CAFTL: A Content-Aware Flash Translation Layer Enhancing the Lifespan of Flash Memory based Solid State Drives

Feng Chen* Tian Luo Xiaodong Zhang

Dept. of Computer Science & Engineering
The Ohio State University
Columbus, OH 43210, USA
{*fchen,luot,zhang*}@*cse.ohio-state.edu*

Abstract

Although Flash Memory based Solid State Drive (SSD) exhibits high performance and low power consumption, a critical concern is its limited lifespan along with the associated reliability issues. In this paper, we propose to build a Content-Aware Flash Translation Layer (CAFTL) to enhance the endurance of SSDs at the device level. With no need of any semantic information from the host, CAFTL can effectively reduce write traffic to flash memory by removing unnecessary duplicate writes and can also substantially extend available free flash memory space by coalescing redundant data in SSDs, which further improves the efficiency of garbage collection and wear-leveling. In order to retain high data access performance, we have also designed a set of acceleration techniques to reduce the runtime overhead and minimize the performance impact caused by extra computational cost. Our experimental results show that our solution can effectively identify up to 86.2% of the duplicate writes, which translates to a write traffic reduction of up to 24.2% and extends the flash space by a factor of up to 31.2%. Meanwhile, CAFTL only incurs a minimized performance overhead by a factor of up to 0.5%.

1 Introduction

The limited lifespan is the *Achilles' heel* of Flash Memory based Solid State Drives (SSDs). On one hand, SSDs built on semiconductor chips without any moving parts have exhibited many unique technical merits compared with hard disk drives (HDDs), particularly high random access performance and low power consumption. On the other hand, the limited lifespan of SSDs, which are built on flash memories with limited erase/program cycles, is still one of the most critical concerns that seriously hinder a wide deployment of SSDs in reliability-sensitive environments, such as data centers [10]. Although SSD manufacturers often claim that SSDs can sustain routine usage for years, the technical concerns about the endurance issues of SSDs still remain high. This is mainly due to three not-so-well-known reasons. First, as bit density increases, flash memory chips become more affordable but, at the same time, less reliable and less durable. In the last two years, for high-density flash devices, we have seen a sharp drop of erase/program cycle ratings from ten thousand to five thousand cycles [7]. As technology scaling continues, this situation could become even worse. Second, traditional redundancy solutions such as RAID, which have been widely used for battling disk failures, are considered less effective for SSDs, because of the high probability of correlated device failures in SSD-based RAID [9]. Finally, although some prior research work [13, 22, 33] has presented empirical and modeling-based studies on the lifespan of flash memories and USB flash drives, both positive and negative results have been reported. In fact, as a recent report from Google® points out, "endurance and retention (of SSDs) not yet proven in the field" [10].

All these aforesaid issues explain why commercial users hesitate to perform a large-scale deployment of SSDs in production systems and why integrating SSDs into commercial systems is proceeding such "painfully slowly" [10]. In order to integrate such a "frustrating technology", which comes with equally outstanding merits and limits, into the existing storage hierarchy timely and reliably, solutions for effectively improving the lifespan of SSDs are highly desirable. In this paper, we propose such a solution from a unique and viable angle.

1.1 Background of SSDs

1.1.1 Flash memory and SSD internals

NAND flash memory is the basic building block of most SSDs on the market. A flash memory package is usually composed of one or multiple *dies* (chips). Each die is segmented into multiple *planes*, and a plane is further divided into thousands (e.g. 2048) of *erase blocks*. An erase block usually consists of 64-128 *pages*. Each page has a data area (e.g. 4KB) and a spare area (a.k.a. metadata area). Flash memories support three major operations. *Read* and *write* (a.k.a. *program*) are performed in units of pages, and *erase*, which clears all the pages in an erase block, must be conducted in erase blocks.

*Currently working at the Intel Labs in Hillsboro, OR.

Flash memory has three critical technical constraints: (1) *No in-place overwrite* – the whole erase block must be erased before writing (programming) any page in this block. (2) *No random writes* – the pages in an erase block must be written sequentially. (3) *Limited erase/program cycles* – an erase block can wear out after a certain number of erase/program cycles (typically 10,000-100,000).

As a critical component in the SSD design, the *Flash Translation Layer* (FTL) is implemented in the SSD controller to emulate a hard disk drive by exposing an array of *logical block addresses* (LBAs) to the host. In order to address the aforesaid three constraints, the FTL designers have developed several sophisticated techniques: (1) *Indirect mapping* – A mapping table is maintained to track the dynamic mapping between logical block addresses (LBAs) and physical block addresses (PBAs). (2) *Log-like write mechanism* – Each write to a logical page only invalidates the previously occupied physical page, and the new content data is appended sequentially in a clean erase block, like a *log*, which is similar to the log-structured file system [41]. (3) *Garbage collection* – A garbage collector (GC) is launched periodically to recycle invalidated physical pages, consolidate the valid pages into a new erase block, and clean the old erase block. (4) *Wear-leveling* – Since writes are often concentrated on a subset of data, which may cause some blocks to wear out earlier than the others, a *wear-leveling* mechanism tracks and shuffles hot/cold data to even out writes in flash memory. (5) *Over-provisioning* – In order to assist garbage collection and wear-leveling, SSD manufacturers usually include a certain amount of over-provisioned spare flash memory space in addition to the host-usable SSD capacity.

1.1.2 The lifespan of SSDs

As flash memory has a limited number of erase/program cycles, the lifespan of SSDs is naturally constrained. In essence, the lifespan of SSDs is a function of three factors: (1) *The amount of incoming write traffic* – The less data written into an SSD, the longer the lifespan would be. In fact, the SSD manufacturers often advise commercial users, whose systems undergo intensive write traffic (e.g. an email server), to purchase more expensive high-end SSDs. (2) *The size of over-provisioned flash space* – A larger over-provisioned flash space provides more available clean flash pages in the allocation pool that can be used without triggering a garbage collection. Aggressive over-provisioning can effectively reduce the average number of writes over all flash pages, which in turn improves the endurance of SSDs. For example, the high-end Intel® X25-E SSD is aggressively over-provisioned with about 8GB flash space, which is 25% of the labeled SSD capacity (32GB) [25]. (3) *The efficiency of garbage collection and wear-leveling mechanisms* – Having been extensively researched, the garbage collection and wear-leveling policies can significantly impact the lifespan of SSDs. For example, *static wear-leveling*, which swaps active blocks with randomly chosen inactive blocks, performs better in endurance than *dynamic wear-leveling*, which only swaps active blocks [13].

Most previous research work [21] focuses on the third factor, garbage collection and wear-leveling policies. A survey [21] summarizes these techniques. In contrast, little study has been conducted on the other two aspects. This may be because incoming write traffic is normally believed to be workload dependent, which cannot be changed at the device level, and the over-provisioning of flash space is designated at the manufacturing process and cannot be excessively large (due to the production cost). In this paper we will show that even at the SSD device level, we can still effectively extend the SSD lifespan by reducing the amount of incoming write traffic and squeezing available flash memory space during runtime, which has not been considered before. This goal can be achieved based on our observation of a widely existing phenomenon – *data duplication*.

1.2 Data Duplication is Common

In file systems data duplication is very common. For example, kernel developers can have multiple versions of Linux source code for different projects. Users can create/delete the same files multiple times. Another example is word editing tools, which often automatically save a copy of documents every few minutes, and the content of these copies can be almost identical.

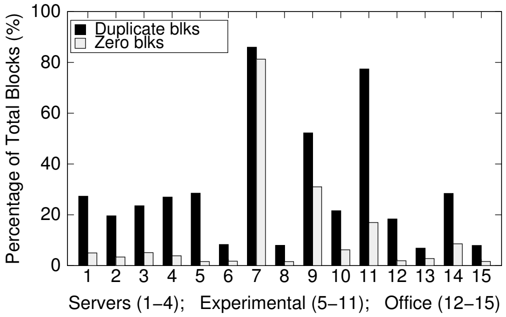

Figure 1: The percentage of redundant data in disks.

To make a case here, we have studied 15 disks installed on 5 machines in the Department of Computer Science and Engineering at the Ohio State University. Three file systems can be found in these disks, namely Ext2, Ext3, and NTFS. The disks are used in different environments, 4 disks from *Database/Web Servers*, 7 disks from *Experimental Systems* for kernel development, and the other 4 disks from *Office Systems*. We slice the disk space into 4KB blocks and use the SHA-1 hash function [1] to calculate a 160-bit hash value for each block. We can identify duplicate blocks by comparing the hash values. Figure 1 shows the *duplication rates* (i.e. the percentage of duplicate blocks in total blocks).

In Figure 1, we find that the duplication rate ranges from 7.9% to 85.9% across the 15 disks. We also find that in only one disk with NTFS, the duplicate blocks are dominated by 'zero' blocks. The duplicate blocks on the other disks are mostly non-zero blocks, which means that these duplicate blocks contain 'meaningful' data. Considering the fact that a typical SSD has an over-provisioned space of only 1-20% of the flash memory space, removing the duplicate data, which accounts for 7.9-85.9% of the SSD capacity, can substantially extend the available flash space that can be used for garbage collection and wear-leveling. If this effort is successful, we can raise the performance comparable to that of high-end SSDs with no need of extra flash space.

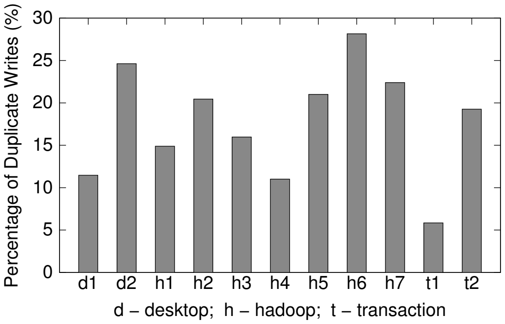

Figure 2: The perc. of duplicate writes in workloads.

Besides the static analysis of the data redundancy in storage, we have also collected I/O traces and analyzed the data accesses of 11 workloads from three categories (see more details in Section 4). For each workload, we modified the Linux kernel by intercepting each I/O request and calculating a hash value for each requested block. We analyzed the I/O traces off-line. Figure 2 shows the percentage of the duplicate writes in each workload. We can find that 5.8-28.1% of the writes are duplicated. This finding suggests that if we remove these duplicate writes, we can effectively reduce the write traffic into flash medium, which directly improves the endurance accordingly, not to mention the indirect effect of reducing the number of extra writes caused by less frequently triggered garbage collections.

1.3 Making FTL Content Aware

Based on the above observations and analysis, we propose a *Content-Aware Flash Translation Layer* (CAFTL) to integrate the functionality of eliminating duplicate writes and redundant data into SSDs to enhance the lifespan at the device level.

CAFTL intercepts incoming write requests at the SSD device level and uses a collision-free cryptographic hash function to generate fingerprints summarizing the content of updated data. By querying a fingerprint store, which maintains the fingerprints of resident data in the SSD, CAFTL can accurately and safely eliminate duplicate writes to flash medium. CAFTL also uses a two-level mapping mechanism to coalesce redundant data, which effectively extends available flash space and improves GC efficiency. In order to minimize the performance impact caused by computing hash values, we have also designed a set of acceleration methods to speed up fingerprinting. With these techniques, CAFTL can effectively reduce write traffic to flash, extend available flash space, while retaining high data access performance.

CAFTL is an augmentation, rather than a complete replacement, to the existing FTL designs. Being *content-aware*, CAFTL is orthogonal to the other FTL policies, such as the well researched garbage collection and wear-leveling policies. In fact, the existing mechanisms in the SSDs provide much needed facilities for CAFTL and make it a perfect fit in the existing SSD architecture. For example, the *indirect mapping mechanism* naturally makes associating multiple logical pages to one physical page easy to implement; the *periodic scanning process* for garbage collection and wear-leveling can also carry out an out-of-line deduplication asynchronously; the *log-like write mechanism* makes it possible to re-validate the 'deleted' data without re-writing the same content; and finally, the *semiconductor nature* of flash memory makes reading randomly remapped data free of high latencies.

CAFTL is also backward compatible and portable. Running at the device level as a part of SSD firmware, CAFTL does not need to change the standard host/device interface for passing any extra information from the upper-level components (e.g. file system) to the device. All of the design of CAFTL is isolated at the device level and hidden from users. This guarantees CAFTL as a drop-in solution, which is highly desirable in practice.

1.4 Our Contributions

We have made the following contributions in this paper: (1) We have studied data duplications in file systems and various workloads, and assessed the viability of improving endurance of SSDs through deduplication. (2) We have carefully designed a content-aware FTL to extend the SSD lifespan by removing duplicate writes (up to 24.2%) and redundant data (up to 31.2%) with minimal overhead. To the best of our knowledge, this is the first study using effective deduplication in SSDs. (3) We have also designed a set of techniques to accelerate the in-line deduplication in SSD devices, which are particularly effective with small on-device buffer spaces (e.g. 2MB) and make performance overhead nearly negligible. (4) We have implemented CAFTL in the DiskSim simulator and comprehensively evaluated its performance and shown the effectiveness of improving the SSD lifespan through extensive trace-driven simulations.

The rest of this paper is organized as follows. In Section 2, we discuss the unique challenges in the design of CAFTL. Section 3 introduces the design of CAFTL and our acceleration methods. We present our performance evaluation in Section 4. Section 5 gives the related work. The last section discusses and concludes this paper.

2 Technical Challenges

CAFTL shares the same principle of removing data redundancy with Content-Addressable Storage (CAS), e.g. [11,24,30,45,47], which is designed for backup/archival systems. However, we cannot simply borrow CAS policies in our design due to four unique and unaddressed challenges: (1) *Limited resources* – CAFTL is designed for running in an SSD device with limited memory space and computing power, rather than running on a dedicated powerful enterprise server. (2) *Relatively lower redundancy* – CAFTL mostly handles regular file system workloads, which have an impressive but much lower duplication rate than that of backup streams with high redundancy (often 10 times or even higher). (3) *Lack of semantic hints* – CAFTL works at the device level and only sees a sequence of logical blocks without any semantic hints from host file systems. (4) *Low overhead requirement* – CAFTL must retain high data access performance for regular workloads, while this is a less stringent requirement in backup systems that can run during out-of-office hours.

All of these unique requirements make deduplication particularly challenging in SSDs and it requires non-trivial efforts to address them in the CAFTL design.

3 The Design of CAFTL

The design of CAFTL aims to reach the following three critical objectives.

- *Reducing unnecessary write traffic* – By examining the data of incoming write requests, we can detect and remove duplicate writes in-line, so that we can effectively filter unnecessary writes into flash memory and directly improve the lifespan of SSDs.
- *Extending available flash space* – By leveraging the indirect mapping framework in SSDs, we can map logical pages sharing the same content to the same physical page. The saved space can be used for GC and wear-leveling, which indirectly improves the lifespan.
- *Retaining access performance* – A critical requirement to make CAFTL truly effective in practice is to avoid significant negative performance impacts. We must minimize runtime overhead and retain high data access performance.

3.1 Overview of CAFTL

CAFTL eliminates duplicate writes and redundant data through a combination of both *in-line* and *out-of-line* (a.k.a post-processing or out-of-band) deduplication. In-line deduplication refers to the case where CAFTL proactively examines the incoming data and cancels duplicate writes before committing a write request to flash. As a 'best-effort' solution, CAFTL does not guarantee that all duplicate writes can be examined and removed immediately (e.g. it can be disabled for performance purposes). Thus CAFTL also periodically scans the flash memory and coalesces redundant data out of line.

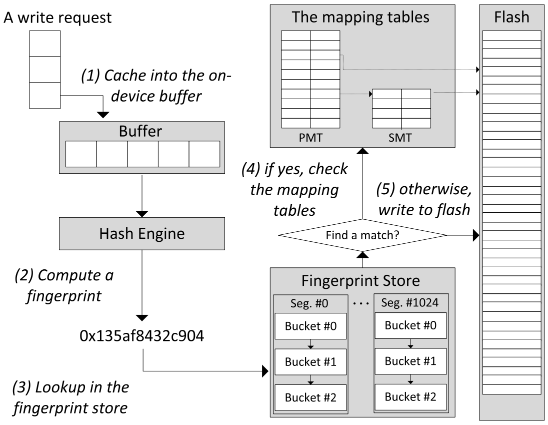

Figure 3: An illustration of CAFTL architecture.

Figure 3 illustrates the process of handling a write request in CAFTL – When a write request is received at the SSD, (1) the incoming data is first temporarily maintained in the *on-device buffer*; (2) each updated page in the buffer is later computed a hash value, also called *fingerprint*, by a *hash engine*, which can be a dedicated processor or simply a part of the controller logic; (3) each fingerprint is looked up against a *fingerprint store*, which maintains the fingerprints of data already stored in the flash memory; (4) if a match is found, which means that a residing data unit holds the same content, the *mapping tables*, which translate the host-viewable logical addresses to the physical flash addresses, are updated by mapping it to the physical location of the residing data, and correspondingly the write to flash is canceled; (5) if no match is found, the write is performed to the flash memory as a regular write.

3.2 Hashing and Fingerprint Store

CAFTL attempts to identify and remove duplicate writes and redundant data. A byte-by-byte comparison is excessively slow. A common practice is to use a cryptographic hash function, e.g. SHA-1 [1] or MD5 [40], to compute a collision-free hash value as a fingerprint. Duplicate data can be determined by comparing fingerprints. Here we explain how we produce and manage fingerprints.

3.2.1 Choosing hashing units

CAFTL uses a chunk-based deduplication approach. Unlike most CAS systems, which often use more complicated *variable-sized* chunking, CAFTL adopts a *fixed-sized* chunking approach for two reasons. First, the variable-sized chunking is designed for segmenting a long I/O stream. In CAFTL, we handle a sequence of individual requests, whose size can be very small (a few kilobytes) and vary significantly. Thus variable-sized chunking is inappropriate for CAFTL. Second, the basic operation unit in flash is a page (e.g. 4KB), and the internal management policies in SSDs, such as the mapping policy, are also designed in units of pages. Thus, using pages as the fixed-sized chunks for hashing is a natural choice and also avoids unnecessary complexity.

3.2.2 Hash function and fingerprints

In order to identify duplicate data, a collision-free hash function is used for summarizing the content of pages. We use the SHA-1 [1], a widely used cryptographic hash function, and rely on its collision-resistant properties to index and compare pages. For each page, we calculate a 160-bit hash value as its *fingerprint* and store it as the page's metadata in flash. The SHA-1 hash function has been proven computationally infeasible to find two distinct inputs hashing to the same value [32]. We can safely determine if two pages are identical using fingerprints.

3.2.3 The fingerprint store

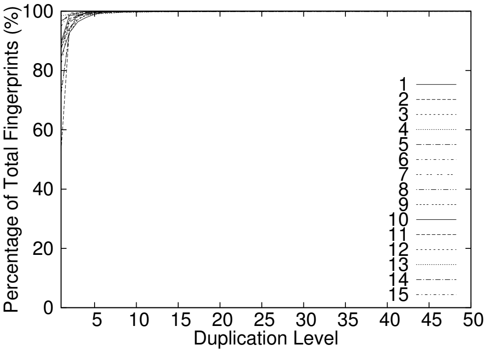

Figure 4: The CDF figure of duplicate fingerprints.

In order to locate quickly the physical page with a specific fingerprint, CAFTL manages an in-memory structure, called *Fingerprint Store*. Apparently, keeping all fingerprints and related information (25 bytes each) in memory is too costly and unnecessary. We have studied the distribution of fingerprints in the 15 disks and we plot a Cumulative Distribution Function (CDF) figure in Figure 4. We can see that the distribution of duplicated fingerprints is skewed – only 10-20% of the fingerprints are highly duplicated (more than 2). This finding provides two implications. First, most fingerprints are unique and never have a chance to match any queried fingerprint. Second, a complete search in the fingerprint store would incur high lookup latencies, and even worse, most lookups eventually turn out to be useless (no match found). Thus, we should only store and search in the most likely-to-be-duplicated fingerprints in memory.

We first logically partition the hash value space into N segments. For a given fingerprint, f, we can map it to segment $(f \ mod \ N)$, and the random nature of the hash function guarantees an even distribution of fingerprints among the segments. Each segment contains a list of *buckets*. Each bucket is a 4KB page in memory and consists of multiple entries, each of which is a key-value pair, $\{$*fingerprint*, (*location*, *reference*)$\}$. The 160-bit *fingerprint* indexes the entry; the 32-bit *location* denotes where we can find the data, either the PBA of a physical flash page or the VBA of a secondary mapping entry (see Section 3.3); the 8-bit *reference* denotes the hotness of this fingerprint (i.e. the number of referencing logical pages). The entries in each bucket are sorted in the ascending order of their fingerprint values to facilitate a fast in-bucket binary search. The total numbers of buckets and segments are designated by the SSD manufacturers.

The fingerprint store maintains the most highly referenced fingerprints in memory. During the SSD startup time, after the mapping tables are built up (to be discussed in Section 3.3), the fingerprint store is also reconstructed by scanning the mapping tables and the metadata in flash to load the key value pairs of $\{$*fingerprint*, (*location*, *reference*)$\}$ into memory. Initially no bucket is allocated in the fingerprint store. Upon inserting a fingerprint, an empty bucket is allocated and linked into a bucket list of the corresponding segment. This bucket holds the fingerprints inserted into the corresponding segment until the bucket is filled up, then we allocate another bucket. We continue to allocate buckets in this way until there are no more free buckets available. If that happens, the newly inserted fingerprint will replace the fingerprint with the smallest reference counter (i.e. the coldest one) in the bucket, unless its reference counter is smaller than any of the resident fingerprints. Note that we choose the inserting bucket in a round-robin manner to ensure a relatively even distribution of hot/cold fingerprints across the buckets in a segment. It is also worth mentioning here that a 8-bit reference counter is sufficiently large for distinguishing the hot fingerprints, because most fingerprints have a reference counter smaller than 255 (see Figure 4). We consider fingerprints with a reference counter larger than 255 as highly referenced and do not further distinguish their difference in hotness. In this way, we can include the most highly referenced fingerprints in memory. Although we may miss some opportunities of identifying the duplicates whose fingerprints are not resident in memory, this probability is considered low (as shown in Figure 4), and we are not pursuing a perfect in-line deduplication. Our out-of-line scanning can still identify these duplicates later.

Searching a fingerprint can be very simple. We compute the mapping segment number and scan the corresponding list of buckets one by one. In each bucket, we use binary search to speed up the in-bucket lookup. However, for a segment with a large set of buckets, this method is still improvable. We have designed three optimization techniques to further accelerate fingerprint lookups. (1) *Range Check* – before performing the binary search in a bucket, we first compare the fingerprint with the smallest and the largest fingerprints in the buckets. If the fingerprint is out of the range, we quickly skip over this bucket. (2) *Hotness-based Reorganization* – the fingerprints in the linked buckets can be reorganized in the descending order of their reference counters. This moves the hot fingerprints closer to the list head and potentially reduces the number of the scanned buckets. (3) *Bucket-level Binary Search* – the fingerprints across the buckets can be reorganized in the ascending order of the fingerprint values by using a merge sort. For each segment we maintain an array of pointers to the buckets in the list. We can perform a binary search at the bucket level by recursively selecting the bucket in the middle to do a Range Check. In this way we can quickly locate the target bucket and skip over most buckets. Although reorganizing the fingerprints requires performing an additional merge sort, our experiments show that these optimizations can significantly reduce the number of comparisons of fingerprint values. In Section 4.3.3 we will show and compare the effectiveness of the three techniques.

3.3 Indirect Mapping

Indirect mapping is a core mechanism in the SSD architecture. SSDs expose an array of logical block addresses (LBAs) to the host, and internally, a *mapping table* is maintained to track the physical block address (PBA) to which each LBA is mapped. For CAFTL, the existing indirect mapping mechanism in SSDs provides a basic framework for deduplication and avoids rebuilding the whole infrastructure from scratch.

On the other hand, the existing 1-to-1 mapping mechanism in SSDs cannot be directly used for CAFTL, which is essentially *N*-to-1 mapping, because of two new challenges. (1) When a physical page is relocated to another place (e.g. in garbage collection), we must be able to identify quickly all the logical pages mapped to this physical page and update their mapping entries to point to the new location. (2) Since a physical page could be shared by multiple logical pages, it cannot be recycled by the garbage collector until all the referencing logical pages are demapped from it, which means that we must track the number of referencing logical pages.

3.3.1 Two-level indirect mapping

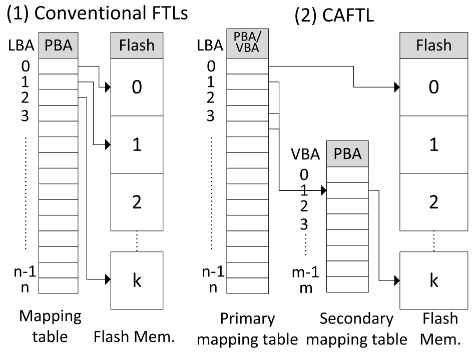

Figure 5: An illustration of the indirect mapping.

We have designed a new indirect mapping mechanism to address these aforementioned issues. As shown in Figure 5, a conventional FTL uses a one-level indirect mapping, from LBAs to PBAs. In CAFTL, we create another indirect mapping level, called *Virtual Block Addresses* (VBAs). A VBA is essentially a pseudo address name to represent a set of LBAs mapped to the same PBA. In this two-level indirect mapping structure, we can locate the physical page for a logical page either through LBA→PBA or LBA→VBA→PBA.

We maintain a *primary mapping table* and a *secondary mapping table* in memory. The *primary mapping table* maps a LBA to either a PBA, if the logical page is unique, or a VBA, if it is a duplicate page. We differentiate PBAs and VBAs by using the most significant bit in the 32-bit page address. For a page size of 4KB, using the remaining 31 bits can address 8,192 GB storage space, which is sufficiently large for an SSD. The *secondary mapping table* maps a VBA to a PBA. Each entry is indexed by the *VBA* and has two fields, $\{PBA, reference\}$. The 32-bit *PBA* denotes the physical flash page, and the 32-bit *reference* tracks the exact number of logical pages mapped to the physical page. Only physical pages without any reference can be recycled for garbage collection.

This two-level indirect mapping mechanism has several merits. First, it significantly simplifies the reverse updates to the mapping of duplicate logical pages. When relocating a physical page during GC, we can use its associated VBA to quickly locate and update the secondary mapping table by mapping the VBA to the new location (PBA), which avoids exhaustively searching for all the referencing LBAs in the huge primary mapping table. Second, the secondary mapping table can be very small. Since CAFTL handles regular file system workloads, most logical pages are unique and directly mapped through the primary table. We can also apply an approach similar to DFTL [23] to further reduce the memory demand by selectively maintaining the most frequently accessed entries of the mapping tables in memory. Finally, this incurs minimal additional lookup overhead. For unique pages, it performs identically to conventional FTLs; for duplicate pages, only one extra memory access is needed for the lookup operation.

3.3.2 The mapping tables in flash

The mapping relationship is also maintained in flash memory. We keep an in-flash copy of the primary and secondary mapping tables along with a *journal* in dedicated flash space in SSD. Both in-flash structures are organized as a list of linked physical flash pages. When updating the in-memory tables (e.g. remapping a LBA to a new location), the update record is logged into a small in-memory buffer. When the buffer is filled, the log records are appended to the in-flash journal. If power failure happens, a capacitor (e.g. a SuperCap [46]) can provide sufficient current to flush the unwritten logs into the journal and secure the critical mapping structures in persistent storage. Periodically the in-memory tables are synced into flash and the journal is reinitialized. During the startup time, the in-flash tables are first loaded into memory and the logged updates in the journal are applied to reconstruct the mapping tables.

3.3.3 The metadata pages in flash

Unlike much prior work, which writes the metadata (e.g. LBA and fingerprint) in the spare area of physical flash pages, we reserve a dedicated number of flash pages, also called *metadata pages*, to store the metadata, and keep a *metadata page array* for tracking PBAs of the metadata pages. The spare area of a physical page is only used for storing the Error Correction Code (ECC) checksum. If each physical page is associated with 24 bytes of metadata (a 160-bit fingerprint and a 32-bit LBA/VBA), for a 32GB SSD with 4KB flash pages, we need about 0.6% of the flash space for storing metadata and a 192KB metadata page array. In this way, we can detach the data pages and the metadata pages, which allows us to manage flexibly the metadata for physical flash pages.

3.4 Acceleration Methods

Fingerprinting is the key bottleneck of the in-line deduplication in CAFTL, especially when the on-device buffer size is limited. Here we present three effective techniques to reduce its negative performance impact.

3.4.1 Sampling for hashing

In file system workloads, as we discussed previously, duplicate writes are not a ‘common case’ as in backup systems. This means that most time we spend on fingerprinting is not useful at all. Thus, we selectively pick only one page as a *sample page* for fingerprinting, and we use this sample fingerprint to query the fingerprint store to see if we can find a match there. If this is true, the whole write request is very likely to be a duplicate, and we can further compute fingerprints for the other pages to confirm that. Otherwise, we assume the whole request would not be a duplicate and abort fingerprinting at the earliest time. In this way, we can significantly reduce the hashing cost.

The key issue here is which page should be chosen as the sample page. It is particularly challenging in CAFTL, since CAFTL only sees a sequence of blocks and cannot leverage any file-level semantic hints (e.g. [11]). We propose to use *Content-based Sampling* – We select the first four bytes, called *sample bytes*, from each page in a request, and we concatenate the four bytes into a 32-bit numeric value. We compare these values and the page with the *largest* value is the sample page. The rationale behind this is that if two requests carry similar content, the pages with the largest sample bytes in two requests would be very likely to be the same, too. We deliberately avoid selecting the sample pages based on hash values (e.g. [11, 30]), because in CAFTL, hashing itself incurs high latency. Thus relying on hash values for sampling is undesirable, so we directly pick sample pages based on their *unprocessed* content data. We have also examined choosing other bytes (Figure 6) as the sample bytes and found that using the first four bytes performs constantly well across different workloads.

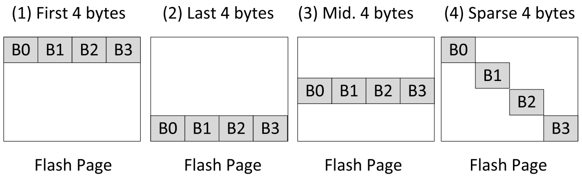

Figure 6: An illustration of four choices of sample bytes.

In our implementation of sampling, we divide the sequence of pages in a write request into several *sampling units* (e.g. 32 pages), and we pick one sample page from each unit. We also note that sampling could affect deduplication – the larger a sampling unit is, the better performance but the lower deduplication rate would be. We will study the effect of unit sizes in Section 4.4.1.

3.4.2 Light-weight pre-hashing

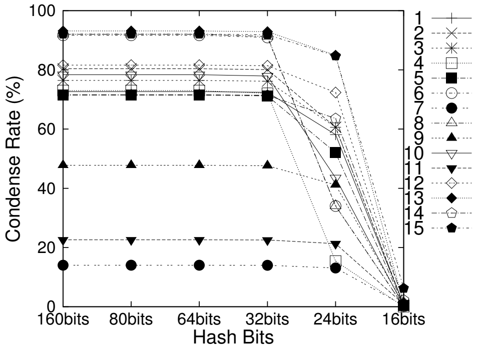

Figure 7: Condense rates vs. hash bits.

Computing a light-weight hash function often incurs lower computational cost. For example, producing a 32-bit CRC32 hash value is over 10 times faster than computing a 160-bit SHA-1 hash value. More importantly, our study shows that reducing the hash strength would *not* incur a significant increase of false positives for a typical SSD capacity. We can see in Figure 7 that using only 32 bits can achieve nearly the same condense rate as using 160 bits. Plus, many SSDs integrate a dedicated ECC engine to compute checksum and detect errors, which can also be leveraged to speed up hashing.

We propose a technique, called *light-weight prehashing*. We maintain an extra 32-bit CRC32 hash value for each fingerprint in the fingerprint store. For a page, we first compute a CRC32 hash value and query the fingerprint store. If a match is found, which means the page is very likely to be a duplicate, then we use the SHA-1 hash function to generate a fingerprint and confirm it in the fingerprint store; otherwise, we abort the high-cost SHA-1 fingerprinting immediately and perform the write to flash. Although maintaining CRC32 hash values demands more fingerprint store space, the significant performance benefit well justifies it, as shown in Section 4.4.2. We have also considered using a Bloom filter [12] for pre-screening, like in the DataDomain® file system [47], but found it inapplicable to CAFTL, because it requires multiple hashings and the summary vector cannot be updated when a fingerprint is removed.

3.4.3 Dynamic switches

In some extreme cases, incoming requests may wait for available buffer space to be released by previous requests. CAFTL provides *dynamic switch* as the last line of defense for performance protection in such cases.

We set a *high watermark* and a *low watermark* to turn the in-line deduplication off and on, respectively. If the percentage of the occupied cache space hits a high watermark (95%), we disable the in-line deduplication to flush writes quickly to flash and release buffer space. Once the low watermark (50%) is hit, we re-enable the in-line deduplication. Although this remedy solution would reduce the deduplication rate, we still can perform out-of-line deduplication at a later time, so it is an acceptable tradeoff for retaining high performance.

3.5 Out-of-line Deduplication

As mentioned previously, CAFTL does not pursue a perfect in-line deduplication, and an internal routine is periodically launched to perform *out-of-line fingerprinting* and *out-of-line deduplication* during the device idle time.

Out-of-line fingerprinting is simple. We scan the metadata page array (Section 3.3.3) to find physical pages not yet fingerprinted. If one such a page is found, we read the page out, compute the fingerprint, and update its metadata. To avoid unnecessarily scanning the metadata of pages already fingerprinted, we use one bit in an entry of the metadata page array to denote if all of the fingerprints in the corresponding metadata page have already been computed, and we skip over such pages.

Out-of-line deduplication is more complicated due to the memory space constraint. We adopt a solution similar to the widely used *external merge sort* [39] in database systems. Supposing we have M fingerprints in total and the available memory space can accommodate N fingerprints, where $M > N$. We scan the metadata page array from the beginning, each time N fingerprints are loaded and sorted in memory, and temporarily stored in flash, then we load and sort the next N fingerprints, and so on. This process is repeated for K times ($K = \lceil \frac{M}{N} \rceil$) until all the fingerprints are processed. Then we can merge sort these K blocks of fingerprints in memory and identify the duplicate fingerprints.

Out-of-line fingerprinting and deduplication can be performed together with the GC process or independently. Since there is no harm in leaving duplicate or unfingerprinted pages in flash, these operations can be performed during idle period and immediately aborted upon incoming requests, and the perceivable performance impact to foreground jobs is minimal.

4 Performance Evaluation

4.1 Experimental Systems

We have implemented and evaluated our design of CAFTL based on a comprehensive trace-driven simulation. In this section we will introduce our simulator, trace collection, and system configurations.

4.1.1 SSD Simulator

CAFTL is a device-level design running in the SSD controller. We have implemented it in a sophisticated SSD simulator based on the Microsoft® Research SSD extension [5] for the *DiskSim* simulation environment [14]. This extension was also used in prior work [6].

The Microsoft extension is well modularized and implements the major components of FTL, such as the indirect mapping, garbage collection and wear-leveling policies, and others. Since the current version lacks an on-device buffer, which is becoming a standard component in recent generations of SSDs, we augmented the current implementation and included a shared buffer for handling incoming read and write requests. When a write request is received at the SSD, it is first buffered in the cache, and the SSD immediately reports completion to the host. Data processing and flash operations are conducted asynchronously in the background [16]. A read request returns back to the host once the data is loaded from flash into the buffer. We should note that this simulator follows a general FTL design [6], and the actual implementations of the SSD on the market can have other specific features.

4.1.2 SSD Configurations

Description	Configuration
Flash Page Size	4KB
Pages per Block	64
Blocks per Plane	2048
Planes per Package	8
# of Packages	10
Mapping policy	Full striping
Over-provisioning	15%
Garbage Collection Threshold	5%

Table 1: Configurations of the SSD simulator.

In our experiments, we use the default configurations from the SSD extension, unless denoted otherwise. Table 1 gives a list of the major config parameters.

Description	Latency
Flash Read/Write/Erase	25 µs/200µs/1.5ms
SHA-1 hashing (4KB)	47,548 cycles
CRC32 hashing (4KB)	4,120 cycles

Table 2: Latencies configured in the SSD simulator.

Table 2 gives the parameters of latencies used in our experiments. For the flash memory, we use the default latencies in our experiments. For the hashing latencies, we first cross compile the hash function code to the ARM® platform and run it on the *SimpleScalar-ARM* simulator [4] to extract the total number of cycles for executing a hash function. We assume a processor similar to ARM® Cortex R4 [8] on the device, which is specifically designed for high-performance embedded devices, including storage. Based on its datasheet, the ARM processor has a frequency from 304MHz to 934MHz [8], and we can estimate the latency for hashing a 4KB page by dividing the number of cycles by the processor frequency. It is also worth mentioning here that according to our communications with SSD manufacturer [3], high-frequency (600+ MHz) processors, such as the Cortex processor, are becoming increasingly normal in high-speed storage devices. Leveraging such abundant computing power on storage devices can be a research topic for further investigation.

4.1.3 Workloads and trace collection

We have selected 11 workloads from three representative categories and collected their data access traces.

- *Desktop* (d1,d2) – Typical office workloads, e.g. Internet surfing, emailing, word editing, etc. The workloads run for 12 and 19 hours, respectively, and feature irregular idle intervals and small reads and writes.
- *Hadoop* (h1-h7)– We execute seven TPC-H data warehouse queries (Query 1,6,11,14,15,16,20) with scale factor of 1 on a Hadoop distributed system platform [2]. These workloads run for 2-40 minutes and generate intensive large writes of temp data.
- *Transaction* (t1,t2) – We execute TPC-C workloads (1-3 warehouses, 10 terminals) for transaction processing on PostgreSQL 8.4.3 database system. The two workloads run for 30 minutes and 4 hours, respectively, and feature intensive write operations.

The traces are collected on a DELL® Dimension 3100 workstation with an Intel® Pentium™4 3.0GHz processor, a 3GB main memory, and a 160GB 7,200 RPM Seagate® hard disk drive. We use Ubuntu 9.10 with the Ext3 file system. We modified the Linux kernel 2.6.32 source code to intercept each I/O request and compute a SHA-1 hash value as a fingerprint for each 4KB page of the request. These fingerprints, together with other request information (e.g. offset, type), are transferred to another machine via *netconsole* [35]. This avoids the possible interference caused by tracing. The collected trace files are analyzed offline and used to drive the simulator for our experimental evaluation.

4.2 Effectiveness of Deduplication

CAFTL intends to remove duplicate writes and extend flash space. In this section, we perform two sets of experiments to show the effectiveness of deduplication in CAFTL. In both experiments, we use an SSD with a 934MHz processor and a 16MB buffer.

4.2.1 Removing duplicate writes

CAFTL identifies and removes duplicate writes via inline deduplication. Denoting the total number of pages requested to be written as n, and the total number of pages being actually written into flash medium as m, the *deduplication rate* is defined as $\frac{n-m}{n}$. Figure 8 shows the deduplication rate of the 11 workloads running on CAFTL. In this figure, *offline* refers to the optimal case, where the traces are examined and deduplicated offline. We also show CAFTL without sampling and with a sampling unit size of 128KB (32 pages), denoted as *no-sampling* and *128KB*, respectively.

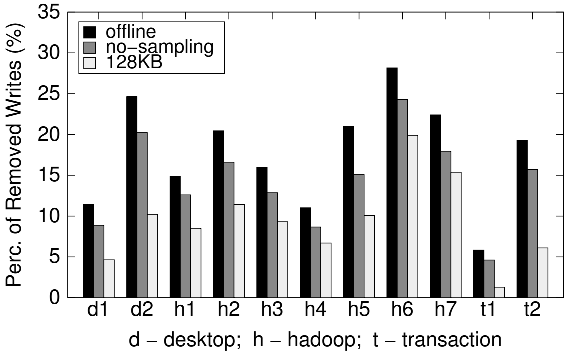

Figure 8: Perc. of removed duplicate writes.

As we see in Figure 8, duplication is highly workload dependent. Across the 11 workloads, the rate of duplicate writes in the workloads ranges from 5.8% (*t1*) to 28.1% (*h6*). CAFTL can achieve deduplication rates from 4.6% (*t1*) to 24.2% (*h6*) with no sampling. Compared with the optimal case (*offline*), CAFTL identifies up to 86.2% of the duplicate writes in *offline*. We also can see that with a larger sampling unit (128KB), CAFTL achieves a lower but reasonable deduplication rate. In Section 4.4.1, we will give more detailed analysis on the effect of sampling unit sizes.

4.2.2 Extending flash space

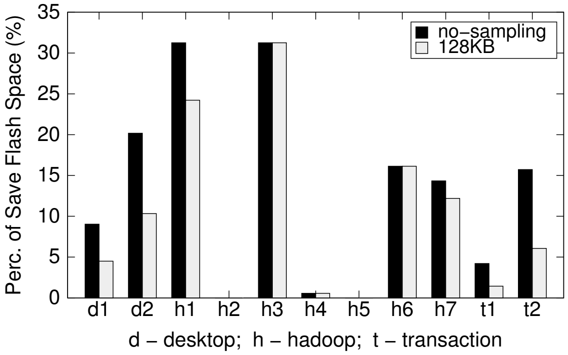

Figure 9: Perc. of extended flash space.

Besides directly removing duplicate writes to the flash memory, CAFTL also reduces the amount of occupied flash memory space and increases the number of available clean erase blocks for garbage collection and wear-leveling. Figure 9 shows the percentage of extended flash space in units of erase blocks, compared to the baseline case (without CAFTL). We show CAFTL without sampling (*no-sampling*) and with sampling (*128KB*).

As shown in Figure 9, CAFTL can save up to 31.2% (*h1*) of the occupied flash blocks for the 11 workloads. The worst cases are *h2* and *h5*, in which no space saving is observed. This is because the two workloads are relatively smaller, the total number of occupied erase blocks is only 176. Although the number of pages being written is reduced by 16.6% (*h2*) and 15% (*h5*), the saved space in units of erase blocks is very small.

4.3 Performance Impact

To make CAFTL truly effective in practice, we must retain high performance and minimize negative impact. Here we study three key factors affecting performance, *cache size*, *hashing speed*, and *fingerprint searching*. The acceleration methods are not applied in experiments.

4.3.1 Cache size

In Figure 10, we show the percentage of the increase of average read/write latencies with various cache sizes (2MB to 16MB). We compare CAFTL with the baseline case (without CAFTL). In the experiments, we configure an SSD with a 934MHz processor. We can see that with a small cache space (2MB), the read and write latencies can increase by a factor of up to 34% (*t1*). With a moderate cache size (8MB), the latency increases are reduced to less than 4.5%. With a 16MB cache, a rather standard size, the latency increases become negligible (less than 0.5%). For some workloads (*d2, h3, h5, h7, t1, t2*), we can even see a slight performance improvement (0.2-0.5%), because CAFTL removes unnecessary writes, which reduces the probability of being blocked by an in-progress flash write operation. In this case we see a negative performance impact with a small cache space, and we will show how to mitigate such a problem through our acceleration methods in Section 4.4.

4.3.2 Hashing speed

Computing fingerprints is time consuming and affects access performance. The hashing speed depends on the capability of processors. Using a more powerful processor can effectively reduce the latency for digesting pages and generating fingerprints. To study the performance impact caused by hashing speed, we vary the processor frequency from 304MHz to 934MHz, based on the Cortex datasheet [8]. We configure an SSD with a 16MB cache space and show the increase of read latencies compared to the baseline case (without CAFTL) in Figure 11. We did not observe an increase of write latencies, since most writes are absorbed in the buffer.

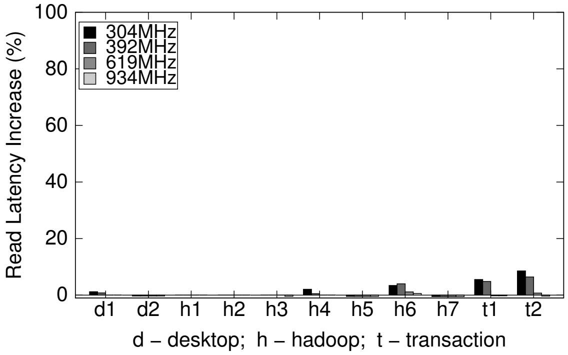

Figure 11: Perf. impact of hashing Speeds.

In Figure 11, we can see that most workloads are insensitive to hashing speed. With a 304MHz processor, the performance overhead is less than 8.5% (*t2*), which has more intensive larger writes. At 934MHz, the performance overhead is merely observable (up to 0.5%). There are two reasons. First, the 16MB on-device buffer absorbs most incoming writes and provides a sufficient space for accommodating incoming reads. Second, the incoming read requests are given a higher priority than writes, which reduces noticeable delays in the critical path. These optimizations make reads insensitive to hashing speed and reduces noticeable latencies. Also

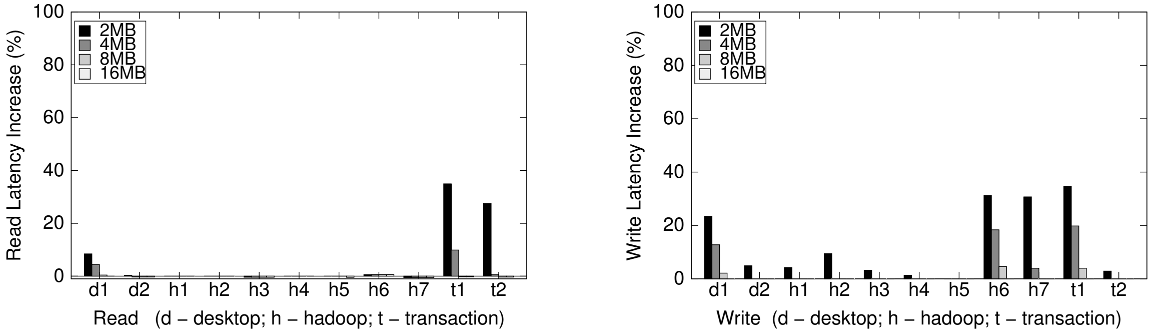

Figure 10: Performance impact of cache sizes (2-16MB).

note that if a dedicated hashing engine is used on the device, the hashing latency could be further reduced.

4.3.3 Fingerprint searching

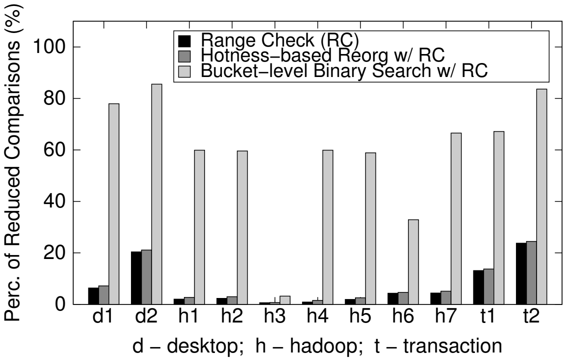

Figure 12: Optimizations on fingerprint searching.

We have proposed three techniques to accelerate fingerprint searching. Figure 12 shows the percentage of reduced fingerprint comparisons compared with the baseline case. We configure the fingerprint store with 256 segments to hold the fingerprints for each workload. We can see that using *Range Check* can effectively reduce the comparisons of fingerprints by up to 23.7% (*t2*). However, *Hotness-based Reorganization* can provide little further improvement (less than 1%), because it essentially accelerates lookups for fingerprints that are duplicated, which is relatively an uncommon case. As expected, *Bucket-level Binary Search* can significantly reduce the average number of comparisons for each lookup. In *d2*, for example, *Bucket-level Binary Search* can effectively reduce the average number of comparisons by a factor of 85.5%. Thus we would suggest applying *Bucket-level Binary Search* and *Range Check* to speed up fingerprint lookups.

4.4 Acceleration Methods

With a small on-device buffer, the high computational latency caused by hashing could be significant and perceived by the users. We have developed three techniques to accelerate fingerprinting. In this section, we will show the effectiveness of each individual technique and then show the effects in aggregate. We configure an SSD with a 934MHz processor and a small 2MB buffer.

4.4.1 Sampling

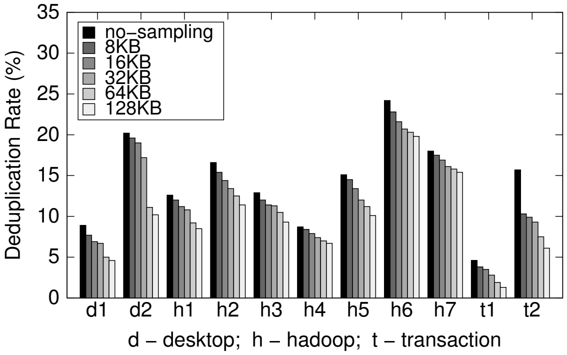

Figure 14: Dedup. with Sampling

As shown in Figure 13 and Figure 14, sampling can significantly improve performance. With the increase of sampling unit size, fewer fingerprints need to be calculated, which translates into a manifold reduction of observed read and write latencies. For example, *h7* achieves a speedup by a factor of 94.1 times for reads and 3.5 times for writes, because of the significantly reduced waiting time for the buffer. Meanwhile, the deduplication rate is only reduced from 18% to 15.4%. Considering such a significant speedup, the minor loss of deduplication rate is acceptable. The maximum speedup, 110.6 times (read), is observed in *t1*, and its deduplication rate drops from 4.6% to 1.3%. This is mostly because for workloads with low duplication rate, the probability of sampling right pages is also relatively low.

4.4.2 Light-weight pre-hashing

Light-weight pre-hashing uses a fast CRC32 hash function to filter most unlikely-to-be-duplicated pages before performing high-cost fingerprinting. Figure 15 shows the speedup of reads and writes by using CRC32 for pre-hashing, compared with CAFTL without pre-hashing. Only pre-hashing is enabled here. We can see that in the best case (*t1*), pre-hashing can reduce the latencies by a factor of up to 148.3 times for reads and 3.9 times for writes. This is because, as mentioned previously,

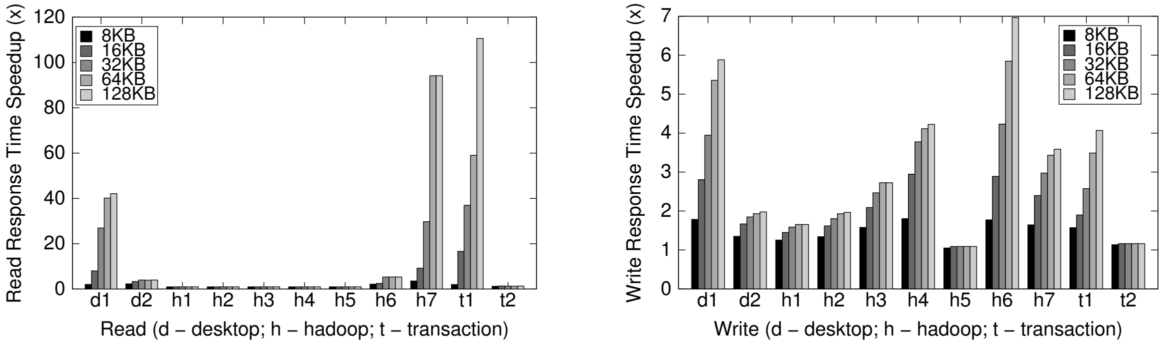

Figure 13: Performance speedup with Sampling (unit size: 8-128KB).

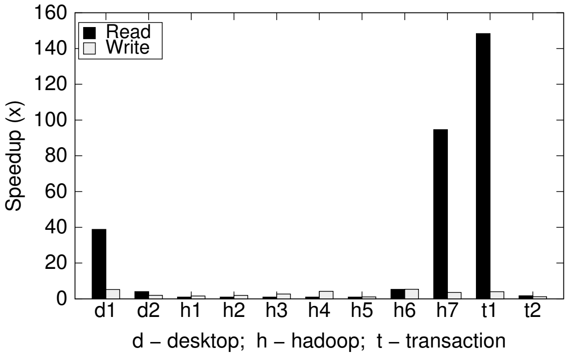

Figure 15: Speedup with pre-hashing.

this workload is write intensive and has a long waiting queue, which makes the queuing effect particularly significant. Similar to sampling, writes receive relatively smaller benefit, because the buffer absorbs the writes with low latency and diminishes the effect of speeding up writes. Meanwhile, we also found negligible difference in deduplication rates, which is consistent with our analysis shown in Figure 7.

4.4.3 Dynamic switch

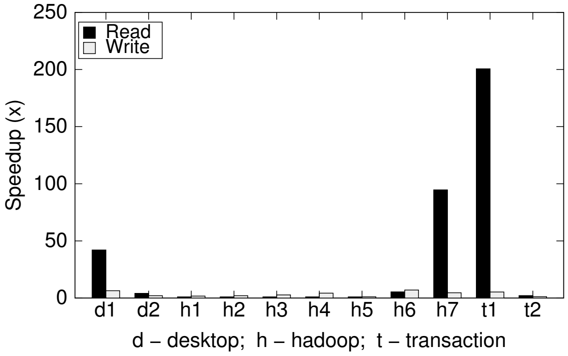

Figure 16: Speedup with dynamic switch.

CAFTL also provides *dynamic switch* to dynamically turn on/off the in-line deduplication, depending on the usage of the on-device buffer. We configure the high watermark as 95% (off) and the low watermark as 50% (on). Figure 16 shows the speedup of reads and writes in the workloads. Again, *t1* receives the most significant performance speedup by a factor of 200.6 times. Some workloads (*h1*-*h5*) receive no benefits, because they are less I/O intensive. For the other workloads, we can observe a speedup of 2.1 times to 94.6 times.

4.4.4 Putting it all together

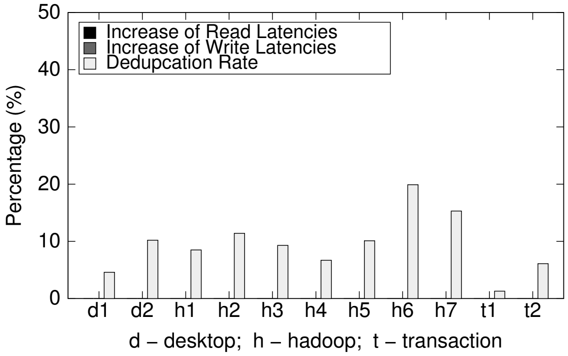

Figure 17: Three acceleration tech. combined

In Figure 17, we enable all the three acceleration techniques and show the increase of read and write latencies, compared with the baseline case (without CAFTL), and the corresponding deduplication rate. We can see that by combining all the three techniques, we can almost completely remove the performance overhead with only a 2MB on-device buffer. In the meantime, we can achieve a deduplication rate of up to 19.9%.

5 Other Related Work

Flash memory based SSDs have received a lot of interest in both academia and industry. There is a large body of research work on flash memory and SSDs (e.g. [6,9,13,15–18,20,23,26–29,31,34,37,38,42,44]). Concerning lifespan issues, most early work focuses on designing garbage collection and wear-leveling policies. A survey [21] summarizes these techniques. Here we only present the papers most related to this work.

Recently Grupp et al. [22] have presented an empirical study on the performance, power, and reliability of flash memories. Their results show that flash memories, particularly MLC devices, exhibit significant error rates after or even before reaching the rated lifetime, which makes using high density SSDs in commercial systems a difficult choice. Another report [13] has studied the write endurance of USB flash drives with a more optimistic conclusion that the endurance of flash memory chips is better than expected, but whole-device endurance is closely related to the FTL designs. A modeling based study on the endurance issues has also been presented in [33]. These studies provide much needed information about the lifespan of flash memory and small-size flash devices. However, so far the endurance of state-of-the-art SSDs has not yet been proven in the field [10].

Early studies on SSDs mainly focus on performance. Some recent studies have begun to look at reliability issues. Differential RAID [9] tries to improve reliability of an SSD-based RAID storage by distributing parity unevenly across SSDs to reduce the probability of correlated multi-device failure. Griffin [42] extends SSD lifetime by maintaining a log-structured HDD cache and migrating cached data periodically. A recent work [36] considers write cycles in addition to storage space as a constrained resource in depletable storage systems and suggests attribute depletion to users in systems like cloud computing. ChunkStash [19] uses flash memory to speed up index lookups for inline storage deduplication. Another work [43] proposes to integrate phase change memory into SSDs to improve the performance, energy consumption, and also lifetime. Our study has made its unique contributions to enhancing the lifespan of SSDs by removing duplicate writes and coalescing redundant data at the device level, as a more general solution.

6 Conclusion and Discussions

Enhancing the SSD lifespan is crucial to a wide deployment of SSDs in commercial systems. In this paper, we have proposed a solution, called CAFTL, and shown that by removing duplicate writes and coalescing redundant data, we can effectively enhance the lifespan of SSDs while retaining high data access performance.

A potential concern about CAFTL is the volatility of the on-device RAM buffer – the buffered data could be lost upon power failure. However, this concern is not new to SSDs. A hard disk drive also has an on-device buffer, but it provides users an option (e.g. using *sdparm* tool) to flexibly enable/disable the buffer on their needs. Similarly, if needed, the users can choose to disable the in-line deduplication and the buffer in an SSD, and the out-of-line deduplication can still be effective.

Although we have striven to minimize memory usage, CAFTL demands more space for storing fingerprints and the secondary mapping table, compared with traditional FTLs. According to our communications with SSD manufacturer [3], memory actually only accounts for a small percentage of the total production cost, and the most expensive component is flash memory. Thus we consider this tradeoff is worthwhile to extend available flash space, and SSD lifespan. If budget allows, we would suggest maintaining the fingerprint store fully in memory, which not only improves deduplication rate but also simplifies designs.

Further improvements are also possible. One is to relax the stringent "one-time programming" requirement. According to the specification, each flash page in a clean erase block should be programmed (written) only once. In practice, flash chips can allow multiple programs to a page and the risk of "program disturb" is fairly low [7]. We can leverage this feature to simplify many designs. For example, we can write multiple versions of LBA/VBA and fingerprints into the spare area of a physical page, which can largely remove the need for metadata pages. Another consideration is to integrate a byte-addressable persistent memory (e.g. PCM) into the SSDs to maintain the metadata, which can remove much design complexity. We are also considering the addition of on-line compression into SSDs to better utilize the high-speed processor on the device. This can further extend available flash space but may require more changes to the FTL design, which will be our future work.

As SSD technology becomes increasingly mature and delivers satisfactory performance, we believe, the endurance issue of SSDs, particularly high-density MLC SSDs, opens many new research opportunities and should receive more attention from researchers.

Acknowledgments

We are grateful to our shepherd Dr. Christos Karamanolis from VMware® and the anonymous reviewers for their constructive comments. We also thank our colleague Bill Bynum for reading this paper and his suggestions. This research was partially supported by the NSF under grants CCF-0620152, CCF-072380, and CCF-0913150.

References

[1] FIPS 180-1, Secure Hash Standard, April 1995.

[2] Hadoop. http://hadoop.apache.org/, 2010.

[3] Personal communications with an SSD architect, 2010.

[4] SimpleScalar 4.0. http://www.simplescalar.com/v4test.html, 2010.

[5] SSD extension for DiskSim simulation environment. http://research.microsoft.com/en-us/downloads/b41019e2-1d2b-44d8-b512-ba35ab814cd4/, 2010.

[6] Agrawal, N., Prabhakaran, V., Wobber, T., Davis, J. D., Manasse, M., and Panigrahy, R. Design tradeoffs for SSD performance. In *Proceedings of USENIX'08* (Boston, MA, June 2008).

[7] Andersen, D. G., and Swanson, S. Rethinking flash in the data center. In *IEEE Micro* (July/Aug 2010).

[8] ARM. Cortex R4. http://www.arm.com/products/processors/cortex-r/cortex-r4.php, 2010.

[9] Balakrishnan, M., Kadav, A., Prabhakaran, V., and Malkhi, D. Differential RAID: Rethinking RAID for SSD Reliability. In *Proceedings of EuroSys'10* (Paris, France, April 2010).

[10] BARROSO, L. A. Warehouse-scale computing. In *Keynote in the SIGMOD'10 conference* (Indianapolis, IN, June 2010).

[11] BHAGWAT, D., ESHGHI, K., LONG, D. D. E., AND LILLIBRIDGE, M. Extreme binning: Scalable, parallel deduplication for chunk-based file backup. In *Proceedings of MASCOTS'09* (London, UK, September 2009).

[12] BLOOM, B. H. Space/time trade-offs in hash coding with allowable errors. In *Communications of the ACM* (1970), vol. 13(7), pp. 422–426.

[13] BOBOILA, S., AND DESNOYERS, P. Write endurance in flash drives: Measurements and analysis. In *Proceedings of FAST'10* (San Jose, CA, February 2010).

[14] BUCY, J., SCHINDLER, J., SCHLOSSER, S., AND GANGER, G. DiskSim 4.0. http://www.pdl.cmu.edu/DiskSim, 2010.

[15] CHEN, F., JIANG, S., AND ZHANG, X. SmartSaver: Turning flash drive into a disk energy saver for mobile computers. In *Proceedings of ISLPED'06* (Tegernsee, Germany, October 2006).

[16] CHEN, F., KOUFATY, D. A., AND ZHANG, X. Understanding intrinsic characteristics and system implications of flash memory based solid state drives. In *Proceedings of SIGMETRICS/Performance'09* (Seattle, WA, June 2009).

[17] CHEN, F., LEE, R., AND ZHANG, X. Essential roles of exploiting internal parallelism of flash memory based solid state drives in high-speed data processing. In *Proceedings of HPCA'11* (San Antonio, TX, Feb 2011).

[18] CHEN, S. FlashLogging: Exploiting flash devices for synchronous logging performance. In *Proceedings of SIGMOD'09* (Providence, RI, June 2009).

[19] DEBNATH, B., SENGUPTA, S., AND LI, J. ChunkStash: Speeding up inline storage deduplication using flash memory. In *Proceedings of USENIX'10* (Boston, MA, June 2010).

[20] DIRIK, C., AND JACOB, B. The performance of PC solid-state disks (SSDs) as a function of bandwidth, concurrency, device, architecture, and system organization. In *Proceedings of ISCA'09* (Austin, TX, June 2009).

[21] GAL, E., AND TOLEDO, S. Algorithms and data structures for flash memories. In *ACM Computing Survey'05* (2005), vol. 37(2), pp. 138–163.

[22] GRUPP, L. M., CAULFIELD, A. M., COBURN, J., SWANSON, S., YAAKOBI, E., SIEGEL, P. H., AND WOLF, J. K. Characterizing flash memory: Anomalies, observations, and applications. In *Proceedings of MICRO'09* (New York, NY, December 2009).

[23] GUPTA, A., KIM, Y., AND URGAONKAR, B. DFTL: a flash translation layer employing demand-based selective caching of page-level address mappings. In *Proceedings of ASPLOS'09* (Washington, D.C., March 2009).

[24] GUPTA, D., LEE, S., VRABLE, M., SAVAGE, S., SNOEREN, A. C., VARGHESE, G., VOELKER, G. M., AND VAHDAT, A. Difference Engine: Harnessing memory redundancy in virtual machines. In *Proceedings of OSDI'08* (San Diego, CA, 2008).

[25] INTEL. Intel X25-E extreme SATA solid-state drive. http://www.intel.com/design/flash/nand/extreme, 2008.

[26] JOSEPHSON, W. K., BONGO, L. A., FLYNN, D., AND LI, K. DFS: A file system for virtualized flash storage. In *Proceedings of FAST'10* (San Jose, CA, February 2010).

[27] KAWAGUCHI, A., NISHIOKA, S., AND MOTODA, H. A flash-memory based file system. In *Proceedings of USENIX Winter* (New Orleans, LA, Jan 1995), pp. 155–164.

[28] KIM, H., AND AHN, S. BPLRU: A buffer management scheme for improving random writes in flash storage. In *Proceedings of FAST'08* (San Jose, CA, February 2008).

[29] LEE, S., AND MOON, B. Design of flash-based DBMS: An in-page logging approach. In *Proceedings of SIGMOD'07* (Beijing, China, June 2007).

[30] LILLIBRIDGE, M., ESHGHI, K., BHAGWAT, D., DEOLALIKAR, V., TREZISE, G., AND CAMBLE, P. Sparse indexing: Large scale, inline deduplication using sampling and locality. In *Proceedings of FAST'09* (San Jose, CA, 2009).

[31] MAKATOS, T., KLONATOS, Y., MARAZAKIS, M., FLOURIS, M. D., AND BILAS, A. Using transparent compression to improve SSD-based I/O caches. In *Proceedings of EuroSys'10* (Paris, France, April 2010).

[32] MENEZES, A. J., V. OORSCHOT, P. C., AND VANSTONE, S. A. Handbook of applied cryptography. In *CRC Press* (1996).

[33] MOHAN, V., SIDDIQUA, T., GURUMURTHI, S., AND STAN, M. R. How I learned to stop worrying and love flash endurance. In *Proceedings of HotStorage'10* (Boston, MA, June 2010).

[34] NARAYANAN, D., THERESKA, E., DONNELLY, A., ELNIKETY, S., AND ROWSTRON, A. Migrating enterprise storage to SSDs: analysis of tradeoffs. In *Proceedings of EuroSys'09* (Nuremberg, Germany, March 2009).

[35] NETCONSOLE. http://www.kernel.org/doc/Documentation/ networking/netconsole.txt, 2010.

[36] PRABHAKARAN, V., BALAKRISHNAN, M., DAVIS, J. D., AND WOBBER, T. Depletable storage systems. In *Proceedings of HotStorage'10* (Boston, MA, June 2010).

[37] PRABHAKARAN, V., RODEHEFFEER, T. L., AND ZHOU, L. Transactional flash. In *Proceedings of OSDI'08* (San Diego, CA, December 2008).

[38] PRITCHETT, T., AND THOTTETHODI, M. SieveStore: A highly-selective, ensemble-level disk cache for cost-performance. In *Proceedings of ISCA'10* (Saint-Malo, France, June 2010).

[39] RAMAKRISHNAN, R., AND GEHRKE, J. Database managment systems. McGraw-Hill, 2030.

[40] RIVEST, R. The MD5 message-digest algorithm. http://www.ietf.org/rfc/rfc1321.txt, April 1992.

[41] ROSENBLUM, M., AND OUSTERHOUT, J. K. The design and implementation of a log-structured file system. In *ACM Transactions on Computer Systems* (1992), vol. 10(1):26-52.

[42] SOUNDARARAJAN, G., PRABHAKARAN, V., BALAKRISHNAN, M., AND WOBBER, T. Extending SSD lifetimes with disk-based write caches. In *Proceedings of FAST'10* (San Jose, CA, February 2010).

[43] SUN, G., JOO, Y., CHEN, Y., NIU, D., XIE, Y., CHEN, Y., AND LI, H. A hybrid solid-state storage architecture for the performance, energy consumption, and lifetime improvement. In *Proceedings of HPCA'10* (Bangalore, India, Jan 2010).

[44] TSIROGIANNIS, D., HARIZOPOULOS, S., AND SHAH, M. A. Query processing techniques for solid state drives. In *Proceedings of SIGMOD'09* (Providence, RI, June 2009).

[45] UNGUREANU, C., ATKIN, B., ARANYA, A., GOKHALE, S., RAGO, S., CALKOWSKI, G., DUBNICKI, C., AND BOHRA, A. HydraFS: A high-throughput file system for the HYDRAstor content-addressable storage system. In *Proceedings of FAST'10* (San Jose, CA, 2010).

[46] WIKIPEDIA. Battery or supercap. http://en.wikipedia.org/wiki/ Solid-state-drive#Battery_or_SuperCap, 2010.

[47] ZHU, B., LI, K., AND PATTERSON, H. Avoiding the disk bottleneck in the data domain deduplication file system. In *Proceedings of FAST'08* (San Jose, CA, 2008).