DEDUPLICATION IN YAFFS

Karthik Narayan {knarayan@cs.wisc.edu}, Pavithra Seshadri Vijayakrishnan{pavithra@cs.wisc.edu}
Department of Computer Sciences,
University of Wisconsin Madison

ABSTRACT

NAND flash memory has been the single biggest change to drive technology in recent years, with the storage medium showing up in data centers, laptops and in memory cards in mobile devices. It addresses the performance problems in storage. Data Deduplication is simple yet effective technology providing solution to the massive storage requirements of data. Deduplication removes redundant files/blocks of data and ensures that only unique data are stored. We embody the advantages of deduplication and solid-state disks making storage efficient, practical. We have implemented deduplication in YAFFS2, the NAND specific Flash file system on Android OS using chunk index, a compact in memory data structure for identifying new file chunks, which abates data storage limitations. The properties of solid-state devices are harnessed to reduce the complexity of implementation. We implement simple chaining and caching for further optimization. We show that the write time for duplicate data and the storage space has been reduced extensively.

1 INTRODUCTION

Large capacity requirements, increased cost and degraded performance have always been a bottleneck for the storage systems. The introduction of Flash memory for data store has boosted the performance remarkably. In addition to enhanced performance NAND flash based SSD facilitates less power usage, faster data access and reliability and most importantly they use the same interface as hard disk drives, thus easily replacing them in most applications. But solid-state disks are an expensive option and their usage is limited in accordance with the criticality of the application. This cost equation is expected to change in future, so we believe that betterment of file system for solid-state disks will provide the best storage solution.

Data growth has been exponential over the years resulting in increased storage space requirements. Data store studies have revealed that extensive quantity of data in the data store is redundant, especially in the case of backup systems. Deduplication is an intelligent compression technique enabling the removal of redundancy thus improving storage utilization. Deduplication can be in-line deduplication where it is applied during writes in the file system or post process deduplication where it is applied after the data is being written. The granularity of deduplication depends on the magnitude of redundancy. A copy on write copy of the redundant data is created. Deduplication, apart from saving storage space, avoids plenty of disk writes, indirectly enhancing storage performance.

We have implemented in-line deduplication in YAFFS2; the popular commercially used robust file system for NAND using content based fingerprinting. Our testing environment is an android

emulator, which runs the virtual CPU called Goldfish. A compact in memory data structure called chunk index is utilized to store the fingerprint value of file chunks. It is proved that the write time is reduced by a factor of X. We implement chained hashing and caching for further optimization. Hash functions like SHA-1 add computational overhead to deduplication declining its performance hence weak hash functions have been used. Hash collisions are handled by reading the entire data from device and comparing, which would have been a costly operation in HDD.

The remainder of this paper is organized as follows; Section 2 discusses related work; Section 3 explores our design, which includes problems in storage system, understanding deduplication, Overview of YAFFS, Chunk fingerprint, Chunk index cache. Section 4 and 5 explain our implementation and experimental results. Section 6 and 7 are conclusion and references respectively.

2 RELATED WORK

The earliest deduplication system used hashes to address files. This single instance storage standard was file level deduplication. This system identified redundancy only at a file level and did not prove to be very efficient but it laid down the path for deduplication era in storage systems. Over the past three years deduplication market has grown extensively.

Venti is a disk-based write once data storage system [1]. It proposed the approach of identifying a block by the Sha1 hash of its contents, which is well suited to archival storage. The write once model and the ability to coalesce duplicate copies of a block makes Venti a useful building block for a number of interesting storage applications. Dedup yaffs2 uses the idea of hashing for file chunks.

Z. Benjamin et al. proposed Summary Vector and Locality Preserved Caching to avoiding the disk bottleneck in the Data Domain Deduplication File System [2]. This system avoids full chunk indexing. The techniques employed by them reduce the number of disk I/Os in high throughput deduplication storage systems. But because of this assumption, this system is not fully deduplication. PRUNE reduces the disk access overhead of fingerprint management [3]. This was brought about using filter mechanism with main memory index lookup structure and workload-aware index partitioning of the index file in the storage. SSD in the form of NAND flash offers extremely high performance, declining the bottlenecks involved in disk access. Dedup yaffs2 assimilates the segment index methodology in [2]. Hence relatively large number of disk accesses is tolerated.

Seo et al. proposed a deduplication file system for non-linear editing [7]. The file system makes it possible to reduce write operations for redundant data by predicting duplication caused by NLE operations considering causality between I/O operations and thus use NAND flash memory space efficiently. As a result, garbage collection overhead can be reduced greatly. However dedup yaffs2 addresses a more general workload rather than just non-linear editing.

FBBM (Fingerprint based backup Method) performs data de-duplication in the backup [8]. It breaks files into variable sized chunks using anchor-based chunking scheme for the purpose of

duplication detection. Chunks are stored on a write once RAID. FBBM outperforms traditional backup methods in terms of storage and bandwidth saving. This post process deduplication can be combined with our inline technique to maximize the efficacy of storage systems.

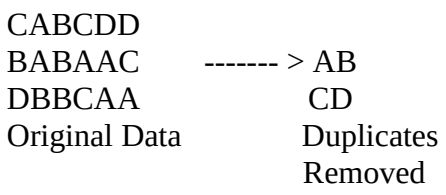
3 DESIGN

3.1 PROBLEMS IN STORAGE SYSTEMS

There is a huge growth of digital data worldwide posing innumerable number of challenges to the storage system. If unneeded redundant data are discarded from storage then large amount of storage capacity can be reclaimed. Deduplication is a simple and straightforward method to dislodge the redundant data from data store. We have designed a deduplication system taking advantage of the properties of NAND flash that reduces the write count and saves as much storage space as possible while not incurring high overheads on the memory, CPU and the device.

3.2 DEDUPLICATION

Deduplication is an intelligent compression technique that eliminates redundant data from data store. Only single unique instance of data is retained and the metadata of the remaining instances are made to point to the device location of the first instance. Deduplication can occur when the data is being written into the device at the file system or after it has been written into the device. The former is referred to as in-line and the latter as post-process deduplication.



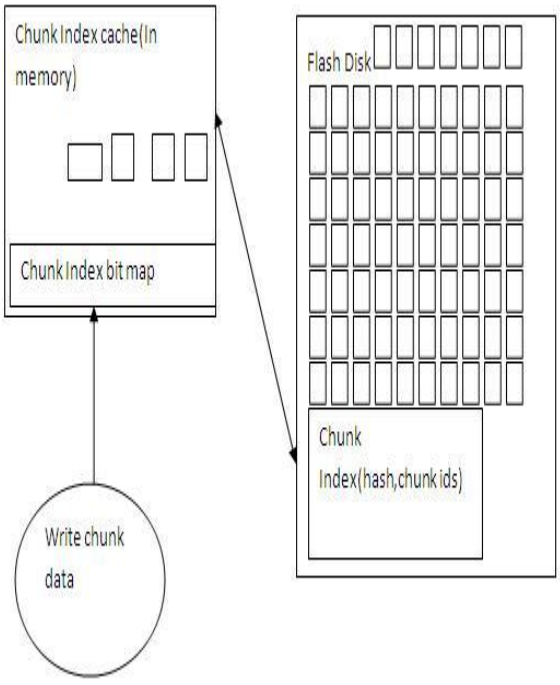
In-line deduplication is more beneficial than post-process; hence we have employed it, reducing the disk I/O count and raising the system throughput. Deduplication can be implemented at file or block level. File level deduplication eliminates duplicate files but this is not very efficient. Block deduplication looks within a file and saves unique iterations of each block. We have incorporated in-line block level deduplication into YAFFS2, a NAND flash file system.

3.3 OVERVIEW OF YAFFS STRUCTURE

YAFFS is a log-structured file system designed for NAND-based flash devices[9]. In the case of flash devices a modified block cannot be written to the original location but it has to be written out to a newly allocated block. YAFFS2 is designed based on this property thus making it and the other flash based file systems predominantly log structured. We study the internals of YAFFS in detail to aid in the incorporation of deduplication into its structure.

In YAFFS, all that is stored in file system are called objects. These are regular data files, directories, hard-links, symbolic links and special objects such as pipes. An object ID references all of these. The objects are divided into chunks. Each chunk has tags associated with it. Object ID and chunk ID uniquely identifies a block within a file.

Exploring YAFFS further reveals the following in-memory structures. YAFFS_DEVICE holds information pertaining to a partition. It contains an array of YAFFS_BLOCKINFO each of which stores information about the file chunks. YAFFS_OBJECT structure exists for every object in the file system. It stores the state of an object. There are different variants of YAFFS_OBJECT to adapt to the different object types. The chunks that belong to an object are stored in a tree structure called YAFFS_TNODES. Also every object will have a pointer to directory structure. The directory structure is maintained as a doubly linked list and relates the sibling objects within a directory. We modify the way in which most of these data structures are updated to incorporate deduplication in YAFFS.



Proposed Architecture

3.4 CHUNK FINGERPRINT

We construct an in memory data structure called a chunk index which is typically a hash table[2] [5]. The chunks of a file are hashed based on its content and the hash value and corresponding chunk ID are stored in the chunk index structure. When a file chunk is to be written into the device, a hash on its content is computed and the value is looked up in the chunk index. If a hash value has a chunk ID entry corresponding to it, the TNODE entry of the corresponding file/object is updated with this chunk ID, so that the logical chunk number is made to point to it. On the contrary, if the look up does not find a chunk ID corresponding to the hash value, we write the chunk data to a new block in the device and the chunk index and the TNODE structure are updated with this new chunk ID value. Also we write this chunk index to the checkpoint region by including this data structure as a part of YAFFS_DEVICE. However there are some serious issues to be addressed:

a) A good choice of hash function- various factors have to be considered before choosing a hash function.

b) Hash collision- as the pigeonhole principle goes; we cannot assume that when a hash value matches the data too would match.

c) Main memory overhead- having all the hash information in memory can lead to a high memory overhead.

We present the following methodologies to tackle these issues.

3.5 CHOICE OF HASH FUNCTION

Data integrity primarily relies on the choice of hash function. If two different pieces of chunk generate the same hash value, then the resulting collision could lead to data corruption as the same chunk ID is pointed to by the object structure of these two chunks [10]. The probability of collision entirely depends on the underlying hash function. The usage of standard cryptographic hash functions such as SHA-1 or SHA-256 reduces the chances of collision greatly. However, there would be an increased computational complexity, which can eat up precious CPU cycles wearing away the performance gain obtained by deduplication. Additional I/Os performed due to hash collision may result in an overhead on hard disk drives. Hence strong hash function would not be a great solution. If we choose a weak hash function, the probability of collision increases.

To overcome these disadvantages we can implement a two level hash function[10]. When there is a hit using a weak hash function hash value is calculated using the strong hash function to verify the duplicity of the data. However, since the cost of read in SSD is very less, we can compare the data directly rather than computing the costly cryptographic hash thereby yielding the CPU for other useful jobs. Therefore, we maintain a simple chain of hashes in memory (array of linked list), where the items in the linked list are chunk IDs corresponding to the hash value. By this we can assure a 100% data integrity without much degradation in performance as we have fully exploited the cheap random read of the Solid State Disks.

3.6 CHUNK INDEX CACHE

Deduplication systems are common in large data centers, where terabytes of data are to be maintained. In such storage systems, the chain of hashes described above would result in a memory overhead proportional to the number of chunks on the flash device. Hence a need arises for a cache of chunk index. A chunk index cache[2] consists of a bit map of hashes to indicate if a hash exists in the cache or not. If it exists the data corresponding to the chunk ID is read and compared with the data to be written out to disk. If there is a match an update of metadata and the cache bitmap are sufficient and no data is written to disk. If the hash value does not exist in the cache then the entry corresponding to the hash value is fetched from the disk and stored in the cache. If the cache has space for the newly arriving chunk then there is no problem else one among the existing chunks will have to be replaced.

A combination of LFU (Least Frequently Used) and LRU (Least Recently Used) replacement policies are determined to be the best to replace a chunk existing in the cache. We have implemented only LFU replacement policy as of now. An extra counter is maintained per hash entry to show the frequency of its usage. So once an entry is to be evicted the least frequently used is removed from the cache and written back to disk providing space for chunk arriving from the disk. Thus every time a hash is accessed its access count is incremented. There is no need to pre-fetch a set of chunk or chunk indexes while working with a flash deduplication system which would have been otherwise necessary in case of a hard disk drive to save seek and rotational time.

Determining the optimal size of the LFU cache was an interesting problem. We tried to incrementally increase the size of cache for a workload of repeated large random writes until the performance of the system matched a system with full-fledged in-memory chunk index. The size of the cache determined by this method were few hundred less than the chunk index size that was present in the earlier implementation.

4 IMPLEMENTATION

We implemented deduplication for YAFFS2 in an android emulator, which runs a virtual CPU called Goldfish. YAFFS_WriteChunkDataToObject is the function in the YAFFS file system code that writes chunk to device. We created an in memory data structure called chunk index and then incrementally developed the system to handle a chunk index cache (LFU only). Before every write operation we perform a lookup operation in the chunk index cache. If there is a chunk match then we prevent a disk write and just update the YAFFS_TNODE structures for the given YAFFS_OBJECT [9]. As discussed in the previous section we avoid CPU overhead by computing weak hashes and we also save main memory by maintaining a chunk index cache.

Read functionality of YAFFS has not been modified. Read operation would just read the chunk using the pointers in the TNODE structure of YAFFS_OBJECT. We add the chunk index cache as a part of YAFFS_DEVICE structure, thus the cache is written at the end of the checkpoint region. The functions YAFFS_ReadCheckPointData and YAFFS_WriteCheckPointData have also been modified to accommodate this. YAFFS provides a custom memory allocation named YMALLOC to reserve a space in heap. We used that to allocate nodes for the chained hash table. However, YMALLOC must be used judiciously. Initially when we developed the system with fully chained hash table all in main memory, the kernel crashed.[3]

The ‘no overwrite’ policy of flash file systems eliminated the need for reference counts on a chunk since new data is written out to a new chunk and hence consistency of YAFFS_TNODE structures is maintained. Also we need not do a pre-fetching operation of chunk indices as in case of hard disks taking advantage of the fast reads on solid state disks thereby reducing the complexity and overhead in implementation.

5 EXPERIMENTAL RESULTS

To evaluate our system, we first issued writes for a small file say a 10KB (10 blocks), the write time decreased by a factor 6.5 after the first file write. Similarly we tested it for a reasonably large 800KB (800 blocks) file. In this case the write time decreased by a factor of 2. We evaluated the performance of YAFFS2 without deduplication and with deduplication and compared the results. It is apparent from the graph that there is a marked difference between them. The absolute value of numbers does not mean anything as we ran the file system on android emulator but it is clear that the file system with deduplication performs better than the file system without deduplication.

We measured the write time of a 10KB file about 15 times. The first time it was written in about 380 µs and the next write

took about 60 µs, a reduction factor of about 6.5 as shown in Figure1.

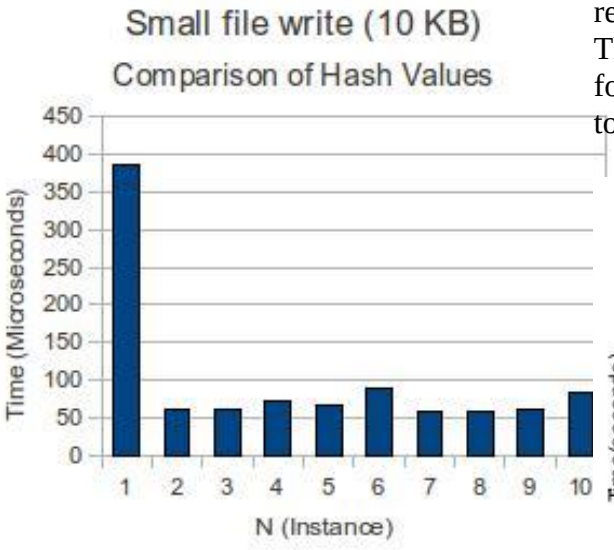


Figure1

Similarly the write time for an 800KB file was measured about 15 times. The first time it was written in about 805 µs and the next write took 395 µs, a reduction of about 2 as shown in Figure2.

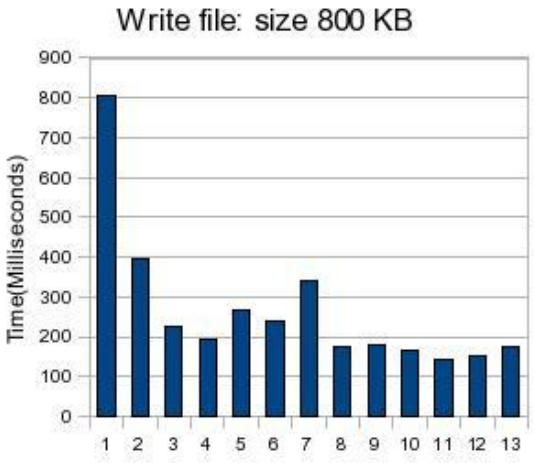


Figure2

We also compared the performance of a YAFFS write without deduplication and with deduplication. As shown in Figure3, Our deduplication system reduces the write time by a considerable amount when compared to YAFFS without deduplication. This measurement was performed for a file with 1000 blocks. Large amount of system storage is reclaimed as data redundancy is avoided. The amount of storage saved is the highest for a backup system and varies according to the workloads.

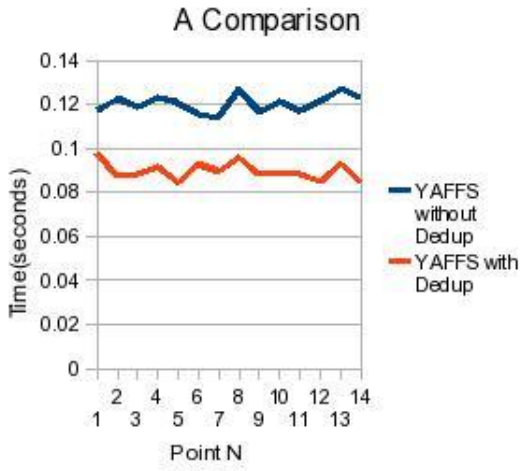


Figure3

6 CONCLUSION

The approach of incorporating deduplication into Solid state disks has proven very effective. The speed of solid state disks dismisses the hindrances involved in direct access of storage system making it the decade's most important data storage technology. Deduplication has brought down the amount of data in storage considerably, reducing the device capacity requirements and henceforth the cost. Deduplication on SSDs would be at the forefront of backup solutions in future. These two technologies together can bring down storage costs without sacrificing performance or reliability. Advancement in deduplication technology and reduction in SSD cost will make these benefits more apparent.

7 REFERENCES

[1] Quinlan S, S Dorward; Venti: a new approach to archival storage In Proceedings of USENIX File And Storage Systems (FAST), 2002

[2] Benjamin Zhu; Kai Li; Hugo Patterson; Avoiding the Disk Bottleneck in the Data Domain Deduplication File System. In Proceedings of USENIX File And Storage Systems (FAST), 2008

[3] Jaegeuk Kim; Heeseung Jo; Hyotaek Shim; Jin-Soo Kim; and Seungryoul Maeng; Efficient Metadata Management for Flash File Systems.In 11th IEEE Symposium on Object Oriented Real-Time Distributed Computing (ISORC)

[4] Youjip Won; Jongmyeong Ban; Jaehong Min; Jungpil Hur; Sangkyu Oh; Jangsun Lee; Efficient index lookup for De-duplication backup system. Modeling, Analysis and Simulation of Computers and Telecommunication Systems, 2008. MASCOTS 2008. IEEE

[5] Thwel, T.T.; Thein, N.L; An Efficient Indexing Mechanism for Data Deduplication . Current Trends in Information Technology (CTIT), 2009. IEEE

[6] Chuanyi Liu; Dapeng Ju; Yu Gu; Youhui Zhang; Dongsheng Wang; Du, D.H.C.;Semantic Data De-duplication for archival storage systems. Computer Systems Architecture Conference, 2008. ACSAC 2008.

[7] Man-Keun Seo; Seung-Ho Lim;Deduplication flash file system with PRAM for non-linear editing. Consumer Electronics, IEEE 2010

[8] Tianming Yang; Dan Feng; Jingning Liu; Yaping Wan;FBBM: A New Backup Method with Data De-duplication Capability.Multimedia and Ubiquitous Engineering, 2008. MUE 2008.

[9] How does Yaffs work http://www.yaffs.net/yaffs-internals

[10]Risk of hash collisions in data deduplication; eXdupe.com December 2010