Using Hints to Improve Inline Block-Layer Deduplication

Sonam Mandal,[1] Geoff Kuenning,[3] Dongju Ok,[1] Varun Shastry,[1] Philip Shilane,[4] Sun Zhen,[1,5]
Vasily Tarasov,[2] and Erez Zadok[1]
[1]*Stony Brook University,* [2]*IBM Research—Almaden,* [3]*Harvey Mudd College,*
[4]*EMC Corporation, and* [5]*HPCL, NUDT, China*

Appears in the proceedings of the 14th USENIX Conference on File and Storage Technologies (FAST'16)

Abstract

Block-layer data deduplication allows file systems and applications to reap the benefits of deduplication without requiring per-system or per-application modifications. However, important information about data context (e.g., data vs. metadata writes) is lost at the block layer. Passing such context to the block layer can help improve deduplication performance and reliability. We implemented a hinting interface in an open-source block-layer deduplication system, *dmdedup,* that passes relevant context to the block layer, and evaluated two hints, NODEDUP and PREFETCH. To allow upper storage layers to pass hints based on the available context, we modified the VFS and file system layers to expose a hinting interface to user applications. We show that passing the NODEDUP hint speeds up applications by up to 5.3× on modern machines because the overhead of deduplication is avoided when it is unlikely to be beneficial. We also show that the PREFETCH hint accelerates applications up to 1.8× by caching hashes for data that is likely to be accessed soon.

1 Introduction

The amount of data that organizations store is growing rapidly [3]. Decreases in hard drive and SSD prices do not compensate for this growth; as a result companies are spending more and more on storage [26]. One technique to reduce storage costs is deduplication, which allows sites to store less raw data. At its core, deduplication systematically replaces duplicate data chunks with references. For many real-world datasets, deduplication significantly reduces raw storage usage [10, 19, 22].

Deduplication can be implemented at several layers in the storage stack. Most existing solutions are built into file systems [4, 28, 32] because they have enough information to deduplicate efficiently without jeopardizing reliability. For example, file sizes, metadata, and on-disk layout are known to the file system; often file systems are aware of the processes and users that perform I/O. This information can be leveraged to avoid deduplicating certain blocks (e.g., metadata), or to prefetch dedup metadata (e.g., for blocks likely to be accessed together).

An alternative is to add deduplication to the block layer, which provides a simple read/write interface. Because of this simplicity, adding features to the block layer is easier than changing file systems. This observation is equally applicable to systems that work directly at the block layer, such as databases and object stores.

However, a block-level deduplication system is unaware of the context of the data it operates on. A typical I/O request contains only the operation type (read or write), size, and offset, without attached semantics such as the difference between metadata and user data. Deduplicating metadata can (1) harm reliability [25], e.g., because many file systems intentionally save several copies of critical data such as superblocks, and (2) waste computational resources because typical metadata (inode tables, directory entries, etc.) exhibits low redundancy. In particular, in-line deduplication is expensive because forming chunks (fixed or variable-length), hash calculation, and hash searches are performed before writing data to disk; it is undesirable to expend resources on data that may not benefit from deduplication.

To allow block-layer deduplication to take context into account, we propose an interface that allows file systems and applications to provide simple *hints* about the context in which the deduplication is being performed. Such hints require only minor file system changes, making them practical to add to existing, mature file systems. We implemented two hints: NODEDUP and PREFETCH, which we found useful in a wide range of cases.

To evaluate the potential benefits of hints, we used an open-source block-layer deduplication system, *dmdedup* [34]. *Dmdedup* is meant for in-line primary-storage deduplication and is implemented as a stackable block device in the Linux kernel. We evaluated our hints under a variety of workloads and mixes of unique vs. deduplicable data. Our results demonstrate that by not deduplicating data that is likely to be unique, the NODEDUP hint can speed up applications by as much as 5.3× over vanilla *Dmdedup*. We also show that by preloading hashes for data that is likely to be deduplicated soon, the PREFETCH hint can speed up applications by as much as 1.8× over vanilla *Dmdedup*.

2 Background

Context recovery. Previous research has addressed the semantic gap between the block layer and a file system and has demonstrated that restoring all or part of the context can substantially improve block-level performance and reliability [2, 18, 29–31, 36]. We build on this observation by recovering partial file system and application context to improve block-level deduplication.

Context recovery can be achieved either by *introspection* or via *hinting*. Introspection relies on block-layer intelligence to infer file-system or application operations. The benefit of introspection is that it does not require any file-system changes; the disadvantage is that a successful implementation can be difficult [33, 35]. In contrast, *hinting* asks higher layers to provide small amounts of extra information to the deduplication system. Although file systems and perhaps applications must be changed, the necessary revisions are small compared to the benefits. Furthermore, application changes can be minimized by interposing a library that can deduce hints from information such as the file or program name, file format, etc. In this work, we use hinting to recover context at the block layer.

Dmdedup. We used an open-source block-layer deduplication system, *dmdedup* [34], to evaluate the benefits of hints. *Dmdedup* uses fixed-size chunking and relies on Linux's crypto API for hashing. It can use one of two metadata back ends: `inram` and `cowbtree`; the former stores the metadata only in RAM (if it is battery-backed), and the latter writes it durably to disk. Others also proposed a soft-update based metadata backend [5].

Figure 1 depicts *dmdedup's* main components and its position in a typical setup. *Dmdedup* rests on top of physical block devices (e.g., disk drives, RAIDs, SSDs), or other logical devices (e.g., dm-crypt for encryption). It typically requires two block devices to operate: one a *data* device that stores actual user data, and a *metadata* device that keeps track of the organizational information (e.g., the hash index). In our experiments, we used an HDD for data and SSD for metadata. Placing metadata on an SSD makes sense because it is much smaller than the data itself—often less than 1% of the data—but is critical enough to require low-latency access. To upper layers, *dmdedup* provides a conventional read/write block interface. Normally, every write to a *dmdedup* instance is hashed and checked against all existing data; if a duplicate is detected, the corresponding metadata is updated and no data is written. New non-duplicate content is passed to the data device and tracked in the metadata. Since only one instance of a given block is stored, multiple files may be affected if it gets corrupted. Therefore, *dmdedup* can be run over RAID or a replication system to minimize the risk of data loss.

Internally, *dmdedup* has five components (Figure 1): **(1)** the *deduplication logic* that chunks data, computes hashes, and coordinates other components; **(2)** a *hash index* that tracks the hashes and locations of all currently stored chunks; **(3)** a *mapping* between Logical Block Numbers (LBNs) visible to the upper layers and the Physical Block Numbers (PBNs) where the actual data is stored; **(4)** a *space manager* that tracks space on the data device, maintains reference counts, allocates new blocks, and reclaims unreferenced data; and **(5)** a *chunk store* that saves user data to the data device.

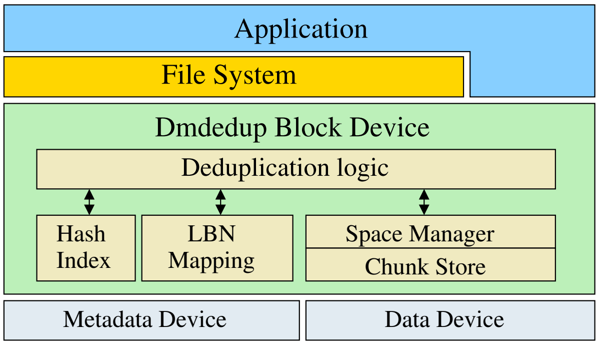


Figure 1: Dmdedup high-level design.

	Ext2	Ext3	Ext4	Nilfs2
% of writes that are metadata	11.6	28.0	18.9	12.1
% of unique metadata writes	98.5	57.6	61.2	75.0

Table 1: Percentage of metadata writes and unique metadata in different file systems.

3 Potential Hints

Bypass deduplication. Some writes are known *a priori* to be likely to be unique. Applications might generate data that should not or cannot be deduplicated. For example, some applications write random, compressed, or encrypted data; others write complex formats (e.g., virtual disk images) with internal metadata that tends to be unique [8]. HPC simulations often generate massive checkpoints with unique data, and high-resolution sensors produce unique data streams.

Attempting to deduplicate unique writes wastes CPU time on hash computation and I/O bandwidth on maintaining the hash index. Unique hashes also increase the index size, requiring more RAM space and bandwidth for lookup, insertion, and garbage collection.

Most file system metadata is unique—e.g., inodes (which have varying timestamps and block pointers), directory entries, and indirect blocks. Table 1 shows the percentage of 4KB metadata writes (unique and overall) for several file systems, using Filebench's [7] Fileserver workload adjusted to write 4KB blocks instead of 1KB (so as to match the deduplication system's chunk size). About 12–28% of the total writes across all file systems were metadata; in all cases at least 57% of the metadata was unique. Ext3 and Ext4 have more metadata duplicates than Ext2 and Nilfs2 (43% vs. 1–25%), a phenomenon caused by journaling: Ext4 initially writes metadata blocks to the journal and then writes the same blocks to their proper location on the disk.

Metadata writes are more important to overall system performance than data writes because the former are often synchronous. Adding extra deduplication overhead might increase the latency of those critical metadata writes. Avoiding excessive metadata deduplication also helps reliability because many file systems store redundant copies of their metadata (e.g., Ext2/3/4 keeps multiple superblocks; ZFS explicitly duplicates metadata to avoid corruption). Deduplicating those copies would obviate this feature. Likewise, file system journals enhance reliability, so deduplicating their blocks might be counterproductive. A deduplicated journal would also lose sequentiality, which could harm performance.

In summary, if a block-level deduplication system can know when it is unwise to deduplicate a write, it can optimize its performance and reliability. We implemented a NODEDUP hint that informs our system that a corresponding request should not be deduplicated.

Prefetch hashes. When a deduplication system knows what data is about to be written, it can prefetch the corresponding hashes from the index, accelerating future data writes by reducing lookup delays. For example, a copying process first reads source data and then writes it back. If a deduplication system can identify that behavior at read time, it can prefetch the corresponding hash entries from the index to speed up the write path. We implemented this hint and refer to it as PREFETCH. Another interesting use case for this hint is segment cleaning in log-structured file systems (e.g., Nilfs2) that migrate data between segments during garbage collection.

Bypass compression. Some deduplication systems compress chunks to save further space. However, if a file is already compressed (easily determined), additional compression consumes CPU time with no benefit.

Cluster hashes. Files that reside in the same directory tend to be accessed together [12]. In a multi-user environment, a specific user's working set is normally far smaller than the whole file system tree [13]. Based on file ownership or on which directories contain files, a deduplication system could group hashes in the index and pre-load the cache more efficiently.

Partitioned hash index. Partitioning the hash index based on incoming chunk properties is a popular technique for improving deduplication performance [1]. The chance of finding a duplicate in files of the same type is higher than across all files, so one could define partitions using, for example, file extensions.

Intelligent chunking. Knowing file boundaries allows a deduplication system to efficiently chunk data. Certain large files (e.g., tarballs) contain many small ones. Passing information about content boundaries to the block layer would enable higher deduplication ratios [15].

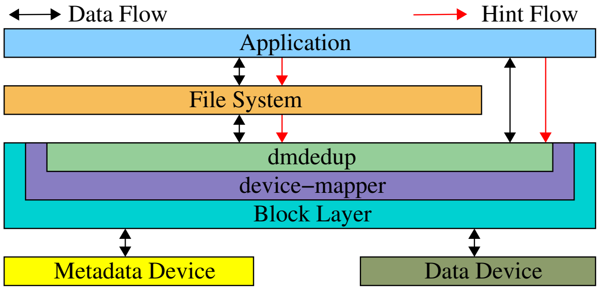


Figure 2: Flow of hints across the storage layers.

4 Design and Implementation

To allow the block layer to be aware of context, we designed a system that lets hints flow from higher to lower layers in the storage stack. Applications and file systems can then communicate important information about their data to lower layers. The red arrows in Figure 2 show how hints are passed to the block layer. We have implemented two important hints: NODEDUP and PREFETCH.

Nodedup. Since deduplication uses computational resources and may increase latency, it should only be performed when there is a potential benefit. The NODEDUP hint instructs the block layer not to deduplicate a particular chunk (block) on writes. It has two use cases: **(1)** unique data: there is no point in wasting resources on deduplicating data that is unlikely to have duplicates, such as sensor or encrypted data; **(2)** reliability: maintaining multiple copies of certain blocks may be necessary, e.g., superblock replicas in many file systems.

Prefetch. One of the most time-consuming operations in a deduplication system is hash lookup, because it often requires extra I/O operations. Worse, hashes are randomly distributed by their very nature. Hence, looking up a hash often requires random I/O, which is the slowest operation in most storage systems. Also, as previous studies have shown [38], it is impractical to keep all the hashes in memory because the hash index is far too large.

The PREFETCH hint is used to inform the deduplication system of I/O operations that are likely to generate further duplicates (e.g., during a file copy) so that their hashes can be prefetched and cached to minimize random accesses. This hint can be set on the read path for applications that expect to access the same data again. (Note that reads normally only need to access the LBN→PBN index, bypassing the hash index.)

4.1 Implementation

To add support for hints, we modified various parts of the storage stack. The generic changes to support propagation of hints from higher levels to the block layer modified about 77 lines of code in the kernel. We also modified the OPEN system call to take two new flags, O_NODEDUP and O_PREFETCH. User-space applications can use these flags to pass hints to the underlying deduplication block device. If the block layer does not support the flags, they are ignored. Applications that require redundancy or have a small number of duplicates can pass the O_NODEDUP hint when opening for write. Similarly, applications that are aware of popular data blocks, or that know some data will be accessed again, can pass O_PREFETCH when opening for read. Hashes of the blocks being read can then be prefetched, so that on a later write they can be found in the prefetch cache.

We modified *dmdedup* to support the NODEDUP and PREFETCH hints by adding and changing about 741 LoC. In *dmdedup*, if a request has the NODEDUP flag set, we skip lookups and updates in the hash index. Instead, we add an entry only to the LBN→PBN mapping. The read path needs no changes to support NODEDUP.

On the read path in *dmdedup*, the LBN→PBN map is consulted to find whether the given location is known, but no hash calculation is normally necessary because a previous write would have already added the block to the hash→PBN map. If a request has the PREFETCH hint set on the read path then *dmdedup* hashes the data after it is read and puts the corresponding hash→PBN tuple in a prefetch cache. Upon writes, our code saves execution time by checking the cache before searching the metadata backend. When a hash is found in the prefetch cache, it is evicted, since after the copy there is little reason to believe that it will be used again soon.

We also modified some specific file systems to pass the NODEDUP hint for their metadata and also pass the OPEN flags to the block layer if set. For Linux's Nilfs2, we changed about 371 kernel LoC to mark its metadata with hints and propagate them, along with the OPEN flags, from the upper levels to the block layer. Similar changes to Ext4 changed 16 lines of code; in Ext3 we modified 6 lines (which also added support for Ext2). The Ext2/3/4 changes were small because we were able to leverage the (newer) REQ_META flag being set on the file system metadata to decide whether to deduplicate based on data type. The rest of the metadata-related hints are inferred; we identify journal writes from the process name, *jbd2*.

5 Evaluation

Experimental Setup. In our experiments we used a Dell PowerEdge R710, equipped with an Intel Xeon E5540 2.4GHz 4-core CPU and 24GB of RAM. The machine ran Ubuntu Linux 14.04 x86_64, upgraded to a Linux 3.17.0 kernel. We used an Intel DC S3700 series 200GB SSD as the *dmdedup* metadata device and a Seagate Savvio 15K.2 146GB disk drive for the data. Both drives were connected to the host using Dell's PERC 6/i controller. Although the SSD is large, in all our experiments we used 1.5GB or less for *dmdedup* metadata.

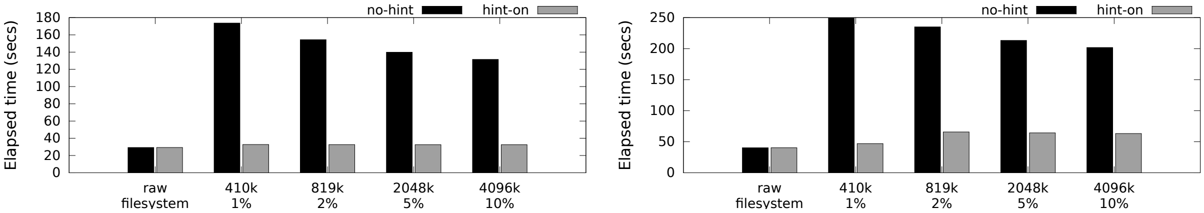
We ran all experiments at least three times and ensured that standard deviations were less than 5% of the mean. To ensure that all dirty data reached stable media in the micro-workload experiments, we called sync at the end of each run and then unmounted the file system; our time measurements include these two steps.

For all experiments we used *dmdedup*'s *cowbtree* transactional metadata backend, since it helps avoid inconsistent metadata states on crashes. *Cowbtree* allows users to specify different metadata cache sizes; we used sizes of 1%, 2%, 5%, and 10% of the deduplication metadata for each experiment. These ratios are typical in real deduplication systems. *Dmdedup* also allows users to specify the granularity at which they want to flush metadata. We ran all experiments with two settings: flush metadata on every write, or flush after every 1,000 writes. In our results we focus on the latter case because it is a more realistic setting. Flushing after every write is like using the O_SYNC flag for every operation and is uncommon in real systems; we used that setting to achieve a worst-case estimate. *Dmdedup* also flushes its metadata when it receives any flush request from the layers above. Thus, *dmdedup*'s data persistency semantics are the same as those of a regular block device.

5.1 Experiments

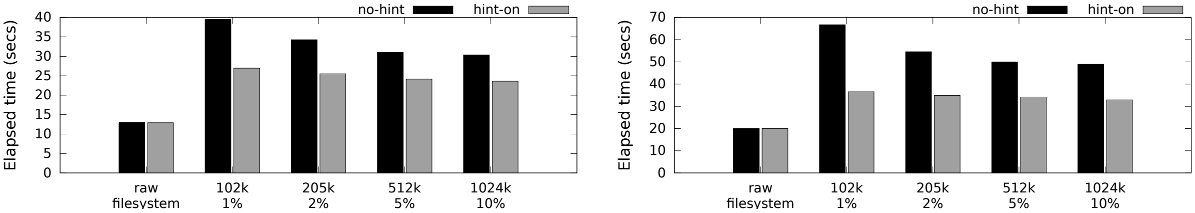
We evaluated the NODEDUP and PREFETCH hints for four file systems: Ext2, Ext3, Ext4, and Nilfs2. Ext2 is a traditional FFS-like file system that updates metadata in place; Ext3 adds journaling and Ext4 further adds extent support. Nilfs2 is a log-structured file system: it sequentializes all writes and has a garbage-collection phase to remove redundant blocks. We show results only for Ext4 and Nilfs2, because we obtained similar results from the other file systems. In all cases we found that the performance of Nilfs2 is lower than that of Ext4; others have seen similar trends [27].

NODEDUP hint. To show the effectiveness of application-layer hints, we added the NODEDUP hint as an open flag on dd's write path. We then created a 4GB file with unique data, testing with the hint both on and off. This experiment shows the benefit of the NODEDUP hint on a system where unique data is being written (i.e., where deduplication is not useful), or where reliability considerations trump deduplicating. This hint might not be as helpful in workloads that produce many duplicates. Figure 3 shows the benefit of the NODEDUP hint for Ext4 and Nilfs2 when metadata was flushed every 1,000 writes; results for other file systems were similar. We found that the NODEDUP hint decreased unique-data write times by 2.2–5.3×. Flushing *dmdedup's* metadata after every write reduced the benefit of the NODEDUP hint, since the I/O overhead was high, but we still observed improvements of 1.3–1.6×.



(a) Ext4 (b) Nilfs2

Figure 3: Performance of using `dd` *to create a 4GB file with unique content, both with and without the* NODEDUP *hint, for different file systems. The X axis lists the metadata cache size used by* dmdedup*, in both absolute values and as a percentage of the total metadata required by the workload. Dmdedup metadata was flushed after every 1,000 writes. Lower is better.*



(a) Ext4 (b) Nilfs2

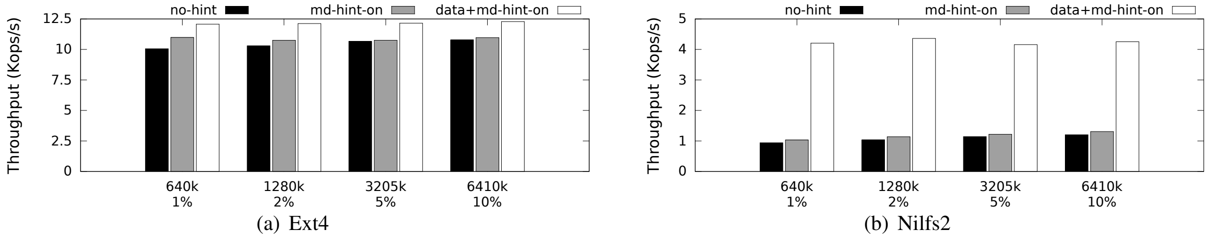
Figure 4: Performance of using `dd` *to copy a 1GB file, both with and without the* PREFETCH *hint, for different file systems. The X axis lists the metadata cache size used by* dmdedup*, in both absolute values and as a percentage of the total metadata required by workload. Dmdedup metadata was flushed after every 1,000 writes. Lower is better.*

PREFETCH hint. To evaluate the PREFETCH hint we modified `dd` to use the O_PREFETCH open flag on the read path so that writes could benefit from caching hashes. We then used the modified `dd` to repeatedly copy a 1GB file with unique content within a single file system. We used unique content so that we could measure the worst-case performance where no deduplication can happen, and to ensure that the prefetch cache was heavily used. We also performed studies on a locally collected dataset of the hashes of the home directories of a small research group. We analyzed the hashes to learn how many duplicate blocks are seen within a file using 4KB chunk sizes, and found that 99% of the files had unique chunks. Thus testing the PREFETCH hint with unique content makes sense. For all four file systems, the results were similar because most file systems manage single-file data-intensive workloads similarly. Figure 4 shows results for Ext4 and Nilfs2. When flushing *dmdedup*'s metadata every 1,000 writes, the reduction in copy time compared to the no-hint configuration was 1.2–1.8×. When we flushed the metadata after every write, the copy times ranged from 16% *worse* to 16% better. The improvement from hints was less significant here because the overhead of flushing was higher than the benefit obtained from prefetching the hashes.

Not suprisingly, the effectiveness of PREFETCH hint depends on the deduplication ratio. For example, when we changed the deduplication ratio to 8:1 in the above experiment, the copy times ranged from 9% *worse* to 55% better depending on file system type and *dmdedup* settings.

Macro workload. We modified *Filebench* to generate data in the form of a given duplicate distribution instead of arbitrary data. We then ran *Filebench's* `Fileserver` workload, modified to write 4KB blocks, to assess the benefit of setting the NODEDUP hint for: (1) file metadata writes, where we mark the metadata blocks and the journal writes with this hint, and (2) file data writes along with the metadata writes. We used a unique-write workload to show the benefits of applying the NODEDUP hint for applications writing unique data. Figure 5 shows the maximal benefit of setting the NODEDUP hint on for file metadata writes alone, and for data and metadata writes. We ran Filebench, with the all-unique writes being flushed after 1,000 writes. When setting the NODEDUP hint only for metadata writes, we saw an increase in throughput of 1–10%. When we set the hint for both data and metadata writes, we saw an improvement in throughput of 1.1–1.2× for Ext4, and 3.5–4.5× for Nilfs2. When we set the NODEDUP hint for metadata only, we observed an increase in performance but a decrease in deduplication. As calculated from Table 1, about 7.3% of all writes in Ext4 and 3.0% of all writes in Nilfs2 are duplicated file-system metadata writes. *Dmdedup* would save extra space by deduplicating these writes if the NODEDUP hint was not set. In other words, the hint trades higher throughput and reliability for a lower deduplication ratio.

We also ran a similar experiment (not shown for brevity) where Filebench generated data with a dedup ratio of 4:1 (3 duplicate blocks for every unique one). We set the NODEDUP hint for metadata writes only (because Filebench generated unique data on a per-write ba-



(a) Ext4

(b) Nilfs2

Figure 5: Throughput obtained using Filebench's `Fileserver` *workload modified to write all-unique content, for different file systems. Throughput is shown with the* NODEDUP *hint off (no-hint); with the hint on for file system metadata only (md-hint-on); and with the hint on for both file system metadata and data (data+md-hint on). The X axis lists the* dmdedup *metadata cache size, in both absolute values and as a percentage of an estimate of the total metadata required by the workload. Dmdedup metadata was flushed after every 1,000 writes. Higher is better.*

sis whereas our hint works on a per-file basis), and compared this to the case where the NODEDUP hint was off. We saw a modest improvement in throughput, ranging from 4–7% for Ext4 and 6–10% for Nilfs2.

6 Related Work

The semantic divide between the block layer and file systems has been addressed previously [2,6,29–31] and has received growing attention because of the widespread use of virtualization and the cloud, which places storage further away from applications [9,11,17,23,24,35].

An important approach to secondary storage is ADMAD, which performs application-aware file chunking before deduplicating a backup [16]. This is similar to our hints interface, which can be easily extended to pass application-aware chunk-boundary information.

Many researchers have proposed techniques to prefetch fingerprints and accelerate deduplication filtering [14,37,38]. While these techniques could be added to *dmdedup* in the future, our current focus is on providing semantic hints from higher layers, which we believe is an effective complementary method for accelerating performance. In addition, some of these past techniques rely on workload-specific data patterns (e.g., backups) that might not be beneficial in general-purpose in-line primary-storage deduplication systems.

Studies of memory deduplication in virtualized environments [20,21] show a benefit of closing the semantic gap caused by multiple virtualization layers. There, memory is scanned by the host OS to identify and merge duplicate pages. Such scanning is expensive, misses short-lived pages, and is slow to identify longer-lived duplicates. However, these studies found that pages in the guest's unified buffer cache are good sharing candidates, so marking requests from the guest OS with a dedup hint can help to quickly identify potential duplicates. This approach is specific to memory deduplication and may not apply to storage systems where we identify duplicates before writing to the disk.

Lastly, others have demonstrated a loss of potential deduplication opportunities caused by intermixing metadata and data [15], showing that having hints to avoid unnecessary deduplication might be beneficial.

7 Conclusions and Future Work

Deduplication at the block layer has two main advantages: (1) allowing any file system and application to benefit from deduplication, and (2) ease of implementation [34]. Unfortunately, application and file system context is lost at the block layer, which can harm deduplication's effectiveness. However, by adding simple yet powerful hints, we were able to provide the missing semantics to the block layer, allowing the dedup system to improve performance and possibly also reliability. Our experiments show that adding the NODEDUP hint to applications like `dd` can improve performance by up to 5.3× when copying unique data, since we avoid the overhead of deduplication for data that is unlikely to have duplicates. This hint can be extended to other applications, such as those that compress or encrypt. Adding the PREFETCH hint to applications like `dd` improved copying time by as much as 1.8× because we cache the hashes and do not need to access the metadata device to fetch them on the write path. Adding hints to macro workloads like *Filebench's* `Fileserver` workload improved throughput by as much as 4.5×. Another important note is that the effectiveness of hints depends on both the overhead added by the deduplication system, the nature of the data being written (e.g., deduplication ratio), and the workload, so all factors need to be considered when choosing to use hints.

Future work. Because of the success of our initial experiments, we intend to add hint support to other file systems, such as Btrfs and XFS. We also plan to implement other hints, discussed in Section 3, to provide richer context to the block layer, along with support to pass additional information (e.g. inode numbers) that can be used to enhance hints. We also plan to add the PREFETCH hint to Nilfs2 for segment cleaning.

Acknowledgments. We thank the anonymous FAST reviewers for their useful feedback. This work was made possible in part thanks to EMC support and NSF awards CNS-1251137 and CNS-1302246.

References

[1] L. Aronovich, R. Asher, E. Bachmat, H. Bitner, M. Hirsch, and S. Klein. The design of similarity based deduplication system. In *Proceedings of the Israeli Experimental Systems Conference (SYSTOR)*, 2009.

[2] L. N. Bairavasundaram, M. Sivathanu, A. C. Arpaci-Dusseau, and R. H. Arpaci-Dusseau. X-RAY: A non-invasive exclusive caching mechanism for RAIDs. In *Proceedings of the Annual International Symposium on Computer Architecture (ISCA)*, pages 176–187, Washington, DC, USA, 2004. IEEE Computer Society.

[3] R. E. Bohn and J. E. Short. How much information? 2009 report on American consumers. `http://hmi.ucsd.edu/pdf/HMI_2009_ConsumerReport_Dec9_2009.pdf`, December 2009.

[4] J. Bonwick. ZFS deduplication, November 2009. `http://blogs.oracle.com/bonwick/entry/zfs_dedup`.

[5] Zhuan Chen and Kai Shen. Ordermergededup: Efficient, failure-consistent deduplication on flash. In *14th USENIX Conference on File and Storage Technologies (FAST 16)*, Santa Clara, CA, February 2016. USENIX Association.

[6] A. Clements, I. Ahmad, M. Vilayannur, and J. Li. Decentralized deduplication in SAN cluster file systems. In *Proceedings of the USENIX Annual Technical Conference*, 2009.

[7] Filebench. `http://filebench.sf.net`.

[8] T. Harter, C. Dragga, M. Vaughn, A. C. Arpaci-Dusseau, and R. H. Arpaci-Dusseau. A file is not a file: understanding the I/O behavior of Apple desktop applications. In *Proceedings of the 23rd ACM Symposium on Operating System Principles (SOSP '11)*, Cascais, Portugal, October 2011. ACM Press.

[9] X. Jiang and X. Wang. "Out-of-the-box" monitoring of VM-based high-interaction honeypots. In *Proceedings of the International Conference on Recent Advances in Intrusion Detection (RAID)*, 2007.

[10] K. Jin and E. Miller. The effectiveness of deduplication on virtual machine disk images. In *Proceedings of the Israeli Experimental Systems Conference (SYSTOR)*, 2009.

[11] S. T. Jones, A. C. Arpaci-Dusseau, and R. H. Arpaci-Dusseau. Geiger: monitoring the buffer cache in a virtual machine environment. In *Proceedings of the International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, pages 14–24, New York, NY, USA, 2006. ACM Press.

[12] T. M. Kroeger and D. D. E. Long. Design and implementation of a predictive file prefetching algorithm. In *Proceedings of the Annual USENIX Technical Conference (ATC)*, pages 105–118, Boston, MA, June 2001. USENIX Association.

[13] G. H. Kuenning, G. J. Popek, and P. Reiher. An analysis of trace data for predictive file caching in mobile computing. In *Proceedings of the Summer 1994 USENIX Technical Conference*, pages 291–303, June 1994.

[14] M. Lillibridge, K. Eshghi, D. Bhagwat, V. Deolalikar, G. Trezise, and P. Camble. Sparse indexing: Large scale, inline deduplication using sampling and locality. In *Proceedings of the Seventh USENIX Conference on File and Storage Technologies (FAST '09)*, 2009.

[15] Xing Lin, Fred Douglis, Jim Li, Xudong Li, Robert Ricci, Stephen Smaldone, and Grant Wallace. Metadata considered harmful ... to deduplication. In *Proceedings of the 7th USENIX Conference on Hot Topics in Storage and File Systems*, HotStorage'15, pages 11–11, Berkeley, CA, USA, 2015. USENIX Association.

[16] C. Liu, Y. Lu, C. Shi, G. Lu, D. Du, and D.-S. Wang. ADMAD: Application-driven metadata aware de-duplication archival storage system. In *Proceedings of the International Workshop on Storage Network Architecture and Parallel I/Os (SNAPI)*, 2008.

[17] Bo Mao, Hong Jiang, Suzhen Wu, and Lei Tian. POD: Performance oriented I/O deduplication for primary storage systems in the cloud. In *28th International IEEE Parallel and Distributed Processing Symposium*, 2014.

[18] Michael Mesnier, Feng Chen, Tian Luo, and Jason B. Akers. Differentiated storage services. In *Proceedings of the Twenty-Third ACM Symposium on Operating Systems Principles*, SOSP '11, pages 57–70, New York, NY, USA, 2011. ACM.

[19] D. Meyer and W. Bolosky. A study of practical deduplication. In *Proceedings of the Ninth USENIX Conference on File and Storage Technologies (FAST '11)*, 2011.

[20] Konrad Miller, Fabian Franz, Thorsten Groeninger, Marc Rittinghaus, Marius Hillenbrand, and Frank Bellosa. KSM++: Using I/O-based hints to make memory-deduplication scanners more efficient. In *Proceedings of the ASPLOS Workshop on Runtime Environments, Systems, Layering and Virtualized*

Environments (RESoLVE'12), London, UK, March 2012.

[21] Konrad Miller, Fabian Franz, Marc Rittinghaus, Marius Hillenbrand, and Frank Bellosa. XLH: More effective memory deduplication scanners through cross-layer hints. In *Proceedings of the 2013 USENIX Annual Technical Conference (USENIX ATC 13)*, pages 279–290, San Jose, CA, 2013. USENIX.

[22] N. Park and D. Lilja. Characterizing datasets for data deduplication in backup applications. In *Proceedings of the IEEE International Symposium on Workload Characterization (IISWC)*, 2010.

[23] D. Reimer, A. Thomas, G. Ammons, T. Mummert, B. Alpern, and V. Bala. Opening black boxes: Using semantic information to combat virtual machine image sprawl. In *Proceedings of the ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments (VEE)*, Seattle, WA, March 2008. ACM.

[24] W. Richter, G. Ammons, J. Harkes, A. Goode, N. Bila, E. de Lara, V. Bala, and M. Satyanarayanan. Privacy-sensitive VM retrospection. In *Proceedings of the USENIX Workshop on Hot Topics in Cloud Computing (HotCloud)*, 2011.

[25] David Rosenthal. Deduplicating devices considered harmful. *Queue*, 9(5):30:30–30:31, May 2011.

[26] J. Rydningcom and M. Shirer. Worldwide hard disk drive 2010-2014 forecast: Sowing the seeds of change for enterprise applications. IDC Study 222797, `www.idc.com`, May 2010.

[27] Ricardo Santana, Raju Rangaswami, Vasily Tarasov, and Dean Hildebrand. A fast and slippery slope for file systems. In *Proceedings of the 3rd Workshop on Interactions of NVM/FLASH with Operating Systems and Workloads*, INFLOW '15, pages 5:1–5:8, New York, NY, USA, 2015. ACM.

[28] Opendedup, January 2012. `www.opendedup.org`.

[29] M. Sivathanu, L. N. Bairavasundaram, A. C. Arpaci-Dusseau, and R. H. Arpaci-Dusseau. Life or death at block-level. In *Proceedings of the USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, pages 379–394, San Francisco, CA, December 2004. ACM SIGOPS.

[30] M. Sivathanu, V. Prabhakaran, A. C. Arpaci-Dusseau, and R. H. Arpaci-Dusseau. Improving storage system availability with D-GRAID. In *Proceedings of the USENIX Conference on File and Storage Technologies (FAST)*, pages 15–30, San Francisco, CA, March/April 2004. USENIX Association.

[31] M. Sivathanu, V. Prabhakaran, F. I. Popovici, T. E. Denehy, A. C. Arpaci-Dusseau, and R. H. Arpaci-Dusseau. Semantically-smart disk systems. In *Proceedings of the USENIX Conference on File and Storage Technologies (FAST)*, pages 73–88, San Francisco, CA, March 2003. USENIX Association.

[32] Kiran Srinivasan, Tim Bisson, Garth Goodson, and Kaladhar Voruganti. iDedup: Latency-aware, inline data deduplication for primary storage. In *Proceedings of the Tenth USENIX Conference on File and Storage Technologies (FAST '12)*, San Jose, CA, February 2012. USENIX Association.

[33] Sahil Suneja, Canturk Isci, Eyal de Lara, and Vasanth Bala. Exploring VM introspection: Techniques and trade-offs. In *Proceedings of the 11th ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments*, 2015.

[34] V. Tarasov, D. Jain, G. Kuenning, S. Mandal, K. Palanisami, P. Shilane, and S. Trehan. Dmdedup: Device-mapper deduplication target. In *Proceedings of the Linux Symposium*, pages 83–95, Ottawa, Canada, July 2014.

[35] Vasily Tarasov, Deepak Jain, Dean Hildebrand, Renu Tewari, Geoff Kuenning, and Erez Zadok. Improving I/O performance using virtual disk introspection. In *Proceedings of the USENIX Workshop on Hot Topics in Storage and File Systems (HotStorage)*, June 2013.

[36] Eno Thereska, Hitesh Ballani, Greg O'Shea, Thomas Karagiannis, Antony Rowstron, Tom Talpey, Richard Black, and Timothy Zhu. Ioflow: A software-defined storage architecture. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*, SOSP '13, pages 182–196, New York, NY, USA, 2013. ACM.

[37] W. Xia, H. Jiang, D. Feng, and Y. Hua. SiLo: A similarity-locality based near-exact deduplication scheme with low RAM overhead and high throughput. In *Proceedings of the USENIX Annual Technical Conference*, 2011.

[38] B. Zhu, K. Li, and H. Patterson. Avoiding the disk bottleneck in the Data Domain deduplication file system. In *Proceedings of the Sixth USENIX Conference on File and Storage Technologies (FAST '08)*, 2008.