DBLK: Deduplication for Primary Block Storage

Yoshihiro Tsuchiya, Takashi Watanabe
Fujitsu Limited
Kanagawa, Japan
{tsuchiya.yoshi,wtnb}@jp.fujitsu.com

Abstract— The Deduplication BLocK device (DBLK)[1] is a deduplication and compression system with the block device interface. It is used as a primary storage and its block-wise deduplication is done in inline. Since deduplication for primary storage requires low latency and choosing block-wise deduplication creates large amount of metadata, it is necessary to efficiently use the memory of the system. We solved this problem and developed the Multilayer Bloom Filter (MBF) to reduce the size of the data structure in memory to index duplicate data.

I. DBLK design

The DBLK is a block-level storage system working as a backing store of an iSCSI target (Slide 2). Slide 4 shows the major data structures in the DBLK.

The DBLK splits data into a fixed size (4 KB), computes *SHA-1* hash value of the data, deduplicates and compresses data, and writes them sequentially as a *data log* on disk drives. Metadata are also stored in data log. Logs are kept in *chunks*, which is the fixed size unit in disks.

The DBLK must maintain the mapping between hash values to *physical block addresses (PBAs)*. The DBLK appends hash information, including a new hash value and the PBA of the data, to the *hash log*. Hash information is written sequentially in the hash log.

The DBLK maintains the mapping between a *logical block number (LBA)* and an SHA-1 hash value in a *block map*.

The DBLK uses a Bloom filter to check if incoming data are duplicates of existing data.

Since the size of an entire hash log, which contains the mapping of hash values and PBAs, will be bigger than the physical memory of a server, the DBLK requires a *hash index* to find locations of hash information in the hash log from hash values.

The size of the hash index was our problem. If the DBLK manages an 8 TB RAID, for example, there are 2 billion blocks (=8TB/4KB) which contains deduplicated data. Each data log has 40-byte hash log (160-bit SHA-1, PBA etc.) in the hash log. The size of the hash log will be 80 GB. If a hash index, the mapping between the 160-bit SHA-1 value to the address of hash log, is implemented by regular hash table or B-tree, its size will be more than 56 GB (=2 billion x (160-bit+PBA)), which will not fit in memory of the DBLK server. What is worse, since SHA-1 value does not have locality, if a hash index is stored in a hard drive, a hash index will be accessed randomly and cache will not work well.

II. MBF and MBF-BT

To solve this problem, we extend the Bloom filter from detecting duplicates for finding the hash log block. Under the main Bloom filter, we add two more Bloom filters, each half the size of the first. One corresponds to the left half of the hash log, and the other filter corresponds to the other half. If the first Bloom filter is positive, then we check the two child filters. The area of the hash log can be narrowed down to a single block by repeating this. We call this idea the *Multilayer Bloom filter (MBF,* Slide 5*)*. The lowest layer of the MBF is associated with the blocks of the hash log.

Slide 5 is an example of a *binary* MBF, and MBF can be *N-ary*, which can reduce the 'hight' of the MBF, i.e. less memory consumption.

The *MBF-bitwise transposition (MBF-BT)* optimizes the MBF. Slide 6 shows an example of a layer of MBF with 64 filters. If there are *M* bits in each of the 64 filters, the entire layer can be mapped upon M-64-bit integers by transposition. The MBF-BT lowers the cost of looking up multiple Bloom filters by using integer-wise check rather than bit-wise check.

The DBLK implementation use the MMX instruction set which enables to access 128-bit registers, so the performance improvement is 128x.

By using the MBF-BT, it is possible to reduce the height by increasing *N* of an N-ary MBF. Slide 7 shows the examples of MBFs. For our system, we choose *N*=1664 and the height of the MBF is two, including the top Bloom filter. The top Bloom filter is a collection of 1664 sub-Bloom filters. The size of the MBF is 11 GB. Compared to the hash index with a hash table or B-tree implementation we have discussed previously, which requires more than 56 GB including the Bloom filter, 11 GB is clearly an improvement.

III. Performance

By using the MBF in memory to index metadata on disk drives, the DBLK is able to lower its latency. The performance of the DBLK is shown in the graphs. The DBLK performance is comparable or sometime even better than its base RAID system, because of the log-structured data layout.

[1] This work is supported by the Green IT project of New Energy and Industrial Technology Development Organization (NEDO).

DBLK: Deduplication for Primary Block Storage

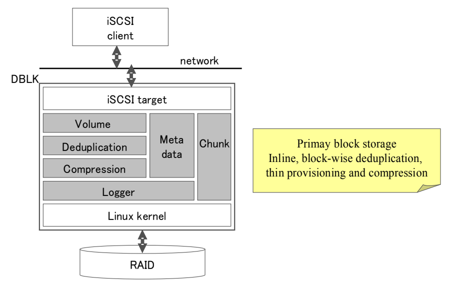
Tsuchiya Yoshihiro, Takashi Watanabe
{tsuchiya.yoshi,wtnb}@jp.fujitsu.com
Fujitsu Ltd.

This work is supported by the Green IT project of New Energy and Industrial Technology Development Organization (NEDO)

Deduplication Background

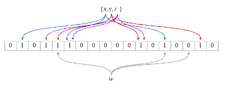
- Dedupliction: Reduce size of storage by removing redundancy in data
- Backup/Primary storage
  - Primary storage
    - Fileserver or datastore of VMware
- Inline vs. Post-process
  - Inline: immediately deduplicate and write data
  - Post process: write data first, deduplicate later, needs extra storage
- Deduplication unit
  - Fixed size block
  - File
  - Variable-sized segment
- Related work & products
  - Mostly for backups, Inline or post-process & variable-sized segment
  - NetApp: primary storage & post-process
  - Exar Bitwakr: primary block storage & inline dedup by hardware accelerator

DBLK iSCSI implementation



Bloom Filter

- The Bloom filter (B. H. Bloom, 1970)
  - Find if an element is in the set
  - Memory efficient
  - False positive
- Example
  - Existing element: x, y, z
  - Three hash functions
  - New data: w

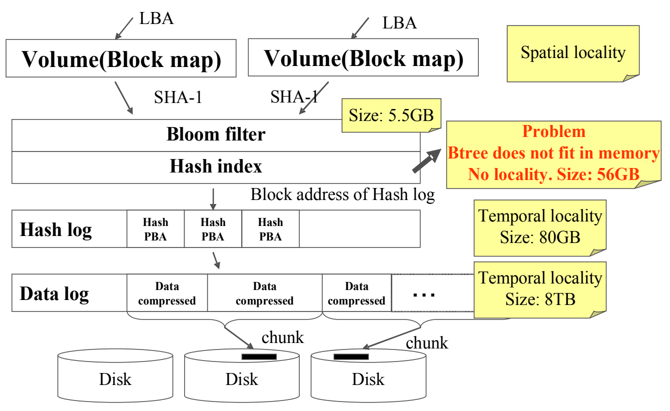


- False positive rate
  - The Bloom filter: m-bit
  - Number of hash functions: k
  - Number of elements: n

$$\left(1-\left(1-\frac{1}{m}\right)^{kn}\right)^{k} \approx \left(1-e^{-kn/m}\right)^{k}$$

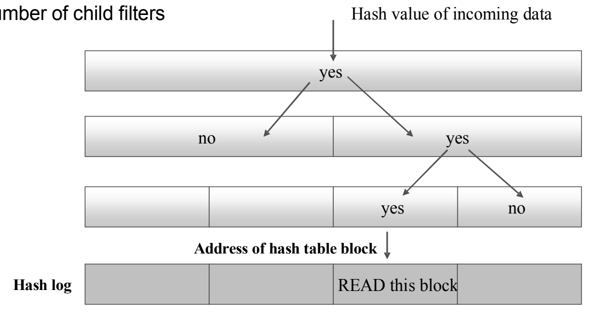
- Useful to detect duplication
  - If a fingerprint is in the current set, it is a duplicate

DBLK Data Structure



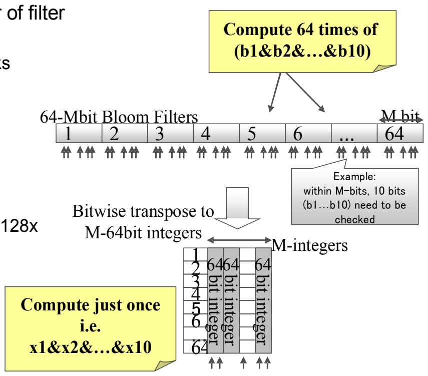
Solution: Multilayer Bloom filter

- Layers of Bloom filters
  - If it reaches to a lowest filter, read the hash log block
- Binary MBF example
  - "Height" is adjustable changing it to "N-ary"
  - N: number of child filters

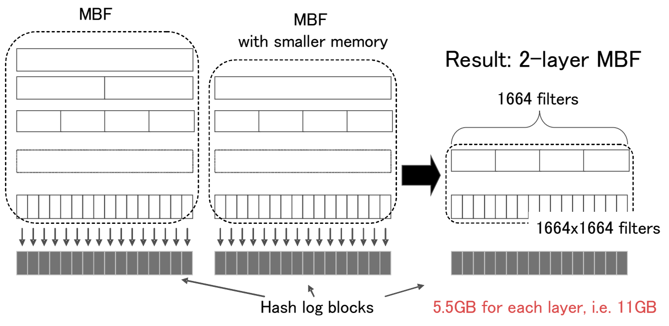


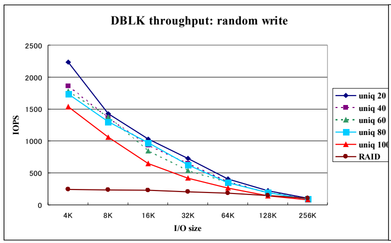
Solution: MBF Optimization: MBF-BT

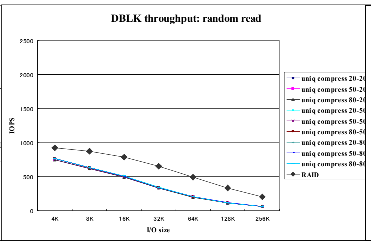
- Key idea
  - Bitmap to integer
  - Use integer operation rather than bit operation
- Logical structure of a layer of filter
  - M-bit bitmap
  - Need check for each blocks
- Physical layout
  - *Bitwise transposition* (*BT*)
  - Maps M-bit on M-integers
  - If 64-bit integer is used speedup: 64x
  - MMX(128bit) is used in the implementation, i.e. 128x
- N-ary MBF with MBF-BT
  - large N can be used
  - We use 1664 for N
  - 2-layers MBF

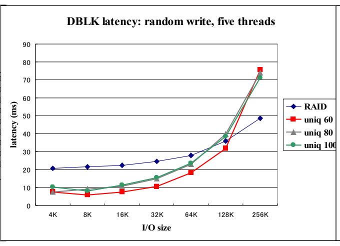


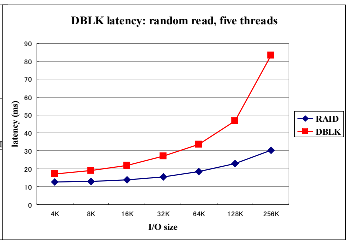
MBF implementation


MBF
MBF
with smaller memory
Result: 2-layer MBF
1664 filters
1664x1664 filters
Hash log blocks
5.5GB for each layer, i.e. 11GB


DBLK throughput: random write
IOPS
I/O size
uniq 20
uniq 40
uniq 60
uniq 80
uniq 100
RAID


DBLK throughput: random read
IOPS
I/O size
uniq compress 20-2
uniq compress 50-2
uniq compress 80-2
uniq compress 20-5
uniq compress 50-5
uniq compress 80-5
uniq compress 20-8
uniq compress 50-8
uniq compress 80-8
RAID


DBLK latency: random write, five threads
latency (ms)
I/O size
RAID
uniq 60
uniq 80
uniq 100


DBLK latency: random read, five threads
latency (ms)
I/O size
RAID
DBLK