Don't stack your Log on my Log

Jingpei Yang, Ned Plasson, Greg Gillis, Nisha Talagala, Swaminathan Sundararaman
SanDisk Corporation

Abstract

Log-structured applications and file systems have been used to achieve high write throughput by sequentializing writes. Flash-based storage systems, due to flash memory's out-of-place update characteristic, have also relied on log-structured approaches. Our work investigates the impacts to performance and endurance in flash when multiple layers of log-structured applications and file systems are layered on top of a log-structured flash device. We show that multiple log layers affects sequentiality and increases write pressure to flash devices through randomization of workloads, unaligned segment sizes, and uncoordinated multi-log garbage collection. All of these effects can combine to negate the intended positive affects of using a log. In this paper we characterize the interactions between multiple levels of independent logs, identify issues that must be considered, and describe design choices to mitigate negative behaviors in multi-log configurations.

1 Introduction

Flash-based devices are frequently used for performance-sensitive applications ranging from databases to key-value stores to persistent messaging. In many of these environments, applications began by using flash as a fast disk and then made optimizations to better match the unique characteristics of flash. Since flash devices are known for asymmetric write performance and garbage collection (GC), a frequent application design pattern is to write in a log structure to optimize for flash devices. Recent examples include twitter fatcache [1], NILFS [11], F2FS [6], and SILT [12].

The log-structured write pattern has been adopted by both user-space applications and file systems. Such software runs atop the SSD's log-structured or data-remapping layer - the Flash Translation Layer (FTL). Therefore, it is possible that two or more log-structured I/O patterns may become stacked on flash media. For example, it is possible to have an application like fatcache write a sequential stream atop a log-structured file system like F2FS, which in turn operates over a log-structured FTL on physical flash media.

While log-structured applications, file systems and log stacking is not new [5], log stacking on flash deserves special attention. First, since flash devices contain a remapping log-like FTL, any log-structured application run atop a flash device creates a stacked log scenario, making such scenarios now more common. Second, flash devices have limited endurance and any additional writes caused by multiple log layers can impact device lifetime. Third, each layer's log-remapping engine frequently reserves some capacity for GC and only exposes part of its usable capacity to the upper layer. Thus a large fraction of, the still relatively expensive, flash media can be consumed as reserve capacity by multiple logs stacked atop it. Fourth, the high performance of flash devices implies that log "aging", or the need for GC to defragment the log, occurs quickly, frequently, and incoherently amongst all the logs involved. This combined incoherent GC behavior, across multiple log layers, critically impacts overall performance and endurance.

We focus on Log on Log - the issues that arise when two or more log layers are stacked on each other. At first glance, we observe that multiple layers of software performing the same function, i.e. data remapping and GC, seems redundant and suboptimal. In a multi-layer log configuration, there are further issues. Each log structure is unaware of the objectives and algorithms of those below or above it. Since each log operates independently towards its own objectives, it is possible that its performance or efficiency goals can be undone by the other log layers. In addition, increased metadata, conflicting and incoherent GC strategies, and fragmentation of work-

INFLOW'14, October 5, Broomfield, CO, USA.

loads, all result in increased write pressure, which greatly impacts flash device performance and endurance. This can also result in a great performance reduction of the overall application using these multiple log layers.

This paper makes the following contributions:

1. We outline the architectural issues that can arise when one or more logs are stacked atop an FTL.

2. We demonstrate the impacts of these issues on flash devices using a combination of two techniques. First, we gather empirical results of workloads on log-structured software running atop a commercially available flash device. We then assess the issues in depth using a purpose-built log-on-log event driven simulator. We measure the impact of multiple uncoordinated log activities and demonstrate that, multi-layer log configurations introduce higher write pressures (up to 33%) from log metadata maintenance, and increased GC activities (up to 32%) due to decoupled cleaning.

3. We propose some optimizations to mitigate the issues found with multi-layer log configurations. We propose optimal sizing of log segment sizes amongst layers and coordination of GC interactions. In addition, we discuss approaches to collapsing logs through new interface semantics.

This paper argues that the increasingly common practice of using log-structured writing to flash is fraught with complexities and opportunities for unpredictable behavior. We outline ways to both understand and mitigate the effects of log stacking, and discuss alternatives to stacked logs over flash.

2 Background

Log-structured data persistence has been employed in storage systems [4], file systems [16], databases [20] and other applications. Some stores are strictly log structured and allow no update-in-place operations, while other stores are more write-anywhere in nature [5] and allow hole plugging. All such stores allow new writes to be directed to free space in the device, and all contain some form of GC (frequently called cleaning) to compact and reuse invalidated physical space. Substantial research has been done on optimizing log-structured stores, particularly for GC [2, 17, 21]. In this paper, we use the term "log-structuring" generally to mean stores with dynamic remapping of writes and GC. Specific configurations of such stores are defined and explored in detail in Sections 3 and 4.

Prior to the arrival of flash, a key motivation for log-structured stores was to accelerate write performance while allowing random reads to be serviced from DRAM cache. Log-appends provide additional advantages, such as enabling snapshots, enabling transactional updates, and eliminating the small write performance problem when used in RAID 5 configurations [5, 14].

Flash creates a new motivation for log structuring. Flash can only be erased in the unit of erase blocks which are typically much larger than the write unit (e.g. 512 write pages per erase block). As such, all new writes must be directed to (freshly erased) blocks. Erased blocks are made available to satisfy new writes through GC. One or more erase blocks are garbage-collected together, making them conceptually similar to cleaned segments in a log-structured file system. Since flash has a limited number of program/erase cycles, flash GC has to balance the efficiency of cleaning with erase block wear leveling to meet reliability requirements. Flash has additional requirements, such as read disturb handling, which require rewrites to maintain data integrity. As such, while some of the factors that drive flash GC are similar to those driving cleaning in higher level log stores, others are flash media specific.

Recently some efforts have been directed towards the reduction of the cost of journaling of journals (similar to a log-stacking model) between the application and file system layer [10, 18]. This work observed, as we do, the general inefficiency of having redundant work done in multiple log layers. Our work is complementary to these efforts in that we aim to understand the behavior of a more generalized multi-log stacking model and its impact on flash, focusing on write amplification, GC overhead and overall performance.

3 Approach

We start by outlining several different models of log stacking that can commonly occur with flash devices. We then define a number of architectural aspects of logging, GC and write amplification which we then use in the subsequent sections to analyze log-on-log interactions.

3.1 Log Stacking Models

Figure 1 outlines some of the log stacking configurations that can occur when log-structured applications meet log-structured file systems and/or flash devices. Figure 1a represents a single log-structured application (or file system) residing on a single FTL-based SSD. This is the most basic example of a log-on-log configuration. Some form of this configuration occurs every time a log-structured application runs on an SSD. The illustration demonstrates the potential complexities that can occur even in a simple log-on-log scenario. In this example, the upper level log has three data types (data, metadata, and garbage collection) that are being written to three sequential streams. The underlying lower level log has two sequential streams. Figure 1b outlines a configuration where a log-based application/filesystem and a non-log based application share one FTL. This configuration can commonly occur when an SSD is divided into two partitions and one partition is used by a log-structured filesystem while the other is used by an application with a very different access pattern. Other configurations of multiple log layers include Figure 1c, where two or more log-structured applications share an FTL, and Figure 1d, where a log-structured application (such as a key-value store), resides on top of another log-structured software module (such as a file system) which itself is on top of an FTL.

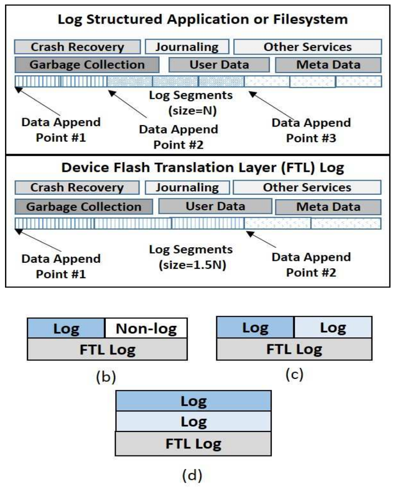


Figure 1: Log-on-log structured approaches can be used in all levels of the storage stack.

3.2 Append Streams

Log stacking is further complicated as each log-structured application can have multiple streams over a single internal address space. Figure 1a shows an example of multiple sequential streams within each log layer. We call each such stream an Append Stream writing to the Append Point. An append stream is a sequential stream of writing and subsequent GC, similar to that used in [8]. We assume that all writes of an append stream occur at the head (the Append Point) for that stream, and that reads can occur from anywhere within the stream. In addition, GC can read from any part of the stream and write subsets of the data to the append point. While some log-structured architectures are strictly single append stream, implying that all writes, incoming, cleaning, metadata, are driven to the same append point, others have multiple streams. F2FS [6], for example, has six logical append streams, twitter fatcache has one, and SILT has several. Similarly, the FTL within a flash device may have one or more append streams depending on the design.

3.3 Write Amplification

As each log layer remaps and garbage collects its data, it generates its own write amplification (WA). The incoming data seen by each log layer includes the amplified writes generated by the log layers above. In this paper, we compute and refer to each log layer's WA separately. Each layer's WA is computed as: the ratio of outgoing writes from that layer, to the incoming writes of that layer. We shall make clear below which log level's WA we are dealing with at the moment. The total combined write amplification (TCWA) is computed as the product of all of the involved write amplification factors.

3.4 Evaluation Methodology

Armed with the above concepts, we explore a number of different log-on-log behaviors. We conduct two classes of experiments.

1. We use F2FS as an example of a flash-optimized log structured file system with multiple append streams. We run experiments with F2FS on top of a commercially available SSD.
2. We developed and used a log-on-log simulator that implements a two level log-on-log structure with up to two independent append streams at each layer. With the simulator, we measure and analyze in detail the WA generated by different log-on-log interactions. The simulator is independent of hardware and operating system configurations so that it could be abstracted as any two-layer log system.

4 Scenarios and Results

In this section, we analyze simple and frequently deployed log on log scenarios and demonstrate some of the issues that arise. We characterize their impact on write pressure, endurance and capacity efficiency.

4.1 Metadata Footprint

The first topic we examine is metadata footprint. At a cursory glance, log stacking is expected to increase metadata footprint, since each log layer will need to add its own metadata for the incoming data to track layout and persist indirection maps.

The amount of metadata added by a log structured store depends heavily on the design of the store and the number of append streams within the store. To understand the potential metadata overhead of log stacking and multiple streams, we perform experiments on

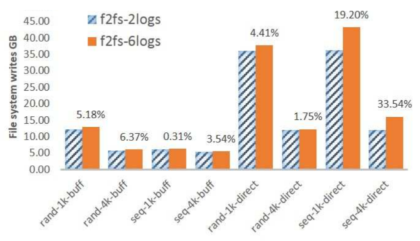
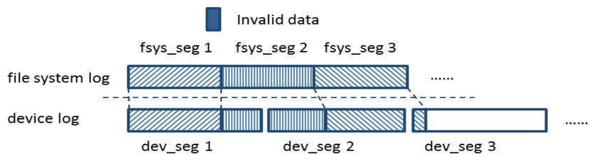


Figure 2: Metadata foot print increases as more append streams are used on file system.

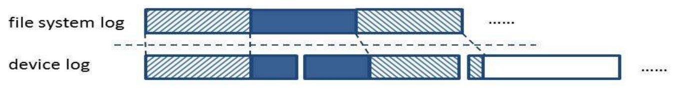
F2FS. F2FS is designed to support up to 6 append streams, making it possible for the file system to identify hot/warm/cold data and separate them to different segments. We measured the total file system write bytes issued to the device under different workloads while varying the number of F2FS append points. We configured F2FS to have 2 and 6 append streams. With 2 append points, F2FS separates user data and metadata, while 6 append points further differentiate each type of data as hot/warm/cold. The workloads were generated using the FIO benchmark tool with various combinations of workload configurations - 1k vs. 4k I/O size, buffered vs. direct I/O, and random vs. sequential writes. As is shown in Figure 2, with an application workload that writes a total of 8GiB, the file system generally writes more data to the device when the number of append points is increased (e.g. 2 to 6 append points). For example, the first column set shows the total number of file system writes issued to the device from an 8GiB random write workload with buffered IO and a 1k IO size. The file system amplifies the original writes due to file metadata and log metadata. Since the workload is the same for 2 and 6 append streams, we assume file metadata used to maintain file status remains the same. Thus, the increased writes from 2 to 6 streams are the consequence of the additional logs' metadata. Our experiment shows that the File System Write Amplification (FSWA) varies based on the number of file system append points, and increases from 1.5 to 2.0 (up to 33% for seq-4k-direct) when growing from 2 to 6 append points. While this is only one example, it does suggest that the number of append streams can be a factor in the WA generated by a log-structured store. FSWA is the amplification of the application workload by the filesystem. It is not the same as TCWA as it is the amplification of that amplification by the device.

4.2 Fragmentation

A key goal of log-structured systems is sequentializing writes. However, if the FTL is shared by two log-structured applications (or even a single application with multiple append streams), the incoming data into the FTL is likely to look random or disjoint. Additionally, GC in the upper layer can further complicate the traffic stream seen by the lower layer.



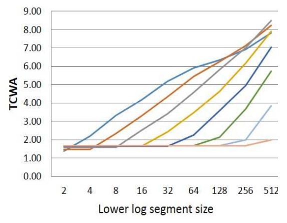
(a) Data is written sequentially to the lower log.



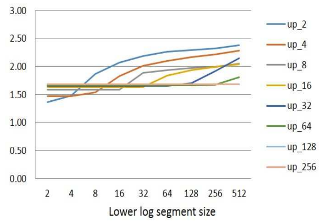
(b) Deleting one fsys_seg spread across two dev_seg.

Figure 3: Fragmented logs.

Even when each log layer has exactly one append stream, complexities exist that can cause the underlying device to see non-sequential traffic. One such complexity is segment size mismatch - where the upper and lower logs both do GC, but at different segment boundaries and sizes. Figure 3 illustrates this issue with an example of two logs, each with one append stream but GC-ing at different segment sizes. Data from one upper log segment is spread across two segments in the lower log. When GC occurs in the upper log stream, a deleted upper log segment (fsys_seg 2 in the example) results in partial invalidation of two lower log segments. Reclaiming space in the lower log now requires GC of two dev_seg, and results in higher WA in the lower log (see Section 4.5 for more detailed discussion on segment cleaning).



(a) upper/lower log capacity ratio 90%



(b) upper/lower log capacity ratio 70%

Figure 4: TPC-E: overall system WA (TCWA) varying log capacity ratio and segment size ratio.

When each log has a single append stream, this issue can be mitigated to some extent by matching segment sizes between upper and lower logs. We measured the impact of different upper/lower log segment size ratios using our log-on-log simulator. Figure 4 depicts one such result for a TPCE-like workload trace. For each line of a fixed upper log segment size in Figure 4, there is a dramatic change of slope when lower log segment size exceeds upper log segment size. This is because the reuse of upper segments (seen as *invalidation* by the lower log) cannot cover the entire lower log segment, and causes data fragmentation on the lower layer. As both layers GC becomes active, a large portion of valid data in each lower segment is copied forward resulting in higher lower log WA and hence higher TCWA.

This result further demonstrates that optimal segment sizes for log-structured GC that held true for standalone logs may not hold true for the whole system if it has a log-on-log scenario. A segment is the smallest unit for GC processing. In a standalone log, generally smaller segment sizes provide improved flexibility on GC victim selection and hence achieves lower WA. This is not true for a log-on-log configuration.

The above example illustrates the fragmentation and cleaning overheads that can result from two single stream logs being stacked atop one another. If each log were to have multiple append streams, the situation worsens since segments in the lower log are far more likely to have inter-mixed content from many upper log segments. It is also not clear that segment size matching can overcome the issues since data intermixing will still occur.

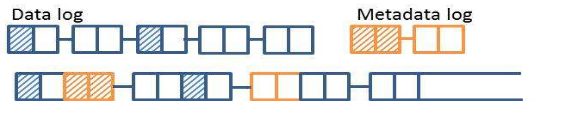
4.3 Aggregate Reserve Capacities

Since GC re-arranges data, many GCs rely upon some fraction of the underlying capacity to be reserved. When logs are stacked, each log layer's capacity reserve eats into the capacity available for user data. The behavior of a log-on-log configuration also depends on the capacity used (and reserved) by the GC at each level. Figure 4(b) shows the same log configuration but with more reserve capacity in the upper log. The turning slope in Figure 4(b) is at a larger lower segment size. This gives the upper log more flexibility on tuning its segment size to achieve lower FSWA. On the other hand, if each log's reserve capacity ratio is low, the lower log has more spare capacity exposed to the upper log, then device GC is triggered less actively.

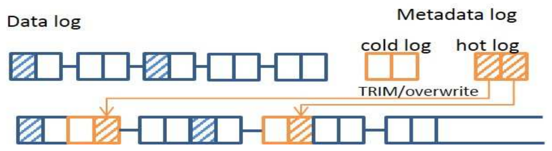
Our analysis of log-structured applications like NILFS and F2FS, as well as FTLs, has shown that each log has its own metadata which is invisible to the higher level logs. Due to this metadata, a segment contains metadata inter-mixed with data from upper logs. Hence cleaning of segments at one log layer doesn't preclude the need to clean the segments at another layer. As our simulation is conducted with no other traffic nor log metadata, the real log-on-log system will be more complex and introduce higher degree of log fragmentation, making the impact of size ratio between two layers harder to predict.

4.4 Multiple Append Streams/Points

Fragmentation and associated complexity only increases if upper layers have more than one append stream. Multiple append points are useful, for example, to separate hot and cold data during GC, or to separate data with different characteristics. As shown in Figure 5, if a lower



(a) With fewer append points, data invalidations are more aggregated in lower log.



(b) Increasing upper append ponts tends to distribute invalidation across more lower segments, resulting in higher fragmentation and higher WA.

Figure 5: The higher the ratio of upper to lowers logs, the higher the degree of data fragmentation.

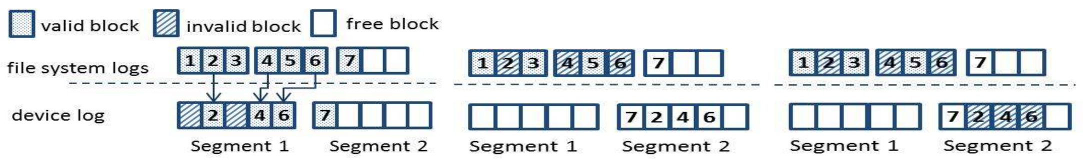
level log does not support multiple append points, data from multiple higher level append streams will become inter-mixed at the lower layer.

If a log-structured store supports advanced capabilities such as separating data of different update frequencies to reduce GC overhead, each log can be expected to show different activeness according to its data characteristics. For example, a file system metadata log can be smaller and more active than a data log, if the user workload involves many file creates and deletes. While the original workload remains the same, separating data to multiple append points tends to make the invalidation further distributed across segments in a lower log append stream. The distributed invalidity and different validity views limit the flexibility for GC selection [22]. As the number of upper log append points increases, the randomness in the invalidity at the lower log layer increases thus further limiting GC selection.

If m upper logs are stacking on n lower logs, the ratio of (m/n) can indicate the degree of data fragmentation on device. The higher the ratio is, the more likely that a single lower log append stream may contain data from more different upper logs. With different log activities, such as update, invalidation, and GC frequencies, the lower log will suffer from high fragmentation, and hence higher WA over time. If upper logs have a diverse range of activeness, for example, some applications may be more active than others during a certain period of time, or some logs are much smaller than others, it increases the degree of data fragmentation as seen by the lower logs and results in high device GC overhead.

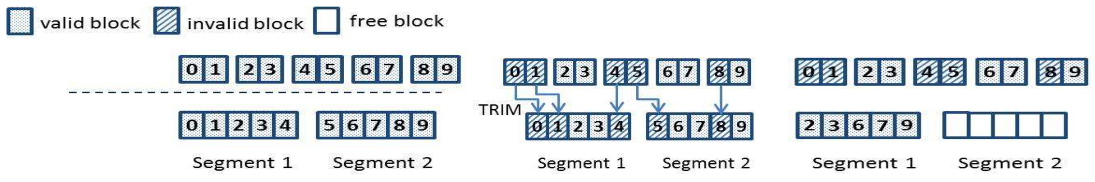
4.5 Layered Garbage Collection

Each log has its own GC or cleaning process which operates independently. We now examine the combined effects of such layered independent GC. By design, logs work in isolation (i.e., manage their free space themselves) and are unaware of other logs above or under-



(a) Initial status. (b) Upper log deletes blocks before device GC starts. (c) Upper log GC starts after device GC finishes.

Figure 6: Decoupled segment cleaning without TRIM



(a) Initial status. (b) TRIM by upper log. (c) Device GC starts cleaning.

Figure 7: Decoupled segment cleaning with TRIM.

neath. In addition to the obvious inefficiencies of the same kinds of work being redundantly done in multiple layers, layered GC has several other problems.

4.5.1 Need for Layered TRIMs

The first issue that arises is the need for TRIM. Without TRIM, data presumed to be valid at the lower layer need not be valid at the upper layer. Invalidations at the upper logs need not trickle down to the lower log, as a result, lower logs operate with outdated validity information. In this scenario the lower-log layer will perform segment cleaning operations that move data that is invalid at the upper log but valid in the lower log. As such an increase in write pressure is incurred from moving data that is invalid from the upper layers perspective.

The need for TRIM is well known [15]. However, TRIM is still only commonly implemented between the device and driver software or between file system and device. All layers of logs need the ability to communicate an equivalent of TRIM to their lower layers. Such capabilities are now starting to become available through new APIs and extensions of existing POSIX calls for user space apps [3, 19] and are needed to make log-on-log configurations effective for user-level log-structured applications.

4.5.2 Uncoordinated Garbage Collection

The second issue that arises is lack of timing coordination between GCs. Each log layer performs segment cleaning while being agnostic of the activities in the other log layers. Consider the case where the segment cleaner of the lower log append stream(s) runs ahead of the segment cleaners of the upper log append stream(s) . In this situation, the lower log could clean a segment that contains one or more segments from the upper log. After the lower log cleaning is done, the segment cleaner(s) of the upper log will move the data and rewrite the segments in the lower log again, causing avoidable writes and impacting endurance.

A high degree of data fragmentation due to different log characteristics and activities (discussed in Section 4.2) increases device GC pressure and WA, and garbage collection on both layers further fragments the data layouts. While a file system makes an effort to write, overwrite, and invalidate its segments sequentially, holes could be made at the layer with larger segment size due to the unmatched segment size or page size as well as the timing issue for both layers' GC. Without TRIM, Figure 6(c), when the file system GC invokes after device GC has copied block 2, 4 and 6, the overwrite operation sent to the device will invalidate those blocks, and causes fragmentation on the media. As a result, when device GC wakes up again, valid data (Block 7 in this case) will be copied forward. This further increases the WA due to the mixed placement of valid data and invalid data. Even with TRIM, Figure 7, the fragmentation problem still exists. When the file system TRIMs the entire segment (Figure 7(b)), the underlying device segments only invalidates a portion of them. Moreover, if the file system GC happens after device GC, those valid blocks will be copied at least twice. In addition, the device GC process usually involves several stages including segment scan, victim segment selection, and valid data re-read and rewrite. In order to not block incoming requests in a high performance system, this process is multi-threaded and the system alternates its activity with the handling of new IO requests. As such data written by the GC will be intermixed with new write operations, it thusly increases the degree of fragmentation.

4.5.3 Conflicting Optimizations

As the above sections show, optimizations such as multiple append points, which can be quite desirable in a single log store, generate complexities and unpredictable behavior in a log-on-log scenario. For example, segment cleaning based on hot and cold segments in an upper log need not hold true at a lower log. even if the lower log has multiple append streams. Reasons include interspersed data from segments across (and within) logs, segment cleaning at an upper log layer translating to data being misclassified as "hot" data in the lower log layer, and segment cleaning at an lower log moving "cold" data from the upper log (which could have been invalidated).

random	log	F2FS	FTL	erase	GC data	GC
dist	#	Upper log	Lower log	cnt	GB	WA
		W GB	W GB			
zipf:0.8	2	120.55	221.02	370	98.28	1.82
zipf:0.8	6	121.08	267.88	473	144.58	2.19
zipf:1.1	2	122.05	222.78	374	98.52	1.81
zipf:1.1	6	122.53	277.10	493	152.35	2.24
uniform	2	96.32	137.94	188	39.96	1.41
uniform	6	96.42	141.25	195	43.15	1.45

Table 1: Device WA varies with different number of upper layer append points.

4.5.4 Experimental results

To examine the potential impacts of uncoordinated segment cleaning, we measured the FTL GC overhead with 2 and 6 upper logs using F2FS on an SSD. Under the same workload of 60GB random writes, 4k IO size and direct IO, the file system with 6 logs writes slightly more data to the device than 2 logs (Column 3 - fsys W GB in Table 1). However, the 6-log case suffers from a much higher WA. Different distributions of hot and cold data, the reduced size of invalidated extents, and uncoordinated GC increases fragmentation in the 6-log configuration, and results in much higher device GC and WA than in the 2-log case. In the 6-log case, the increased log metadata also contributes to more device writes and higher GC. However in this experiment, we can subtract the metadata effect by assuming such increased writes are amplified by the same scale as other writes in the device layer. Thus, the much higher WA and erase counts with 6 logs are mostly caused by the effect of different data activities distributed across logs and decoupled cleaning.

4.6 Discussion

Through our exploration of log-on-log effects, we have made a few key observations. First, the issues:

(a) There are many redundant operations and inherent inefficiencies when logs are stacked atop each other. These include redundant GC at each layer as well as possibly redundant data reorganization (such as into hot and cold) at each layer. Left untuned, these effects can be counterproductive, resulting in higher WA and reduced device lifetime.

(b) TRIM is critical to pass intelligence across log layers. While TRIM is now supported by many FTLs, TRIM implementations in user space are only starting to emerge.

(c) Even with the optimizations in (b) and (c), it may still be necessary to coordinate GC across log layers to reduce overall WA. more efficient

(d) The situation is further complicated by the potential existence of multiple append points within each log layer. While in some software (such as F2FS) the number of append points and their focus is documented, most SSDs do not advertise the number of append points. As such, arbitrary layering combinations are possible in real systems, where logs with m append points can be placed on logs with n append points where: (1) m $<$ n; (2) m = n; and (3) m $>$ n.

(e) Finally, we have noted that optimizing each log structured module in a vacuum can lead to suboptimal behavior at the system level when logs are stacked. In particular, common algorithms for hot/cold separation and segment size choice, which seem optimal for a single log are often suboptimal for log-on-log configurations.

That said, there are often good reasons for log structures to exist at the application and file level, since they provide additional functionality like snapshots and transaction rollback. Given this, it may not be possible to collapse logs entirely. Our studies also uncovered a number of ways that log-on-log structures can be optimized, namely by Log Aware Coordination between layers:

- We have shown that log and segment size impact the log-on-log systems. When TCWA is determined by the combination of both layers' WAs, the decision on each layer's segment size is not stand-alone. Generally, if the upper/lower size capacity ratio is lower, it is beneficial to choose a smaller upper segment size for reduced FSWA and TCWA. When the log sizes are close to each other, making upper segment greater than or equal to the lower segment size can achieve reduced device WA and TCWA.

- Data is moved multiple times due to different views of invalidation across logs. While both layers GC are working actively, coordinating them to avoid unnecessary data moves becomes critical. With support from TRIM, each log is able to keep the same view of data validity. In addition, if lower log can postpone its GC process while upper GC is active, by the time lower GC starts, more data will be invalidated, which reduces device WA.

The work in [8] argues for a better awareness of streaming behavior from applications to flash FTLs, forming another possible way to improve stacked log behavior if it cannot be eliminated.

5 Collapsing logs

We have discussed the interactions between multiple levels of independent logs and several practical scenarios which arise in today's deployed systems, as well as several ways to optimize multiple log layers if they must exist. In this section, we describe briefly alternative directions for such systems to collapse the log layers entirely and remove redundant behavior.

Redundant log layers exist in today's systems partly because the block device semantics are not rich enough to expose the characteristics of the underlying log layers. Several attempts have been made to overcome this through richer interfaces. As is shown in the left part of Figure 8, each layer of log offers similar functionalities which is isolated by the block layer. By breaking the block interface, redundant behavior could be eliminated. Same capabilities could be either kept in file system layer with a lightweight flash device design, or vice versa, as is shown in the right part of Figure 8.

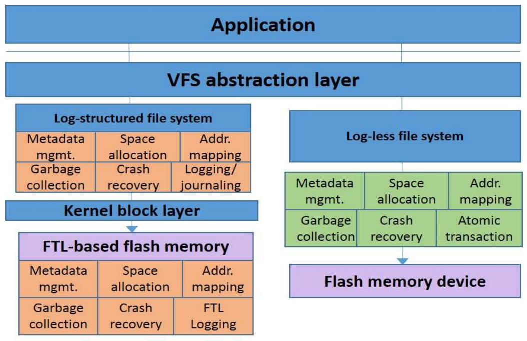


Figure 8: Log-less flash-aware system design.

Sparse Addressing and Transactional Semantics NVMFS (formerly called DirectFS) is an extension of the ideas which leverage the underlying FTL log structure via four primitives [7]. A sparse address space represented by the FTL enables NVMFS to eliminate its own mappings of <file, offset> to physical block. Rather, directFS maps each file to a small number of virtual extents, and relies upon the FTL to allocate physical blocks to these virtual addresses as blocks are consumed. In place of a journal, NVMFS uses two primitives, atomic writes and persistent TRIMs [15] provided by the underlying FTL. The atomic writes are executed entirely or not at all, and persistent TRIMs (also atomic) provide transactional deletes of virtual address ranges. By using these in combination as a group of transactional updates and deletes, NVMFS can move from a transactionally consistent state to transactionally consistent state without an independent journal. Finally, directFS uses statistics exported by the FTL on allocated block counts to maintain accurate counts of physical space consumption, which limits updates of superblocks when files are extended.

Object-based storage. Another alternative is to break the fixed-size block interface via direct specification of objects or extents that can then be managed by the lower level log layer (and for example can be exported by an FTL). The use of object-based file system [9] or object-based FTL [13] leverages the underlying flash translation layer to manipulate objects placement which is transparent to the upper layer. Meanwhile, object-based file system only manages name resolution, thus no log or other complicated device-dependent mechanism is needed in the upper layer. This provides portability and compatibility design to various types of devices. Meanwhile, advanced features can be embedded into object-based device through a rich object interface.

6 Conclusion

In this paper we demonstrated the impact of stacking one or more layers of logs on top of a log-structured flash device. Through our simulation and empirical results, we show that the increased write pressure and destroyed sequentiality due to unaligned segment sizes, unpredictable workloads, and uncoordinated log activities such as garbage collection negates many of the positive affects of using a log. While applications and file systems will continue to use log-structure for their own performance and reliability purposes, we propose some optimizations to mitigate the issues found, e.g., log segment size adjustment amongst layers and coordination of GC interactions. We plan to further explore alternatives to collapsing logs for flash in future work.

7 Acknowledgements

We thank the anonymous reviewers for helping us refine this paper. We also thank our colleagues in the Advanced Development Group at SanDisk Corporation and Prof. Scott Brandt at UC Santa Cruz for their feedback.

References

[1] FatCache. https://github.com/twitter/fatcache.

[2] Trevor Blackwell, Jeffrey Harris, and Margo Seltzer. Heuristic cleaning algorithms in log-structured file systems. In *Proceedings of the USENIX 1995 Technical Conference Proceedings*, TCON'95, pages 23–23, Berkeley, CA, USA, 1995. USENIX Association.

[3] Dhananjoy Das. NVM-Compression: flash enabled compression. Percona Live MySQL Conference and Expo, 2014.

[4] Wiebren de Jonge, M. Frans Kaashoek, and Wilson C. Hsieh. The logical disk: a new approach to improving file systems. In *Proceedings of the fourteenth ACM symposium on Operating systems principles*, 1993.

[5] Dave Hitz, James Lau, and Michael Malcolm. File system design for an NFS file server appliance. In *Proceedings of the USENIX Winter 1994 Technical Conference on USENIX Winter 1994 Technical Conference*, WTEC'94.

[6] Kim Jaegeuk. F2FS:flash-friendly file system. http://en.wikipedia.org/wiki/F2FS.

[7] William K. Josephson, Lars A. Bongo, Kai Li, and David Flynn. DFS: A file system for virtualized flash storage. *Trans. Storage*, pages 14:1–14:25, 2010.

[8] Jeong-Uk Kang, Jeeseok Hyun, Hyunjoo Maeng, and Sangyeun Cho. The multi-streamed solid-state drive. In *6th USENIX Workshop on Hot Topics in Storage and File Systems (HotStorage 14)*, Philadelphia, PA, June 2014. USENIX Association.

[9] Yangwook Kang, Jingpei Yang, and Ethan L. Miller. Object-based SCM: An efficient interface for storage class memories. In *Proceedings of the 27th IEEE Conference on Mass Storage Systems and Technologies (MSST 2011)*, May 2011.

[10] Wook-Hee Kim, Beomseok Nam, Dongil Park, and Youjip Won. Resolving journaling of journal anomaly in android i/o: Multi-version b-tree with lazy split. In *Proceedings of the 12th USENIX Conference on File and Storage Technologies*, FAST'14, pages 273–285, Berkeley, CA, USA, 2014. USENIX Association.

[11] Ryusuke Konishi, Yoshiji Amagai, Koji Sato, Hisashi Hifumi, Seiji Kihara, and Satoshi Moriai. The linux implementation of a log-structured file system. *SIGOPS Oper. Syst. Rev.*, 40(3):102–107, July 2006.

[12] Hyeontaek Lim, Bin Fan, David G. Andersen, and Michael Kaminsky. SILT: a memory-efficient, high-performance key-value store. In *Proceedings of the Twenty-Third ACM Symposium on Operating Systems Principles*, SOSP '11, 2011.

[13] Youyou Lu, Jiwu Shu, and Weimin Zheng. Extending the lifetime of flash-based storage through reducing write amplification from file systems. In *Proceedings of the 11th Conference on File and Storage Systems (FAST 2013)*, February 2013.

[14] Ian Murdock and John H. Hartman. Swarm: a log-structured storage system for linux. In *Proceedings of the annual conference on USENIX Annual Technical Conference*, ATEC '00, pages 28–28, Berkeley, CA, USA, 2000. USENIX Association.

[15] David Nellans, Michael Zappe, Jens Axboe, and David Flynn. PTRIM + EXISTS: Exposing new FTL primitives to applications. In *2nd Annual Non-Volatile Memories Workshop*, 2011.

[16] Mendel Rosenblum. The design and implementation of a log-structured file system. Technical report, Berkeley, CA, USA, 1992.

[17] Stephen M. Rumble, Ankita Kejriwal, and John Ousterhout. Log-structured memory for dram-based storage. In *Proceedings of the 12th USENIX Conference on File and Storage Technologies (FAST 14)*, pages 1–16, Santa Clara, CA, 2014. USENIX.

[18] Kai Shen, Stan Park, and Meng Zhu. Journaling of journal is (almost) free. In *Proceedings of the 12th USENIX Conference on File and Storage Technologies*, FAST'14, pages 287–293, Berkeley, CA, USA, 2014. USENIX Association.

[19] Nisha Talagala. OpenNVM: From standards to solutions - software optimizations for non-volatile memory. Percona Live MySQL Conference and Expo, 2014.

[20] TokuTek. Tokudb. http://www.tokutek.com/.

[21] Jun Wang and Yiming Hu. Wolf–a novel reordering write buffer to boost the performance of log-structured file system. In *Proceedings of the 1st USENIX Conference on File and Storage Technologies*, FAST '02, Berkeley, CA, USA, 2002. USENIX Association.

[22] Jingpei Yang, Ned Plasson, Greg Gillis, and Nisha Talagala. HEC: Improving endurance of high performance flash-based cache devices. In *Proceedings of the 6th International Systems and Storage Conference*, SYSTOR ’13, pages 10:1–10:11, New York, NY, USA, 2013. ACM.