Block-level Inline Data Deduplication in ext3

Aaron Brown
Kristopher Kosmatka
University of Wisconsin - Madison
Department of Computer Sciences

December 23, 2010

Abstract

Solid State Disk (SSD) media are increasingly being used as primary storage in consumer and mobile devices. This trend is being driven by factors including low power demands, resistance to environmental shocks and vibrations, and by superior random access performance. However, SSDs have some important limitations including high cost, small capacity, and limited erase-write cycle lifespan. Inline data deduplication offers one possible way to ameliorate these problems by avoiding unnecessary writes and enabling more efficient use of space. In this work we propose an inline block-level deduplication layer for ext3 called Dedupfs. To identify potential deduplication opportunities Dedupfs maintains an in-memory cache of block hashes. Block reference counts are monitored for each block in the filesystem in order to prevent freeing a still-referenced block. The new metadata structures in Dedupfs are independent of and complimentary to existing ext3 structures which ensures easy backward compatibility.

1 Introduction

The market for primary storage devices has been dominated over the past several decades by the basic spinning-disk design. The disk platform offers many benefits including very large capacity, low cost per size, and quite good sequential read and write performance. However, the mechanical nature of these devices presents a number of inherent limitations. Importantly, random access is much slower than sequential access. Extensive work has been focussed on overcoming this limitation with approaches ranging from the first disk-aware filesystems like Unix FFS [1], to taking advantage of parallel I/O using RAID systems [2]. Over time, however, improvements to CPU and memory performance have far outstripped that of disk I/O. The hard disk remains a significant and increasing bottleneck in system design.

The development of NAND-flash based solid-state storage devices (SSDs) has the potential to address many of the limitations of disks and to revolutionize the design of storage systems. SSDs offer a range of benefits including high overall read bandwidth and exceptional random access performance in comparison to mechanical disks. Additionally, they possess a number of characteristics that are attractive to modern consumer mobile devices including a low power usage profile, resistance to environmental shocks and vibrations, and silent operation. Until recently the cost of SSDs has limited their adoption to high end specialized applications, but as costs continue to decline their price to benefit ratio will lead to much wider use.

SSDs are not a silver bullet, they carry a range of important limitations of their own [3]. First, and most simply, manufacturing and price constraints assure that for some time the storage capacity of SSDs will remain substantially less than disks. Second, the NAND-flash hardware requires that

data must be erased (zeroed) before new data can be written. Further, the minimum physical unit of erases is much larger than the minimum unit of reads and writes. Read/write pages are generally 2-4 KB whereas erase blocks are 64-128 pages in size. Because of this discrepancy, reads are much faster (on the order of 100-200 μs) than writes (approximately 1.5 ms). Third, NAND-flash has a fixed erase-write cycle lifespan before the medium is no longer usable. After a certain number of writes to a page it becomes unusable and the firmware must mark it as dead.

Together these limitations of the NAND-flash hardware can be thought of as the *painful write problem*. Writes are expensive, we only get a fixed number of them per page, and we have scarce space in which to put them. Thus we should seek to minimize the total number of writes that are performed over the lifespan of the device in order to avoid making pages unusable and to lessen the erase cycle overhead. Further due to capacity constraints, when we must write we should do it in the most space efficient way possible.

Data deduplication is one promising approach to achieve both reduction of writes and efficient space usage. Deduplication is a coarse grained data compression technique that reduces redundancy by eliminating large chunks of identical data. In this work we examine block-level inline deduplication as a means to ameliorate the SSD painful write problem. We present a model implementation of an inline block-level deduplication layer added to the ext3 filesystem that we call Dedupfs. We conclude that the addition of block-level inline deduplication to an existing filesystem is quite feasible, effective, and carries potentially little overhead.

In section 2 we describe some related work on deduplication that influenced our design choices for Dedupfs. In section 3 we present the high level architectural featers in the Dedupfs system. In section 4 we delve a bit deeper into the implementation details in ext3. In section 5 we present some of measures of performance and efficacy using Dedupfs. Finally, in section 6 we close with a discussion of our experiences and conclusions.

2 Related Work

Deduplication techniques have a long history in the enterprise storage market with a wide range of implementation approaches. Existing designs are broadly separable by their level of duplicate granularity (from block-level up to whole file level), when they perform deduplication (inline vs out-of-line), and their method of duplicate detection (hashing vs. bytewise comparison). Inline deduplication refers to a process that identifies duplicate data before it hits disk, whereas out-of-line deduplication first writes data to disk then retroactively deduplicates it in the background. Hashing relies on cryptographic hash digests of data chunks in order to identify duplicate data whereas bytewise comparison explicitly compares the data itself.

The goal of Dedupfs is to preemptively prevent unnecessary writes before they occur. We therefore adopt an inline deduplication approach in which a write is intercepted at the last possible moment and diverted to a deduplication if possible. Inline deduplication requires some kind of index of the current state of the disk in order to identify when duplicates exist. With very large disk capacities this leads to the so-called *chunk-lookup disk bottleneck problem*: the index is much too large to keep in memory and a disk-based index is too slow.

The *chunk-lookup disk bottleneck* has been addressed by Zhu et al. [4] in the Data Domain File System using a Bloom filter and intelligent index cacheing. Together these strategies allow their system to maintain a complete index on the underlying data while avoiding a majority of index-related disk I/Os when performing lookups. While the approach does avoid disk I/O it unfortunately consumes a significant amount of memory resources.

Lillibridge et al. [5] improved on this memory overhead with a sparse-indexing approach that takes advantage of sampling and locality. Data is split into large segments and as it is streamed into the sytem only segments that are spatially local to each other are indexed and deduplicated. Thus over time during operation the in-memory index is kept small. Importantly, this exposes some chance of missing

deduplication opportunities in favor of performance.

An interesting approach to deduplication at the memory subsystem level was offered by Waldspurger in the VMWare ESX server system [6]. ESX server provides a content-based mechanism for memory page sharing among several virtual machines. The designers accomplish this with a best-effort approach based on duplicate hints rather than ground truth. As pages are scanned hashes are entered in a table and when a collision occurs it represents an opportunity for coalescing. For singly referenced pages this hash is not trusted since it may be overwritten at any time, instead it is treated as a hint and rehashed on future matches before coalescing.

In the spirit of ESX Server, Dedupfs takes a best-effort approach to hashing that in our case favors performance over completeness. The hash index in Dedupfs is simply a cache of observed mappings to assist the system in identifying duplicates. The hash cache is consulted to find candidate duplicates but not trusted to be internally consistent. All hits are verified with a bytewise comparison before coalescing. Thus it may miss some deduplication opportunities that are latent in the underlying data in favor of a much smaller index and the option of using faster less strict hash digests.

3 Dedupfs Architecture

The principles of our design are as follows:

- Provide a working filesystem which tries to save space and avoid writes via best effort attempts to find duplicate blocks.
- Minimize memory usage of any added mechanisms in order to avoid competing with the in memory disk cache and reducing performance.
- Maintain backwards read compatibility with the existing file system and make converting between filesystems simple.

In order to find potential duplicates, we intervene at the existing filesystem's write procedure. We compare the data block that is about to be written to data that has been previously written. If the new data block matches one that is already on disk, instead of writing out the new block, we update the file's metadata to point to the existing block on disk.

In order to identify potential duplicate blocks, we maintain an in memory mapping of data hashes to block numbers. This mapping is incomplete in that only some of blocks will appear, and it only serves as a hint, in that a hash listed for a block may be incorrect. Hash to block mappings are added whenever there is a write. In order to minimize memory usage, we limit the amount of mappings that we store and we simply discard old entries to make room for more recent ones. Since the mapping is incomplete, we do not guarantee that we will find all duplicate blocks. Since the mappings are just hints, we do not employ any complex procedures to remove stale mappings. We simply allow stale mappings to remain since they will eventually be replaced with more recent ones. However, we verify mappings when we find a match using a full bytewise comparison of the suspected duplicate block.

In order to maintain correct filesystem behavior, we need to track which blocks are duplicated, that is, which blocks are referred to by multiple block pointers. To do this we add an extra piece of filesystem metadata for each block which simply keeps a count of the number of references to that block. If a block has one reference it is safe to allow overwrites, and if a block has zero references it is safe to free that block. This is the only piece of persistent metadata we add, and it needs to be written out to the disk.

To achieve backward compatibility the block reference count metadata are stored as a regular file in the filesystm. This file simply contains an array of block reference counts. Since our changes only affect writes, the original filesystem and the deduplication filesystem are fully read compatible. To convert from the original filesystem, we just need to create the block reference count file. To do this we would simply scan all the block pointers and mark all in use blocks as having a reference count of one. To convert back to the original filesystem, we scan through the reference count file and copy any

deduplicated blocks the needed number of times and then update the block pointers to each point to a unique block. Such a procedure can be accomplished using the common `fsck` utility.

In the following paragraphs, we list some contrasting approaches to deduplication, and why we consider our approach to be the preferred approach.

3.1 Block level deduplication

Deduplication could be implemented in a block layer device driver, which sits in between the filesystem layer and the actual underlying block device. This approach has the advantage that it is general and can be used for any filesystem. However, extra overhead is necessary to accommodate the extra layer of indirection. Our implementation simply uses the existing block pointers in filesystem metadata, regardless of whether the pointer points to a regular block or a deduplicated block.

3.2 True Hashes instead of hints

A system could identify all possible deduplication opportunities if block hashes were maintained to always be correct. Finding all deduplication opportunities means that more space could potentially be saved. If a strong hash function with a low collision rate is used, such as SHA-1, then it may be acceptable to assume that matches truly refer to the exact same data and thus avoid doing a bytewise comparison. Also, the stored hashes could be used to verify data on reads as well as find duplicate blocks. The approach has the disadvantage that large amounts of space must be used to store block hashes. The data must be persistent on disk in order to avoid re hashing all the blocks on mount. Further, for even moderately sized disks, storing all hash data in memory would consume a large amount of memory and thus greatly reduce the effectiveness of the disk block cache. It is also difficult to cache this sort of information, since hashes are effectively random and any block on the disk could potentially be a match.

3.3 Replace block pointers with hashes

The block pointers in the filesystem could be replaced with hashes of the written data, as is done in some content-addressable systems. This approach has several disadvantages. To avoid collisions, the hash must be considerably larger than a block pointer. Extra machinery is needed to store and retrieve blocks by hash, and to free unneeded blocks. Finally, this approach is not backwards compatible with existing filesystems.

3.4 Offline deduplication

Another approach could include scanning for duplicates while the system is idle. Offline scanning is not exclusive and could be used in addition to on-the-fly Inline deduplication. Offline scanning could find more duplicates since more resources are available when the system is idle. Offline deduplication, however, does not reduce the number of writes during active use, which is of course a primary goal in SSD applications.

4 Implementation

We implemented Dedupfs, a version of the ext3 filesystem for Linux modified to provide on-the-fly inline block-level deduplication. We wrote Dedupfs for the latest stable version of Linux available at the start of the project, Linux version 2.6.36. Our changes cover approximately 900 lines of code. Most of the changes are to the page writeback code, with a few additional changes to block allocation and free logic.

4.1 Hash Cache

The hash cache is a list of data block hash to block number mappings. These mappings are only stored in memory, they are never written to disk. Whenever a data block is written out to the disk, we hash the data in the block and add the mapping to the cache.

The hash function used and the size of the cache are set at mount time as mount options. The Linux cryptography API is used to genereate hashes, so any hash function supported by the API can be used. Reserving a large amount of space for the hash cache allows more mappings to be stored and increases the likelihood of finding a match. However, this also decreases the amount of memory available to the in memory disk cache, which will increase the number of misses to that cache and slow down filesystem operations.

In order to reduce memory usage and to keep the implementation simple, mappings in the hash cache are only treated as hints. We make no attempts to remove stale mappings when blocks are freed or overwritten. As new blocks are written out, their mappings will be added to the cache. As the cache fills up, old mappings will be evicted. We evict using an approximate LRU clock-like algorithm.

The hash cache is stored as a hash table with a small optimization to save some space. The hash of the block is the key and the block number is the stored value. To save space, we spilt the key into two parts. The first part of the key serves as the index into the hash table itself. Since the key is already a hash, simply using the first part of the key as an index will provide sufficient distribution of the keys into buckets. Instead of storing the entire key with each entry in the table, we only store the second part of the key. Since the first part of the key is the index into the table, the entire key can be reconstructed by simple concatenation.

4.2 Reference Counts

We use a regular ext3 file to store a reference count for each block. We store this as one large simple array of reference counts. This data is filesystem metadata that must be persistent and thus written to disk. This reference count data is the only on-disk metadata that we add.

The reference count for a block is incremented whenever a block pointer that points to that block is created. This occurs in during a normal write when blocks are allocated as well as when a block is deduplicated. The reference count is decremented whenever a block pointer is changed to not point to that block. This occurs when a file is truncated or deleted, or when copy-on-write is done on that block.

Ext3 journals metadata in order to maintain filesystem consistency and to provide fast recovery in the event of a crash. We journal the updates to the reference counts in the ext3 journal in order to provide the same assurances. We also make use of existing ext3 routines to change block pointers and to allocate or free blocks, so those metadata changes are journaled as well.

4.3 Write flow

The main difference between Dedupfs and ext3 is the writeback procedure. This procedure is called whenever a dirty page in the in memory disk cache should be written out to the disk. This occurs when the user requests a sync, after a timeout (every few seconds), or when the system is under memory pressure. In ext3 in ordered mode, each dirty block in the page is just written out to its correct location on disk.

The writeback procedure in Dedupfs checks for duplicates before writing out each block (Fig. 1). The data in the block is hashed, and then Dedupfs does a lookup of the calculated hash in the hash cache. If no matching hashes are found, we check the block's reference count to see if it is safe to overwrite the block. If the reference count is one, we add the hash-to-block mapping into the cache and then simply write out the data block. If the reference count is more than one, we decrement the reference count, allocate a new block, and redirect the write to go to the newly allocated block.

If a match is returned, this means that at some point we wrote out a block to disk that hashes to the same value. So we do a bytewise comparison between the writeback block and the existing block. If the existing block is not in the in memory cache, then it must be read from the disk. If the block data does not match, we check the reference count and write out the data as before. The new hash to block mapping will also be added to the hash cache and replace the old incorrect one.

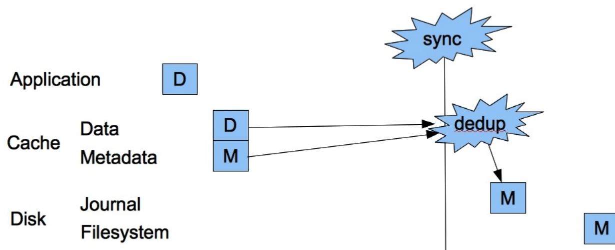

Figure 1: The flow of data during a write in Dedupfs. The write is interrupted at the last possible moment before a block is flushed to disk. At that point the hash cache and reference counts metadata are consulted to determine if a deduplication is possible. If a valid duplicate block is found the flush can be cacelled and the appropriate metadata updated to point to the duplicate block

If the block data matches exactly, then it is safe to do deduplication. In this case we increment the reference counter for the existing block, update the file's block pointer to point to the new block, and then mark the new block as clean without actually writing it to disk.

5 Evaluation

Several synthetic workloads were designed to evaluate the performance of Dedupfs. A minimal Debian distribution was compiled with Dedupfs support and built as a User Mode Linux (UML) executable. These tests were run inside UML on a host machine with dual Pentium 4 3.2 GHz CPUs and 2 GB of RAM. The Dedupfs filesystem image was a unix file on this host system. Each test was run 10 times and total running time was recorded.

In order to observe inherent overhead costs for Dedupfs we tested a synthetic completely duplication free workload. 1000 unique one block files were created, each was forcibly flushed to disk, finally the files were deleted. The total running time for the workload was recorded. The mean time for Dedupfs was 17.89 ± 0.21 s, the mean time for ext3 was 17.75 ± 0.08 s. The results indicate a slight performance hit for Dedupfs although not significant.

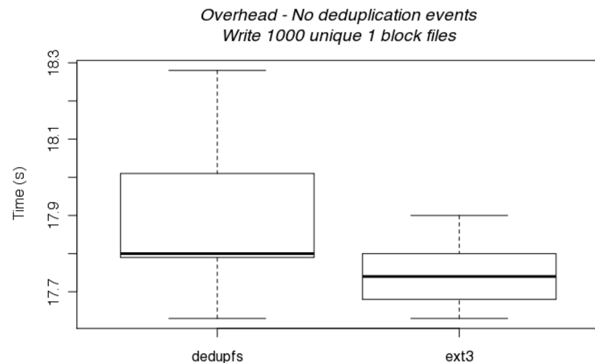

Figure 2: Performance of Dedupfs under a duplicate-free workload. 1000 unique one block files were created, forced to disk, and deleted. The results reveal little or no overhead.

We also sought to observe the performance of an ideal workload with the maximum opportunity for deduplication. 1000 identical one block files were created, forced to disk, and subsequently deleted. Again, the total running time for the workload was recorded. The mean time for Dedupfs was 17.89 ± 0.10. The mean time for ext3 was 17.81 ± 0.10. Once again Dedupfs appears to have a slight but not significat performance hit.

In addition to observing the overhead performance of Dedupfs, we also sought to observe the latent opportunity for deduplication that exists in collections of real data. We performed recursive copies of several directory trees common to many *nix distributions, and recorded the number of deduplication events that Dedupfs was able to recover through these operations. Table 1 shows that Dedupfs was able to detect a significant amount of duplication in several of these data sets. Intrestingly, these results indicate the amount of duplication is not directly related to the total amount of data involved, it is highly dependent on the specific nature of the data involved.

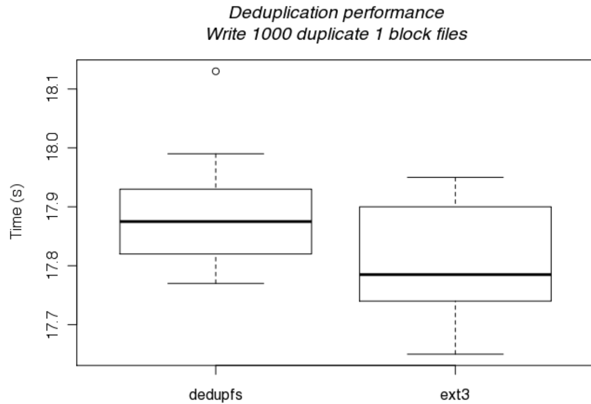

Figure 3: Performance of Dedupfs under a duplication rich workload. 1000 identical one block files were created, forced to disk, and deleted. The results reveal little or no overhead, but also no observable performance benefit.

Directory	Size (MB)	Duplicate Blocks
/bin	3.5	147
/lib	6.4	53
/usr/bin	14.0	0
/usr/lib	29.0	139

Table 1: Duplications revealed by recursive copying of several data collections in Linux. Block size in this filesystem was 1 KB.

6 Conclusion

The results are encouraging and indicate low overhead for Dedupfs while still detecting and deduplicating significant amounts of latent duplicate data. We predicted to see a performance gain in the deduplication rich performance test, but in fact observed approximately equal performance with ext3. This may suggest that there are two factors tugging on performance in deduplication heavy workloads. On one hand we are avoiding disk write I/O, but on the other hand we must perform a hash lookup and a bytewise comparison in order to do it. We view this as acceptable since performance gain is not the primary goal, rather our goal is to avoid writes and achieve space savings.

The best effort hint-based approach to duplicate detection in Dedupfs achieves our goals of good performance and small size. This allows it to be kept entirely in memory without the need to add it to the persistent filesystem metadata. This design, however, does allow the possibility of missing deduplication opportunities that a complete duplicate indexing system would catch. Ultimately the size of the hash cache and its replacement policy will govern how effective it is at detecting duplicates. In this work we did not test the effects of cache replacement policy but we expect it will form an important line of investigation in the future.

We view Dedupfs, and block-level inline deduplication in general, as a viable approach to the painful write problem on SSDs. Our experience in ext3 showed that with a modest addition to an existing widely used filesystem we can achieve effective deduplication on several common workloads with minimal or negligible overhead cost. We view the minimal persistent metadata and backward compatibility of Dedupfs as a major benefit. Many applications that use SSDs as primary storage simply use an existing standard filesystem. Providing a simple and transparent migration path to and from Dedupfs is crucial in order to promote real-world adoption. Just as the original FFS in Unix added disk awareness to an otherwise hardware oblivious filesystem, we see inline deduplication as adding a crucial and needed layer of SSD awareness. As industry transitions away from the spinning disk model and toward solid-state, this kind of approach will become increasingly critical.

References

[1] M. K. McKusick, W. N. Joy, S. J. Leffler, and R. S. Fabry, “A fast file system for UNIX,” *ACM Transactions On Computer Systems*, vol. 2, no. 3, pp. 181–197, 1984.

[2] D. A. Patterson, G. Gibson, and R. H. Katz, “A case for redundant arrays of inexpensive disks (RAID),” in *Proceedings of the ACM SIGMOD International Conference on Management of Data*, 1988.

[3] N. Agrawal, V. Prabhakaran, T. Wobber, J. D. Davis, M. S. Manasse, and R. Panigrahy, “Design tradeoffs for SSD performance,” in *USENIX Annual Technical Conference* (R. Isaacs and Y. Zhou, eds.), pp. 57–70, USENIX Association, 2008.

[4] B. Zhu, K. Li, and R. H. Patterson, “Avoiding the disk bottleneck in the data domain deduplication file system,” in *FAST* (M. Baker and E. Riedel, eds.), pp. 269–282, USENIX, 2008.

[5] M. Lillibridge, K. Eshghi, D. Bhagwat, V. Deolalikar, G. Trezis, and P. Camble, “Sparse indexing: Large scale, inline deduplication using sampling and locality,” in *FAST* (M. I. Seltzer and R. Wheeler, eds.), pp. 111–123, USENIX, 2009.

[6] C. A. Waldspurger, “Memory resource management in VMware ESX server,” in *OSDI*, 2002.