OrderMergeDedup: Efficient, Failure-Consistent Deduplication on Flash

Zhuan Chen and Kai Shen, *University of Rochester*

OrderMergeDedup: Efficient, Failure-Consistent Deduplication on Flash

Zhuan Chen and Kai Shen
Department of Computer Science, University of Rochester

Abstract

Flash storage is commonplace on mobile devices, sensors, and cloud servers. I/O deduplication is beneficial for saving the storage space and reducing expensive Flash writes. This paper presents a new approach, called *OrderMergeDedup*, that deduplicates storage writes while realizing failure-consistency, efficiency, and persistence at the same time. We devise a soft updates-style metadata write ordering that maintains storage data consistency without consistency-induced additional I/O. We further explore opportunities of I/O delay and merging to reduce the metadata I/O writes. We evaluate our Linux device mapper-based implementation using several mobile and server workloads—package installation and update, BBench web browsing, vehicle counting, Hadoop, and Yahoo Cloud Serving Benchmark. Results show that OrderMergeDedup can realize 18–63% write reduction on workloads that exhibit 23–73% write content duplication. It has significantly less metadata write overhead than alternative I/O shadowing-based deduplication. Our approach has a slight impact on the application latency and may even improve the performance due to reduced I/O load.

1 Introduction

I/O deduplication [4, 7, 17, 20, 21, 23] has been widely employed to save storage space and I/O load. I/O deduplication is beneficial for storage servers in data centers, as well as for personal devices and field-deployed sensing systems. Flash writes on smartphones and tablets may occur during package installation or through the frequent use of SQLite transactions [11, 19]. In cyber-physical systems, high volumes of data may be captured by field-deployed cameras and stored / processed for applications like intelligent transportation [22].

A deduplication system maintains metadata such as logical to physical block mapping, physical block reference counters, block fingerprints, etc. Such metadata and data structures must remain consistent on storage after system failures. It is further essential to persist writes in a prompt manner to satisfy the storage durability semantics. On Flash storage, I/O deduplication must also minimize the expensive Flash writes resulted from metadata management. This paper presents a new I/O dedu-

plication mechanism that meets these goals.

Specifically, we order the deduplication metadata and data writes carefully so that any fail-stop failure produces no other data inconsistency (on durable storage) than uncollected garbage. This approach is, in concept, similar to soft updates-based file system design [6]. It is efficient by not requiring additional I/O traffic for maintaining consistency (in contrast to logging / journaling or shadowing-based I/O atomicity). While the original file system soft updates suffer from dependency cycles and rollbacks [18], the relatively simple structure of deduplicated storage allows us to recognize and remove all possible dependency cycles with no impact on performance.

The metadata I/O overhead can be further reduced by merging multiple logical writes that share common deduplication metadata blocks. In particular, we sometimes delay I/O operations in anticipation for metadata I/O merging opportunities in the near future. Anticipatory I/O delay and merging may prolong responses to the user if the delayed I/O is waited on due to the data persistence semantics. We show that the performance impact is slight when the delay is limited to a short duration. It may even improve the application latency due to reduced I/O load. With failure-consistent I/O ordering and anticipatory merging, we name our deduplication approach *OrderMergeDedup*.

We have implemented our OrderMergeDedup approach in Linux 3.14.29 kernel as a custom device mapper target. Our prototype system runs on an Intel Atom-based tablet computer and an Intel Xeon server machine. We have experimentally evaluated our system using a range of mobile and server workloads.

Data consistency over failures was not ignored in prior deduplication systems. iDedup [20] relied on a non-volatile RAM to stage writes (in a log-structured fashion) that can survive system failures. Other systems [4, 7] utilized supercapacitors or batteries to allow continued data maintenance after power failures. Our deduplication approach does not assume the availability of such hardware aids. Venti [17] provided consistency checking and repair tools that can recover from failures at a significant cost in time. dedupv1 [13] maintained a redo log to recover from failures but redo logging incurs the cost of additional writes (even at the absence of failures). Most recently, Dmdedup [21] supported data consistency after failures through I/O shadowing, which incurs the cost of

additional index block writes. It achieved efficiency by delaying and batch-flushing a large number of metadata updates but such delayed batching is hindered by synchronous writes in some applications and databases.

2 Design of OrderMergeDedup

I/O deduplication eliminates duplicate writes in the I/O stream. We capture all write I/O blocks at the device layer for deduplication. With a fixed-sized chunking approach, each 4 KB incoming data block is intercepted and a hashed fingerprint is computed from its content. This fingerprint is looked up against the fingerprints of existing storage blocks to identify duplicates.

A deduplication system maintains additional metadata information. Specifically, a logical-to-physical block mapping directs a logical block access (with its logical address) to its physical content on storage. For each physical block, the associated reference counter records the number of logical blocks mapped to it, and the fingerprint is computed to facilitate the block content matching. A write request received by a deduplication system can result in a series of physical writes to both the block data and metadata. For deduplication metadata management, it is challenging to realize (1) *failure consistency*—data/metadata writes must be carefully performed to enable fast, consistent recovery after failures; (2) *efficiency*—the additional I/O cost incurred by metadata writes should not significantly diminish deduplication I/O saving; (3) *persistence*—the deduplication layer should not prematurely return an I/O write in violation of persistence semantics.

2.1 I/O Ordering for Failure-Consistency

File and storage systems [6, 9] have recognized the importance of atomic I/O to support consistent failure-recovery. Existing techniques include journaling, shadowing [1, 9], and soft updates [6]. In journaling, an atomic I/O operation is recorded in a redo log before writing to the file system. A failure after a partial write can be recovered at system restart by running the redo log. In shadowing, writes to existing files are handled in a copy-on-write fashion to temporary shadow blocks. The final commit is realized through one atomic I/O write to a file index block that points to updated shadow data/index blocks. Index blocks (potentially at multiple hierarchy levels) must be re-written to create a complete shadow. Both journaling and shadowing require additional write I/O to achieve failure consistency of durable data.

The soft updates approach [6] carefully orders writes in file system operations such that any mid-operation failure always leaves the file system structure in a consistent state (except for possible space leaking on temporar-

ily written blocks). While it requires no I/O overhead during normal operations, rollbacks may be necessary to resolve cyclic dependencies in the block commit order. Seltzer et al. [18] showed that such rollbacks sometimes led to poor soft updates performance on the UNIX Fast File System. Due to relatively simple semantics of a deduplicated storage (compared to a file system), we show that a careful design of all deduplication I/O paths can efficiently resolve possible dependency cycles. We next present our soft updates-style deduplication design.

A unique aspect of our design is that our physical block reference counter counts logical block references as well as a reference from the physical block's fingerprint. Consequently the reclamation of block fingerprint does not have to occur together with the removal of the last logic block referencing the physical block. Separating them into two failure-consistent transactions makes each less complex and reduces the probability of cyclically dependent write ordering. It also allows fingerprint reclamation to be delayed—e.g., performed offline periodically. Lazy fingerprint reclamation may improve performance since the same data rewritten after a period of non-existence may still be deduplicated. Such scenario has been shown to happen in certain workloads [16].

Specifically, we maintain the following ordering between I/O operations during deduplication.

1. The physical data block should always be persisted before being linked with the logical address or the computed fingerprint. A failure recovery may leave some data block inaccessible, but will never lead to any logical address or fingerprint that points to incorrect content.

2. For reference counters, we guarantee that when a sudden failure occurs, the only possibly resulted inconsistency is higher-than-actual reference counters for some physical blocks. A higher-than-actual reference counter may produce garbage (that can be reclaimed asynchronously) while a lower-than-actual reference counter could lead to the serious damage of premature block deletion. To achieve this goal, a new linkage that points to a physical block from some logical address or fingerprint must be preceded by the increment of the physical block's reference counter, and the corresponding unlinking operations must precede the decrement of the physical block's reference counter.

3. Meanwhile, the update of the logical-to-physical block mapping and fingerprints can be processed in parallel since there is no failure-consistent dependency between them.

Figure 1 illustrates our complete soft updates-style write ordering in different write conditions.
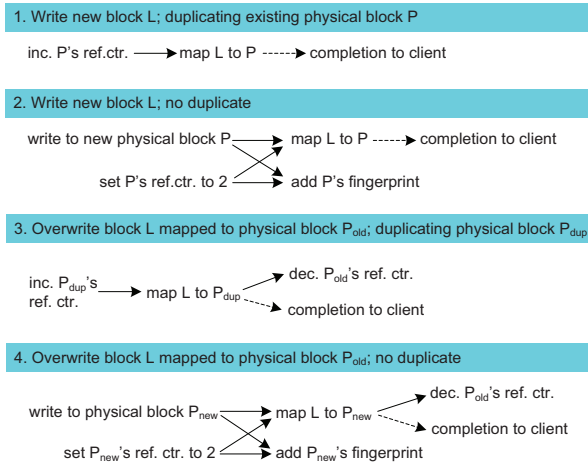
Figure 1: Failure-consistent deduplication write ordering at different write conditions (new write / overwrite, duplicate identified or not, etc.). Solid arrows indicate ordering of writes. The dashed arrow in each case shows when a completion signal is sent to client. Note that a new physical block' reference counter begins at two—one reference from the first logical block and the other from the fingerprint.

A block metadata entry is much smaller than the block itself (e.g., we use the 64-bit address for physical block indexing and the 8-bit reference counter [1]). The I/O cost can be reduced when multiple metadata writes that fall into the same metadata block are merged into one I/O operation. Merging opportunities arise for metadata writes as long as they are not subject to any ordering constraint.

While the metadata write merging presents apparent benefit for I/O reduction, the merging may create additional write ordering constraints and lead to cyclic dependencies or deadlocks. For example, for cases 3 and 4 in Figure 1, if the reference counters for new block P_{dup} / P_{new} and old block P_{old} are on the same metadata block, then the merging of their reference counter updates would cause a deadlock. No metadata write merging is allowed for such situation.

Cyclic dependencies prevent metadata write merging for cost saving and also complicate the implementation. We resolve this issue by delaying the *non-critical* metadata updates involved in the cyclic dependencies. A non-critical update is the write operation that the client completion signal does not depend on, particularly for the same example above, the decrement of P_{old}'s reference counter and the follow-up work in cases 3 and 4 of Figure 1. A delay of those operations until their associated dependencies are cleared simply eliminates the cyclic de-

---

[1]A reference counter overflow will lead to the allocation of another new physical block to hold the same block content. Later writes with such content will be mapped to the new block.

pendencies. Since the delayed action is not on the critical path of I/O completion, client I/O response will not be affected. Fortunately, under our soft updates-style deduplication metadata management, all the potential deadlocks can be resolved in this way.

2.2 Metadata I/O Merging for Efficiency

As mentioned, substantial I/O cost reduction may result from the merging of multiple metadata writes that fall into the same metadata block. We may enhance the opportunities of metadata I/O merging by delaying metadata update I/O operations so their chances of merging with future metadata writes increase. We explore several opportunities of such I/O delay.

Weak persistence Our deduplication system supports two persistence models with varying performance trade-offs. Specifically, a *strong persistence* model faithfully preserves the persistence support of the underlying device—an I/O operation is returned by the deduplication layer only after all corresponding physical I/O operations are returned from the device. On the other hand, a *weak persistence* model performs writes asynchronously under which a write is returned early while the corresponding physical I/O operations can be delayed to the next flush or Force Unit Access (FUA)-flagged request.

Under weak persistence, I/O operations can be delayed aggressively to present metadata I/O merging opportunities. Such delays, however, may be hindered by synchronous writes in some applications and databases.

Non-critical I/O delay and merging The example in Section 2.1 shows the removal of cyclic dependencies through the delay of some non-critical metadata updates. In fact, those updates can be free if the delay lasts long enough to merge with future metadata writes that reside on the same metadata block. Moreover, this applies to all the non-critical metadata writes. Specifically, besides the example mentioned in Section 2.1, we also aggressively delay the operations of fingerprint insertion for P / P_{new} in cases 2 and 4 of Figure 1. A sudden system failure may leave some of the reference counters to be higher than actual values, resulting in unreclaimed garbage, or lose some fingerprints for the physical block deduplication chances, but no other serious inconsistency occurs.

Anticipatory I/O delay and merging Two metadata writes to the same physical block will generally result in separate device commits if the interval between their executions is longer than the typical cycle upon which a deduplication system writes to the physical device. If such interval is small, it may be beneficial to impose a short idle time to the physical device (by

stop issuing writes to it) to generate more opportunities of metadata write merging. This is reminiscent of the I/O anticipation scheduling which was proposed as a performance-enhancing seek-reduction technique for mechanical disks [10]. In our case, we temporarily idle the physical device in anticipation of soon-arriving desirable requests for metadata update merging with the existing one.

Anticipatory I/O delay and merging may gain high benefits under a high density of write requests because a short anticipatory device idle period would produce high degree of merging. On the other hand, a light load affords little merging opportunity so that anticipatory device idling only prolongs the I/O response latency. To maximize the benefit of anticipatory I/O delay and merging, we apply a simple heuristic hint as the guidance—the frequency of incoming write requests received by the deduplication system. Intuitively, if the idling period can cover more than one incoming write request, a metadata write merging is likely to happen. We only enable the anticipatory I/O delay and merging under this situation.

3   Implementation

We have implemented our deduplication mechanism at the generic operating system block device layer to enable transparent full-device deduplication for software applications. Specifically, our mechanism is implemented in Linux 3.14.29 kernel as a custom device mapper target. Our implementation follows the basic block read/write interfaces for deduplication checks in the open-source Dmdedup [21] framework.

Duplicates of 4 KB I/O blocks are identified through their hashed fingerprints. We use the widely adopted SHA-1 cryptographic hash algorithm to produce 160-bit (20-byte) block fingerprints. The SHA-1 hashes are collision resistant [14] and we deem two blocks as duplicates if they have matching fingerprints without performing a full block comparison. This is a widely-accepted practice in data deduplication [4, 15, 17, 23] since the chance of hash collision between two different blocks is negligible—less than the error rate of memory and network transfer. A hash table is maintained to organize fingerprints in memory. We partition the fingerprint value space into N segments according to the total number of physical data blocks, and for each fingerprint f, map it to the corresponding segment ($f \bmod N$).

For simplicity, we organize metadata blocks on storage as linear tables. A possible future enhancement is to use a radix-tree structure. The radix tree hierarchical writes could be incorporated into our failure-consistent write ordering without introducing cyclic dependencies.

File systems maintain redundant durable copies of critical information such as the superblock for reliability.

For Ext4 file systems, multiple copies of the superblock and block group descriptors are kept across the file system while the main copy resides at the first a few blocks. Deduplicating these blocks could harm such reliability-oriented redundancy measure. We adopt a simple approach to prevent the deduplication of the main copy of the critical file system information (with recognizable block addresses). Specifically, we do not keep their fingerprints in the cache for deduplication; we do not attempt to deduplicate a write to such a block either. A possible future enhancement is to assist such decisions based on hints directly passed from the file system [12].

In our implementation, we delay the non-critical metadata writes for 30 seconds after their failure-consistent dependencies are cleared (during this period they may be merged with other incoming metadata updates residing on the same metadata block). We choose the 1-millisecond idling period for the anticipatory I/O delay and merging which is at the same magnitude as the Flash write latency of our experimental platforms. Our deduplication system is configured with the weak persistence model by default. For better balance between performance and persistence, we periodically commit the delayed metadata writes (besides the synchronous flush or FUA-flagged requests and non-critical metadata writes) to the physical device every 1 second. This is the same setup supported by other device mapper targets in Linux (e.g., `dm-cache`). When data durability is critical, our deduplication system also supports the strong persistence model described in Section 2.2. Our evaluation will cover both models.

4   Evaluation

We evaluate the effectiveness of our proposed deduplication systems on mobile and server workloads. We will compare I/O saving and impact on application performance under several deduplication approaches. We will also assess the deduplication-resulted storage space saving and impact on mobile energy usage.

4.1   Evaluation Setup

Mobile system evaluation setup   Our mobile experiments run on an Asus Transformer Book T100 tablet. It contains a 1.33 GHz quad-core Atom (x86) Z3740 processor and 2 GB memory. We deploy the Ubuntu 12.04 Linux distribution with 3.14.29 kernel. The tablet has an internal 64 GB Flash storage with the following random read/write latency (in mSecs)—

	4KB	8KB	16KB	32KB	64KB	128KB
Read	0.27	0.32	0.40	0.63	0.89	1.45
Write	2.91	2.86	4.33	4.96	7.64	11.60

We use the following mobile application workloads—

Advanced Packaging Tool (APT) is a software package installation and maintenance tool on Linux. We study two common package management scenarios via the `apt-get` command: 1) the global package index update (`sudo apt-get update`) and 2) the installation of Firefox (`sudo apt-get install firefox`). We evaluate these workloads under a Ubuntu 12.04 `chroot` environment to facilitate the capture of I/O throughout the root directory. To minimize the noises such as network latencies, we set up the complete Ubuntu 12.04 software repository and pre-download the necessary packages outside the `chroot` jail. The package index update and installation workloads exhibit 23% and 30% write content duplication respectively.

BBench [8] is a smartphone benchmarking tool to assess a web-browser's performance. We run BBench under Firefox 40.0.3 with its provided workloads which include some of the most popular and complex sites on the web. The same setup of Ubuntu 12.04 `chroot` environment (as above) is used along with the BBench web sites workloads located outside the jail. The workload exhibits 73% write duplication.

A field sensor-based *vehicle counting* application that monitors the number and frequency of passing vehicles can help detect the traffic volume, congestion level, and abnormal traffic patterns. Our application leverages the Canny edge detector algorithm [3] from the OpenCV computer vision library. It observes the moving vehicles and records the frames at the time when those vehicles enter the monitored zones. Nearby images in a data stream are often substantially similar, and exhibit block-level redundancy under JPEG/JFIF-style image formats that split an image into multiple sub-regions and encode each separately. We use the pre-collected California highway video streams (at 10 frames per second) from the publicly accessible Caltrans live traffic data [2]. The workload exhibits 27% write duplication.

Server system evaluation setup Our server experiments run on a dual-socket machine where each socket contains an Intel Xeon E5-2620 v3 "Haswell" processor. We deploy the Fedora 20 Linux distribution with 3.14.29 kernel. We perform I/O on a Samsung 850 Pro SSD (256GB) with the following I/O latency (in mSecs)—

	4KB	8KB	16KB	32KB	64KB	128KB
Read	0.12	0.13	0.15	0.18	0.28	0.45
Write	4.70	4.96	5.45	6.13	7.18	7.35

We use the following server / cloud workloads—

Hadoop software library is a framework to use simple programming models for large data sets processing across computers, each offering local computation and storage. We apply the regular expression match of "`dedup[a-z]*`" via Hadoop to all files in the `Documentation` directory of Linux 3.14.29 kernel re-

lease. The workload exhibits 55% write duplication on Hadoop's temporary files.

Yahoo Cloud Serving Benchmark (YCSB) [5] is a benchmarking framework for could evaluation. It particularly focuses on the online read / write access-based web serving systems. We perform the YCSB-0.5.0 client under MongoDB-3.2 database. Server load is generated based on the provided `workloada` of YCSB. We tune the parameters of `recordcount` and `operationcount` to 100 and 20,000 respectively, and set `fieldlength` to 8 KB. The workload exhibits 24% write duplication.

4.2 Evaluation on Deduplication Performance

We compare the total volume of Flash I/O writes (including the original application write data and our deduplication metadata, in the 4 KB unit) and application performance (execution latency) between the following system setups—1) the original system that does not support I/O deduplication; 2) I/O shadowing-based Dmdedup [21]; 3) our deduplication system with failure-consistent I/O ordering; our further optimizations of 4) non-critical I/O delay and merging and 5) anticipatory I/O delay and merging. Traces are acquired at the storage device layer to compare the write volumes sent to the storage device under different system setups.

This section evaluates the deduplication performance under a weak persistence model—device writes are performed asynchronously in batches for high efficiency; a write batch is issued at the arrival of a flush or FUA-flagged request, or issued every second at the absence of any flush or FUA-flagged request. We also adapt Dmdedup to follow this pattern. The performance of supporting strong persistence is reported in the next section.

Figure 2 (A) illustrates the results of normalized Flash I/O write volumes under different system conditions. The blue line in the figure indicates the ideal-case deduplication ratio that can only be realized without any deduplication metadata writes. We use the original execution without deduplication as the basis. Dmdedup achieves 7–33% I/O savings for package update / installation, Vehicle counting, Hadoop, and YCSB, but adds 7% I/O writes for BBench. In comparison, our deduplication system with failure-consistent I/O ordering reduces the Flash writes by 17–59% for all the workload cases. The optimization of non-critical I/O delay and merging brings slight benefits (up to 2%) except for the BBench case where 8% additional saving on Flash I/O writes is reached. The optimization of anticipatory I/O delay and merging further increases the I/O saving up to another 6% for all the workloads. Overall, we save 18–63% Flash I/O writes compared to the original non-deduplicated case. These results are very close to the ideal-case deduplication ratios for these workloads.
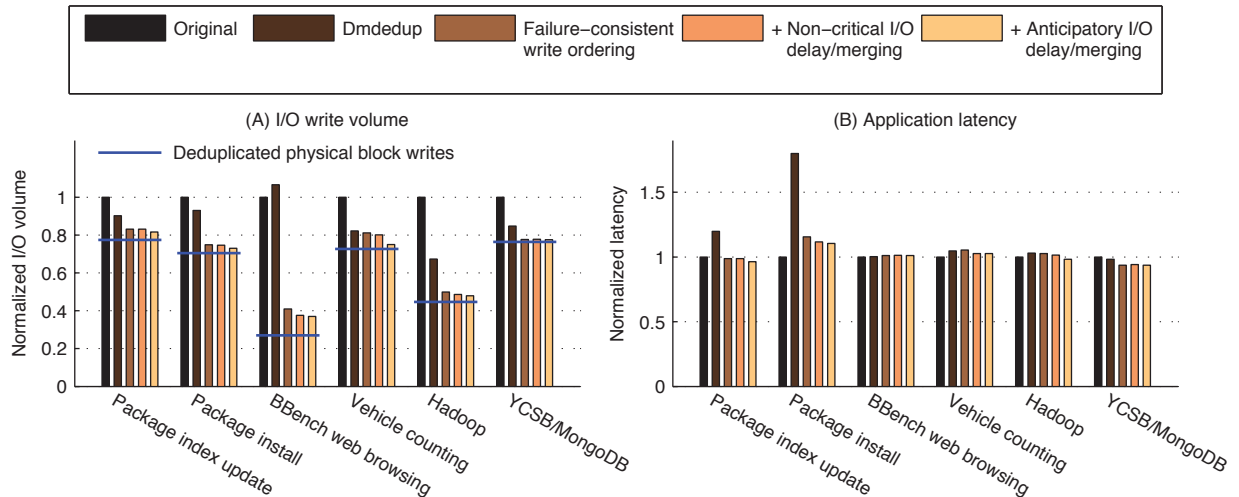
Figure 2: Flash I/O write volume and application performance of different I/O deduplication approaches. The performance in each case is normalized to that under the original (non-deduplicated) execution.
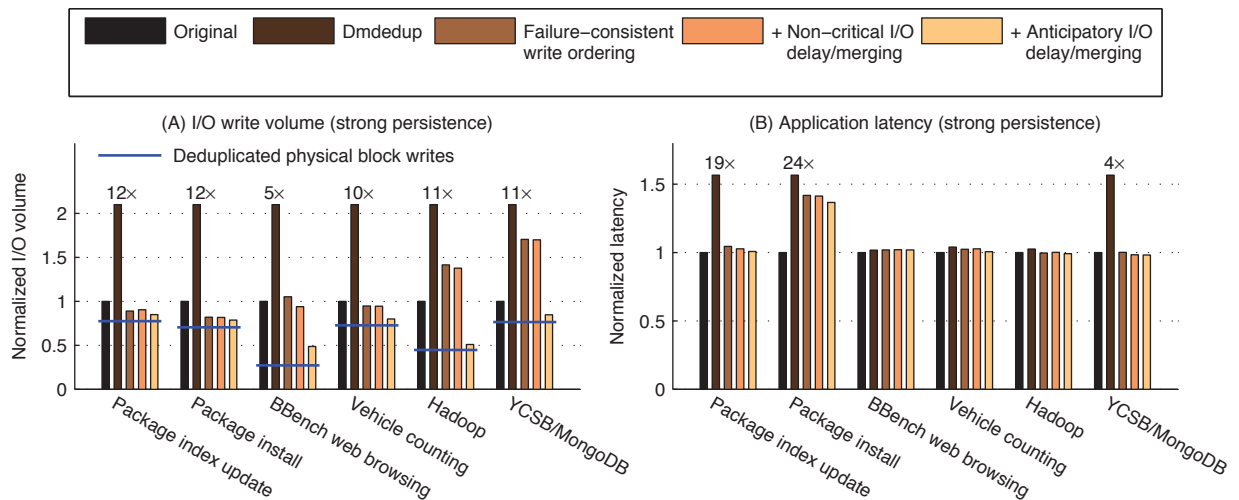

Figure 3: Flash I/O write volume and application performance when supporting strong I/O persistence. The performance in each case is normalized to that under the original (non-deduplicated) execution.

We also evaluate the application performance impact due to deduplication. Figure 2 (B) shows application execution latencies under the same system setups as above. We use the original execution without deduplication as the basis. While Dmdedup has small impact (less than 5% of performance overhead) to the workloads of BBench, Vehicle counting, Hadoop, and YCSB, it increases the costs to $1.2\times$ and $1.8\times$ for package update and installation respectively. In comparison, our deduplication system, either with or without optimizations, only imposes moderate overhead to the package installation case (around 11–15%). The impacts to other workloads' performance are small (less than 5%) and in some cases we actually achieve slight performance improvement (around 1–6%).

4.3 Evaluation under Strong Persistence

We evaluate the performance of our deduplication system between the similar system setups as Section 4.2, but with the support of strong persistence model—an I/O operation is returned by the deduplication layer only after all corresponding physical I/O operations are returned from the device. Dmdedup is configured accordingly with the single-write transaction size—the corresponding data / metadata updates are committed after every write operation [21].

Figure 3 (A) illustrates the results of normalized Flash I/O write volumes under different system conditions. We use the original execution without deduplication as the basis. Dmdedup adds large overhead for all the work-

loads from 5× to 12×. In comparison, our deduplication system with failure-consistent I/O ordering reduces the Flash writes by 5–18% for package update / installation and vehicle counting. Meanwhile, it adds 5% I/O volumes for BBench while such overhead becomes large for Hadoop and YCSB (around 41–71%). The optimization of non-critical I/O delay and merging brings slight benefits (up to 3%) except for the BBench case where 11% additional saving on Flash I/O writes is reached. The optimization of anticipatory I/O delay and merging exhibits significant benefit for all the workloads under the strong persistence model. Specifically, enabling it brings up to 63% additional saving on Flash I/O writes. Overall, we save 15–51% Flash I/O writes compared to the original non-deduplicated case.

Figure 3 (B) shows application execution latencies under the same system setups as above. We use the original execution without deduplication as the basis. Dmdedup adds large overhead for package update / installation and YCSB from 4× to 24× while the performance impact is small for other workloads (less than 4%). In comparison, our deduplication system, either with or without optimizations, only imposes large overhead to the package installation case (around 37–42%). The impacts to other workloads' performance are small (less than 4%).

4.4 Evaluation on Storage Space Saving

We compare the space usage between the non-deduplicated system and our deduplication system. Under the non-deduplicated execution, we directly calculate the occupied blocks during the workload running. For our deduplication system, the space usage is obtained by putting together the following items— 1) the space for physical blocks written along with the corresponding physical block metadata (reference counters and fingerprints); 2) the space for logical block metadata (logical-to-physical block mapping) for the occupied logical blocks. The table below shows that the workload executions exhibit strong deduplication space saving—

Workload	Space usage		Saving
	Original	Dedup	
Update	168.8 MB	131.5 MB	22%
Install	137.9 MB	98.9 MB	28%
BBench	11.0 MB	4.8 MB	56%
Vehicle	12.8 MB	9.4 MB	27%
Hadoop	80.0 MB	39.8 MB	50%
YCSB	800.6 MB	618.5 MB	23%

4.5 Evaluation on Mobile Energy Usage

We assess the energy impact of our deduplication system on mobile platforms. The energy usage of a workload is the product of its power consumption and runtime. The runtime is normally the application latency, except in the case of our vehicle counting workload where the application operates at a fixed frame-per-second rate and therefore its runtime is not affected by the frame processing latency. We compare the power usage, runtime difference, and energy usage between the original (non-deduplicated) system and our deduplication system—

Workload	Power (Watts)		Runtime	Energy
	Orig.	Dedup	impact	impact
Update	6.35	6.41	-3.6%	-3%
Install	6.03	6.06	+10.5%	+11%
BBench	6.10	6.15	+1.1%	+2%
Vehicle	6.70	6.70	0.0%	0%

Results show that our deduplication mechanism adds 11% energy usage for package installation, mostly due to the increase of runtime. The energy impact is no more than 2% in the other three workloads. The energy usage even decreases by 3% for package index update primarily due to a reduction in runtime.

5 Conclusion

This paper presents a new I/O mechanism, called OrderMergeDedup, that deduplicates writes to the primary Flash storage with failure-consistency and high efficiency. We devise a soft updates-style metadata write ordering that maintains data / metadata consistency over failures (without consistency-induced additional I/O) on the storage. We further use anticipatory I/O delay and merging to reduce the metadata I/O writes. We have made a prototype implementation at the Linux device mapper layer and experimented with a range of mobile and server workloads.

Results show that OrderMergeDedup is highly effective—realizing 18–63% write reduction on workloads that exhibit 23–73% write content duplication. We also save up to 56% in space usage. The anticipatory I/O delay optimization is particularly effective to increase metadata merging opportunities when supporting the strong I/O persistence model. OrderMergeDedup has a slight impact on the application latency and mobile energy. It may even improve the application performance due to reduced I/O load.

References

[1] M. M. Astrahan, M. W. Blasgen, D. D. Chamberlin, K. P. Eswaran, J. N. Gray, P. P. Griffiths, W. F. King, R. A. Lorie, P. R. McJones, J. W. Mehl, et al. System R: Relational approach to database management. *ACM Transactions on Database Systems (TODS)*, 1(2):97–137, 1976.

[2] Live traffic cameras, california department of transportation. `video.dot.ca.gov`.

[3] J. Canny. A computational approach to edge detection. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, (6):679–698, 1986.

[4] F. Chen, T. Luo, and X. Zhang. CAFTL: A content-aware flash translation layer enhancing the lifespan of Flash memory based solid state drives. In *the 9th USENIX Conf. on File and Storage Technologies (FAST)*, San Jose, CA, Feb. 2011.

[5] B. F. Cooper, A. Silberstein, E. Tam, R. Ramakrishnan, and R. Sears. Benchmarking cloud serving systems with YCSB. In *the First ACM Symp. on Cloud Computing (SOCC)*, pages 143–154, Indianapolis, IN, June 2010.

[6] G. R. Ganger, M. K. McKusick, C. A. N. Soules, and Y. N. Patt. Soft updates: A solution to the metadata update problem in file systems. *ACM Trans. on Computer Systems*, 18(2):127–153, May 2000.

[7] A. Gupta, R. Pisolkar, B. Urgaonkar, and A. Sivasubramaniam. Leveraging value locality in optimizing NAND flash-based SSDs. In *the 9th USENIX Conf. on File and Storage Technologies (FAST)*, San Jose, CA, Feb. 2011.

[8] A. Gutierrez, R. Dreslinski, T. Wenisch, T. Mudge, A. Saidi, C. Emmons, and N. Paver. Full-system analysis and characterization of interactive smartphone applications. In *the IEEE Intl. Symp. on Workload Characterization (IISWC)*, pages 81–90, Austin, TX, Nov. 2011.

[9] D. Hitz, J. Lau, and M. A. Malcolm. File system design for an NFS file server appliance. In *USENIX Winter Technical Conf.*, San Francisco, CA, Jan. 1994.

[10] S. Iyer and P. Druschel. Anticipatory scheduling: A disk scheduling framework to overcome deceptive idleness in synchronous I/O. In *the 8th ACM Symp. on Operating Systems Principles (SOSP)*, pages 117–130, Banff, Alberta, Canada, Oct. 2001.

[11] H. Kim, N. Agrawal, and C. Ungureanu. Revisiting storage for smartphones. In *the 10th USENIX Conf. on File and Storage Technologies (FAST)*, San Jose, CA, Feb. 2012.

[12] S. Mandal, G. Kuenning, D. Ok, V. Shastry, P. Shilane, S. Zhen, V. Tarasov, and E. Zadok. Using hints to improve inline block-layer deduplication. In *the 14th USENIX Conf. on File and Storage Technologies (FAST)*, Santa Clara, CA, Feb. 2016.

[13] D. Meister and A. Brinkmann. dedupv1: Improving deduplication throughput using solid state drives (SSD). In *IEEE 26th Symp. on Mass Storage Systems and Technologies (MSST)*, pages 1–6, May 2010.

[14] A. J. Menezes, P. C. van Oorschot, and S. A. Vanstone. *Handbook of Applied Cryptography*. CRC Press, 1996.

[15] A. Muthitacharoen, B. Chen, and D. Mazières. A low-bandwidth network file system. In *the 8th ACM Symp. on Operating Systems Principles (SOSP)*, pages 174–187, Banff, Alberta, Canada, Oct. 2001.

[16] P. Nath, M. A. Kozuch, D. R. O'Hallaron, J. Harkes, M. Satyanarayanan, N. Tolia, and M. Toups. Design tradeoffs in applying content addressable storage to enterprise-scale systems based on virtual machines. In *USENIX Annual Technical Conf.*, pages 71–84, Boston, MA, June 2006.

[17] S. Quinlan and S. Dorward. Venti: A new approach to archival storage. In *the First USENIX Conf. on File and Storage Technologies (FAST)*, Monterey, CA, Jan. 2002.

[18] M. I. Seltzer, G. R. Granger, M. K. McKusick, K. A. Smith, C. A. N. Soules, and C. A. Stein. Journaling versus soft updates: Asynchronous metadata protection in file systems. In *USENIX Annual Technical Conf.*, San Deigo, CA, June 2000.

[19] K. Shen, S. Park, and M. Zhu. Journaling of journal is (almost) free. In *the 12th USENIX Conf. on File and Storage Technologies (FAST)*, pages 287–293, Santa Clara, CA, Feb. 2014.

[20] K. Srinivasan, T. Bisson, G. Goodson, and K. Voruganti. iDedup: Latency-aware, inline data deduplication for primary storage. In *the 10th USENIX Conf. on File and Storage Technologies (FAST)*, San Jose, CA, Feb. 2012.

[21] V. Tarasov, D. Jain, G. Kuenning, S. Mandal, K. Palanisami, P. Shilane, S. Trehan, and E. Zadok.

Dmdedup: Device mapper target for data deduplication. In *Ottawa Linux Symp.*, Ottawa, Canada, July 2014.

[22] P. F. Williams. Street smarts: How intelligent transportation systems save money, lives and the environment. Technical report, ACS Transportation Solutions Group, Xerox, Feb. 2009.

[23] B. Zhu, K. Li, and H. Patterson. Avoiding the disk bottleneck in the data domain deduplication file system. In *the 6th USENIX Conf. on File and Storage Technologies (FAST)*, pages 269–282, San Jose, CA, Feb. 2008.