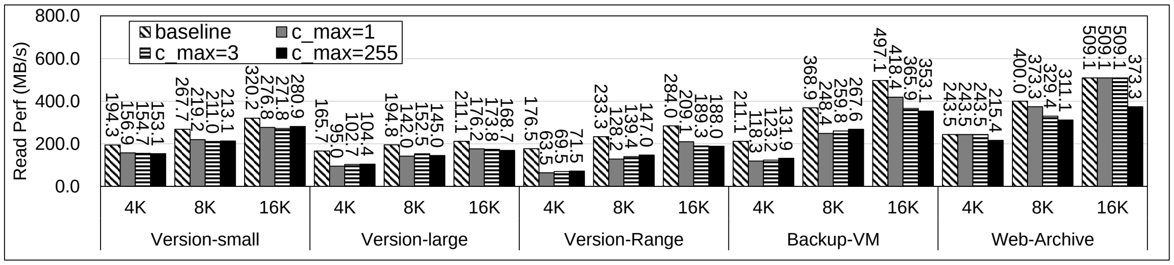
Read throughput. BURST read throughput drops because we introduce more read requests than baseline (OpenZFS). The baseline has no deduplication table reference operations during the read procedures, which show better read performance but cannot perform efficient data deduplication. However, Figure 6(b) shows that the read throughput of our method does not drop significantly in the Version-Small, Version-Large, and Backup-VM cases. In the Version-Small cases where the block size is set as 255, the read throughput of BURST with c_max = 255 is **87%** of read throughput of the baseline. The reason is that there are fewer additional read requests on reference blocks. For Web-Archive dataset, with 16KB block size and c_max set to 255, our method achieves poor read throughput, as there are too many burst in this case so we are more likely to read a burst-alike block which results in two independent read requests. The results from Figure 6(b) also demonstrates that the read performance gap between baseline and BURST becomes smaller as the block size increases. In the Version-Range cases with the block size as 16KB, the read throughput of BURST with c_max = 1 is **73%** of the read throughput of the baseline. This situation is caused by the fact that fewer read I/O requests are issued with larger chunk sizes.

For the dataset such as Backup-VM, it is necessary to make trade-off between read throughput and dedup-ratio by adjusting the blocksizes. However, for the datasets such as Web-Archive and Version-Range, the dedup-ratio is not sensitive to the block sizes, so a large chunk (block) size is preferred.

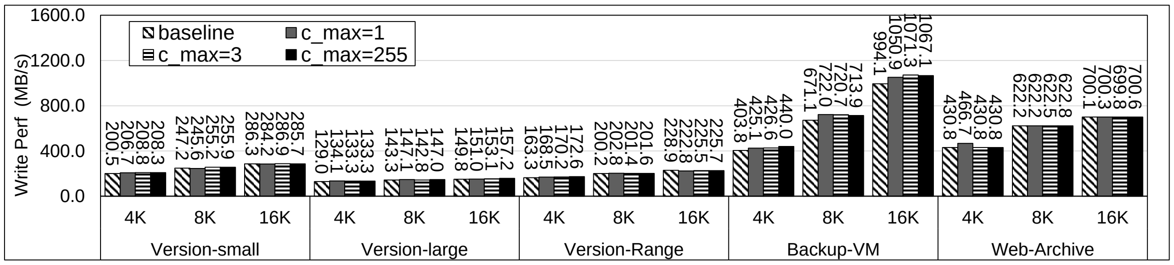
Write throughput. Figure 6(c) demonstrates that the BURST write throughput is almost identical to that of baseline with the same block size. The main reason is that BURST will choose to skip deduplicating a block whose the reference block



(a) Deduplication Ratio



(b) Read Performance

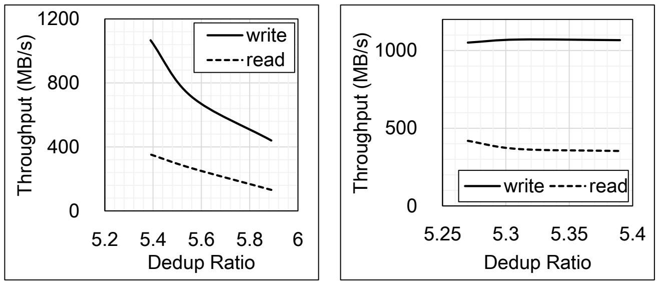


(c) Write Performance

Fig. 6. The deduplication ratio and throughput performance on different real-world data sets.

is not written to storage devices on time. When we increase the value of c_max, BURST deduplication file system cannot get higher throughput, although fewer I/O requests are issued and fewer fingerprints are computed.

D. Relationship between dedup-ratio and I/O throughput



(a) with different block sizes (b) with different c_max

Fig. 7. The relationship between dedup-ratio and throughput.

In previous subsections, we show results of dedup-ratio and I/O throughput respectively. These illustrations can exploit the relationship between deduplication and I/O throughput. In this subsection, We focus on Backup-VM to exploit the relationship between dedup-ratio and I/O throughput and discuss a trade-off between them in Figure 7.

Figure 7(a) shows write and read throughput and dedup-ratio with c_max fixed to 255 and block size set to 4KB, 8KB and 16KB. We find that as block size increases, the dedup-ratio of BURST decreases mildly while the baseline drops sharply; on the other hand, both write and read throughput meaningfully elevates. Figure 7(b) shows write and read throughput and dedup-ratio with block size fixed to 16KB and c_max set to 1, 3 and 255. As c_max increases, the dedup-ratio also increases but the write throughput keeps flat and the read throughput decrease marginally.

E. CPU and Memory Overhead

Three BURST variants are devised to further demonstrate the trade-off between dedup-ratio and CPU/memory overhead, including:

1) **BURST-Head**, a variant of BURST system that only generates the head for each unique block.
2) **BURST-Raw**, a variant of BURST system that sets size of head/tail as 32B (256-bit) and directly uses the raw 32B data as their fingerprints.
3) **BURST-Head-Raw**, a variant of BURST system that only generates the head with 32B size for each unique block.

These variants of BURST use different implementation details about the head/tail configuration. We use the dataset Backup-VM to test the overhead as a example, and we set c_max as 255 and block size to 4KB or 16KB.

Table II reflects the evaluation results of baseline, BURST and its three variants. This table record the memory resource

TABLE II
RESULTS OF DEDUP-RATIO AND MEMORY & CPU OVERHEAD FOR BURST AND ITS 3 VARIANTS.

	Blocksize	Dedup Ratio	**DT** (size)	**XDT** (size)	**BT** (size)	Lock (utilization)	Checksum (utilization)
Baseline	4k	5.49	196MB	/	/	8.3%	6.2%
	16k	5.22	53MB	/	/	2.5%	1.6%
BURST	4k	5.89	168MB	40.2MB	7.6MB	8.6%	23.1%
	16k	5.39	51MB	3.3MB	0.4MB	2.8%	7.2%
BURST-Head	4k	5.56	182MB	20.4MB	6.1MB	8.6%	18.7%
	16k	5.29	52MB	1.7MB	0.3MB	2.7%	5.1%
BURST-Raw	4k	5.89	167MB	40.1MB	7.4MB	8.5%	6.5%
	16k	5.37	51MB	3.2MB	0.4MB	2.7%	2.8%
BURST-Head-Raw	4k	5.61	175MB	20.2MB	6.4MB	8.5%	6.7%
	16k	5.31	52MB	1.8MB	0.3MB	2.8%	2.8%

utilization of **DT**, **HeadDT**, **TailDT** and **BT** respectively. Columns of "Lock" and "Checksum" record normalized CPU overhead of Locking and checksum computation, which are two main types of functions occupying CPU resource. Specifically, 100 % CPU utilization refers to that one CPU core resource is fully occupied.

Firstly, we analyse the BURST file system and their variants with different block sizes. We note that when increasing block size form 4KB to 16 KB, the average CPU utilization is lower while the memory cost with 16KB is about a quarter of that with 4KB. Therefore, to lower memory and CPU overhead, larger block size is a better choice although it results in low deduplication ratio performance.

Then we analyse how to get better balance between higher dedup-ratio and lower Memory overhead. In general, BURST file system and its variants cost more memory for deduplication metadata than baseline. This is because additional *head-dte*, *tail-dte* and *bte* are allocated to record metadata for BURST deduplication. The memory cost is acceptable because memory cost for deduplication metadata (215.8MB, with BURST and 4KB block size) is **2.5%** of size of corresponding data (8.1GB). We note that BURST-head and BURST-head-raw do save memory compared to BURST system although they delete tail deduplication table. This is because it fails detect burst-alike blocks if a burst happens to cross the head chunk.

We also analyse the trade-off between dedup-ratio and CPU/memory overhead. In general, our method costs the same number of CPU cores on locking as baseline. This is because we do not introduce additional locks or locking functions but reuse the existing lock related to *dte* in baseline to lock all the four deduplication tables. BURST-raw eliminates CPU overhead on checksum computation for directly using data as checksums, at the same time retaining data reduction. Burst-head lowers CPU overhead to some extent by computing one fewer checksum for each block and halves memory overhead, however, its dedup-ratio drops noticeably. BURST-head-raw achieves as low CPU overhead as BURST-Raw and also halves memory overhead. Its resulting dedup-ratio sits between BURST-Head and BURST-RAW.

F. Experimental Analysis on Dedup-ratios

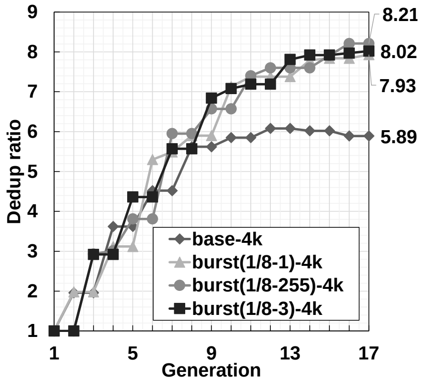
During the incremental evaluations on 5 real world datasets, we note that dedup-ratio saturates along backing up more files. To further manifest this phenomenon, we design an artificial dataset called **Backup-Artificial**, which contains 17 versions of a 128MB file generated artificially. Its first version is randomly filled with ASCII characters, subsequently each new version is built on the preceding version through the following steps: 1) Split the preceding version into 32768 blocks with 4KB size; 2) Randomly choose 1/4 (8192 blocks) from all the blocks; 3) Replace 1KB continuous data with arbitrary offset in each chosen block by random ASCII characters.

We record dedup-ratio along backing up each new version. The results are shown in Figure 8. In this figure, x axis refers to the number of files deduplicated so far. With legacy method, when each newer version is incrementally backed up, the dedup-ratio first increases but then decreases a bit in the end. The intuitive reason is that each newer version steadily introduces more distinct contents from the previous versions, so the dedup-ratio deteriorates after the initial increasing. However, with our method, the dedup-ratio keep moving much higher until becoming saturated, especially when the block size is configured as 4KB or 8KB. This is because we can recognize burst-alike blocks from early versions so an entire data chunk write can be replaced by a small burst write. Upon the above observations, we claim that our method is particularly beneficial in certain real scenarios such as periodical logs backup and VM snapshots.

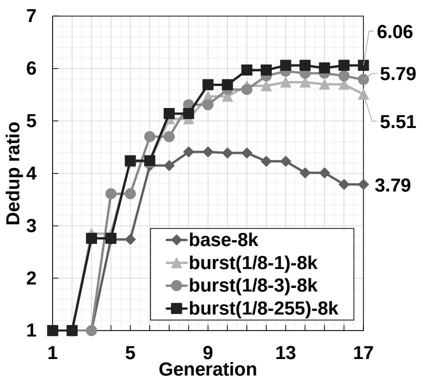
VI. LIMITATION OF BURST AND DISCUSSION

Limitation: The current BURST concentrates on the mainstream file systems using fixed-sized chunks, so integrating the novel design of burst-encoded fingerprint matching methods into the variable-sized chunk-based systems remains as an important future work. Fortunately, BURST can be easily improved to provide different configuration options (such as setting partitioning methods, fingerprint calculation methods, etc.) to adapt to different settings or more comlicated application scenario (such as prioritizing high deduplication rates or low system overhead).

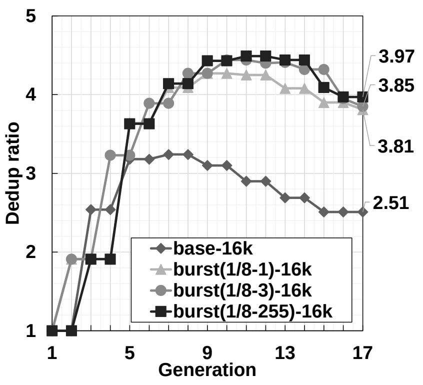
Overhead Discussion: In current BURST implementation, the pattern calculation method of BURST file system uses traditional CPU to calculate checksums, resulting in some CPU usage. Currently, hardware-accelerated calculating checksums researches and methods [32] can be integrated into the BURST prototype to avoid errors and reduce the additional CPU overhead.



(a) backup with block-size=4k



(b) backup with block-size=8k



(c) backup with block-size=16k

Fig. 8. The dedup-ratio in the Backup-Artificial test case.

VII. Related Works

Researches on deduplication file systems [25], [33]–[35] have been concentrating on providing high deduplication ratio and low overhead towards generic data deduplication scenarios against the previous hardware-implemented data deduplication methods on specific storage medias [15], [36], [37], [37], [38]. Fixed-length deduplication has been mainly deployed in primary storage systems [3], [4], [9]. Content-based variable-length deduplication achieves higher data reduction by meticulously handling the boundary shift issue, however, at the price of much longer latency than the block deduplication (e.g., [5], [6], [8], [17]). It is advantageous in the secondary storage systems, wherein latency is not much a concern to allow for computationally intensive segmenting operations. Fingerprint is used to find identical chunks from the existing data storage. There are two types of fingerprints, namely, weak and strong. When strong fingerprint (e.g., SHA-1, SHA-2) is applied, two blocks with identical fingerprints are safeguarded to be identical (noting the probability of data collision is way way below storage failure tolerance) (e.g., [3], [8], [39]–[41]. On the other hand, weak fingerprint (e.g., CRC, or ECC parity) is much shorter and computationally much lighter than the counterpart strong ones (e.g., [16], [27]). when weak fingerprint is applied, the probability of two different blocks with identical fingerprint can no longer be neglected. Therefore, the reference chunk must be read out and matched byte-wise if two weak fingerprints are matched. Evidently, the weak fingerprint approach uses much smaller memory footprint and ensures collision-free at the expense of extra read and match operations. So our BURST design help to provide system prototype implementation insights to reach better balances between deduplication efficiency and the resource overhead.

In delta compression [42]–[44], the key research issue is how to accurately detect a fairly similar candidate for delta compression with low overheads. Manber [20] proposes a basic approach to find similar files in a large collection of files by computing a set of polynomial-based fingerprints (i.e., [19]); the similarity between two files is proportional to the fraction of fingerprints common between them. This approach has been used in [23] to detect similar files and then delta encode them. The super-feature approaches [45], [46] are based on Broder's theorem [47]. Stream-informed delta compression [12] shows that post-deduplication delta compression can further improve the data reduction ratio by a factor of 3 5 when replicating between EMC's deduplicated backup storage systems. Deduplication-aware resemblance detection [24] extends this work of [12] by detecting potential similar chunks for delta compression based on the existing duplicate-adjacent information after deduplication, i.e., considering two chunks similar if their respective adjacent chunks are determined as duplicate in a deduplication system.

VIII. Conclusions

Inspired by an insightful observation that in backup systems a unique data chunk may best match a chunk depository by a short burst, we presented BURST, a novel chunk-based data deduplication System with burst-encoded fingerprint matching to efficiently deduplicate the imperfectly-matched data chunks. BURST creates head and tail fingerprints, associated with the beginning and ending data respectively, on top of the conventional fingerprint. We implemented and evaluated BURST in the open-source storage platform (OpenZFS) and our system prototype is open-sourced. BURST consistently beats the legacy methods on the data reduction across all 5 datasets, wherein the larger dataset produces the more striking gap. BURST can achieve 2.06 × optimization over the legacy method. BURST exhibits almost identical write throughput compared to legacy method while the read performance is 87% of legacy method. Furthermore, BURST with 16KB chunk size is shown to consistently achieve higher data reduction than the legacy method with (smaller) 4KB chunk size, rendering higher performance and lower system cost.

Acknowledgment

We sincerely thank our shepherd, Ramprasad Chinthekindi, and the anonymous reviewers for their suggestions to improve the paper. This work was funded by the National Key Research & Development Program of China (No. 2022YFB4502002), and NSFC (No. 62032008), the HighTech

Support Program from STCSM (No.22511106200), STCSM (No. 23511100100). The corresponding author is Jianguo Yao.

References

[1] S. R. Department, "https://www.statista.com/statistics/871513/worldwide-data-created/", 2021.

[2] B. Debnath, S. Sengupta, and J. Li, "ChunkStash: Speeding up inline storage deduplication using flash memory," in *Proceedings of 2010 USENIX Annual Technical Conference (USENIX ATC 10)*, 2010.

[3] D. E. Technologies, "Introduction to the emc xtremio storage array," "https://www.emc.com/collateral/white-papers/h11752-intro-to-XtremIO-array-wp.pdf", 2015.

[4] P. S. Corp., "https://www.purestorage.com/products/purity/flash-reduce.html", 2022.

[5] A. El-Shimi, R. Kalach, A. Kumar, A. Ottean, J. Li, and S. Sengupta, "Primary data {Deduplication—Large} scale study and system design," in *Proceedings of 2012 USENIX Annual Technical Conference (USENIX ATC 12)*, 2012, pp. 285–296.

[6] G. Wallace, F. Douglis, H. Qian, P. Shilane, S. Smaldone, M. Chamness, and W. Hsu, "Characteristics of backup workloads in production systems." in *Proceedings of USENIX FAST*, vol. 12, 2012, pp. 4–4.

[7] S. Quinlan and S. Dorward, "Venti: A new approach to archival data storage," in *Proceedings of Conference on File and Storage Technologies (FAST 02)*, 2002.

[8] B. Zhu, K. Li, and R. H. Patterson, "Avoiding the disk bottleneck in the data domain deduplication file system." in *Proceedings of Fast*, vol. 8, 2008, pp. 269–282.

[9] L. DuBois and M. Amaldas, "Key considerations as deduplication evolves into primary storage," "http://www.netapp.com/us/media/wp-key-considerations-deduplication-evolves-into-primary-storage.pdf".

[10] N. Zhao, H. Albahar, S. Abraham, K. Chen, V. Tarasov, D. Skourtis, L. Rupprecht, A. Anwar, and A. R. Butt, "Duphunter: Flexible high-performance deduplication for docker registries," in *2020 USENIX Annual Technical Conference (USENIX ATC 20)*. USENIX Association, Jul. 2020, pp. 769–783.

[11] A. Muthitacharoen, B. Chen, and D. Mazieres, "A low-bandwidth network file system," in *Proceedings of the eighteenth ACM symposium on Operating systems principles*, 2001, pp. 174–187.

[12] P. Shilane, M. Huang, G. Wallace, and W. Hsu, "Wan-optimized replication of backup datasets using stream-informed delta compression," *Proceedings of ACM Transactions on Storage (ToS)*, vol. 8, no. 4, pp. 1–26, 2012.

[13] Y. Hua, X. Liu, and D. Feng, "Cost-efficient remote backup services for enterprise clouds," *IEEE Transactions on Industrial Informatics*, vol. 12, no. 5, pp. 1650–1657, 2016.

[14] S. Luo, G. Zhang, C. Wu, S. U. Khan, and K. Li, "Boafft: Distributed deduplication for big data storage in the cloud," *IEEE Transactions on Cloud Computing*, vol. 8, no. 4, pp. 1199–1211, 2020.

[15] F. Chen, T. Luo, and X. Zhang, "$caftl : Acontent - aware$ flash translation layer enhancing the lifespan of flash memory based solid state drives," in *Proceedings of 9th USENIX Conference on File and Storage Technologies (FAST 11)*, 2011.

[16] Q. Hu, J. Chen, D. Feng, Q. Yang, and B. Liu, "An efficient design and implementation of deduplication on open-channel ssds," in *Proceedings of 2019 IEEE 21st International Conference on High Performance Computing and Communications; IEEE 17th International Conference on Smart City; IEEE 5th International Conference on Data Science and Systems (HPCC/SmartCity/DSS)*. IEEE, 2019, pp. 562–569.

[17] Q. Corp., "Data deduplication background: A technical white paper," "https://iq.quantum.com/exLink.asp?6668121OP41V56I29836713".

[18] N. I. of Standards and Technology, "Fips 180-4. secure hash standard," "http://nvlpubs.nist.gov/nistpubs/FIPS/NIST. FIPS.180-4.pdf", 2015.

[19] M. O. Rabin, "Fingerprinting by random polynomials," *Technical report*, 1981.

[20] U. Manber *et al.*, "Finding similar files in a large file system." in *Proceedings of Usenix winter*, vol. 94, 1994, pp. 1–10.

[21] K. Eshghi and H. K. Tang, "A framework for analyzing and improving content-based chunking algorithms," *Proceedings of Hewlett-Packard Labs Technical Report TR*, vol. 30, no. 2005, 2005.

[22] N. Bjørner, A. Blass, and Y. Gurevich, "Content-dependent chunking for differential compression, the local maximum approach," *Journal of Computer and System Sciences*, vol. 76, no. 3-4, pp. 154–203, 2010.

[23] F. Douglis and A. Iyengar, "Application-specific delta-encoding via resemblance detection." in *Proceedings of USENIX annual technical conference, general track*. San Antonio, TX, USA, 2003, pp. 113–126.

[24] W. Xia, H. Jiang, D. Feng, and L. Tian, "Dare: A deduplication-aware resemblance detection and elimination scheme for data reduction with low overheads," *IEEE Transactions on Computers*, vol. 65, no. 6, pp. 1692–1705, 2015.

[25] O. Project, "https://openzfs.org/wiki/Main_Page", 2021.

[26] H. W. Copier, "http://www.httrack.com/", 2017.

[27] S. Wu, J. Zhou, W. Zhu, H. Jiang, Z. Huang, Z. Shen, and B. Mao, "Ead: A collision-free and high performance deduplication scheme for flash storage systems," in *2020 IEEE 38th International Conference on Computer Design (ICCD)*. IEEE, 2020, pp. 155–162.

[28] T. G. P. Debugger, "https://www.gnu.org/software/gdb/", 2022.

[29] T. G. C. Collection, "https://gcc.gnu.org/", 2022.

[30] T. L. K. Archives, "https://www.kernel.org/", 2022.

[31] QEMU, "https://www.qemu.org/", 2022.

[32] X. Hu, C. Wei, J. Li, B. Will, P. Yu, L. Gong, and H. Guan, "Qtls: High-performance tls asynchronous offload framework with intel® quickassist technology," in *Proceedings of the 24th Symposium on Principles and Practice of Parallel Programming*, 2019, pp. 158–172.

[33] L. Kernel, "Btrfs."

[34] V. Rabotka and M. Mannan, "An evaluation of recent secure deduplication proposals," *Journal of Information Security and Applications*, vol. 27-28, pp. 3–18, 2016, special Issues on Security and Privacy in Cloud Computing. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S2214212615000393

[35] W. Bolosky, S. Corbin, D. Goebel, and J. Douceur, "Single instance storage in windows 2000," 08 2000.

[36] J. Kim, C. Lee, S. Lee, I. Son, J. Choi, S. Yoon, H.-u. Lee, S. Kang, Y. Won, and J. Cha, "Deduplication in ssds: Model and quantitative analysis," in *2012 IEEE 28th Symposium on Mass Storage Systems and Technologies (MSST)*. IEEE, 2012, pp. 1–12.

[37] Y. Zhou, Q. Wu, F. Wu, H. Jiang, J. Zhou, and C. Xie, "Remap-ssd: Safely and efficiently exploiting SSD address remapping to eliminate duplicate writes," in *19th USENIX Conference on File and Storage Technologies (FAST 21)*. USENIX Association, Feb. 2021, pp. 187–202.

[38] Q. Hu, J. Chen, D. Feng, Q. Yang, and B. Liu, "An efficient design and implementation of deduplication on open-channel ssds," in *2019 IEEE 21st International Conference on High Performance Computing and Communications; IEEE 17th International Conference on Smart City; IEEE 5th International Conference on Data Science and Systems (HPCC/SmartCity/DSS)*. IEEE, 2019, pp. 562–569.

[39] Y. Zhang, W. Xia, D. Feng, H. Jiang, Y. Hua, and Q. Wang, "Finesse: Fine-grained feature locality based fast resemblance detection for post-deduplication delta compression," in *Proceedings of 17th USENIX Conference on File and Storage Technologies (FAST 19)*. Boston, MA: USENIX Association, Feb. 2019, pp. 121–128.

[40] W. Xia, Y. Zhou, H. Jiang, D. Feng, Y. Hua, Y. Hu, Q. Liu, and Y. Zhang, "Fastcdc: a fast and efficient content-defined chunking approach for data deduplication," in *2016 {USENIX} Annual Technical Conference ({USENIX}{ATC} 16)*, 2016, pp. 101–114.

[41] W. Xia, H. Jiang, D. Feng, L. Tian, M. Fu, and Z. Wang, "P-dedupe: Exploiting parallelism in data deduplication system," in *2012 IEEE Seventh International Conference on Networking, Architecture, and Storage*. IEEE, 2012, pp. 338–347.

[42] "xdelta," 2016. [Online]. Available: http://xdelta.org/

[43] W. Xia, H. Jiang, D. Feng, L. Tian, M. Fu, and Y. Zhou, "Ddelta: A deduplication-inspired fast delta compression approach," *Performance Evaluation*, vol. 79, pp. 258–272, 2014, special Issue: Performance 2014. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0166531614000790

[44] D. Trendafilov, N. Memon, and T. Suel, "zdelta: An efficient delta compression tool," 2002.

[45] A. Z. Broder, "Identifying and filtering near-duplicate documents," in *Proceedings of Annual symposium on combinatorial pattern matching*. Springer, 2000, pp. 1–10.

[46] P. Kulkarni, F. Douglis, J. D. LaVoie, and J. M. Tracey, "Redundancy elimination within large collections of files." in *Proceedings of USENIX Annual Technical Conference, General Track*, 2004, pp. 59–72.

[47] A. Z. Broder, "On the resemblance and containment of documents," in *Proceedings. Compression and Complexity of SEQUENCES 1997 (Cat. No. 97TB100171)*. IEEE, 1997, pp. 21–29.