Minimizing Read Seeks for SMR Disk

Mohammad Hossein Hajkazemi*, Mania Abdi†, Peter Desnoyers†
Department of Electrical and Computer Engineering*, College of Computer and Information Science†
Northeastern University
hajkazemi@ece.neu.edu,abdi.ma@husky.neu.edu,pjd@ccs.neu.edu

***Abstract*—Log-structured storage systems are file or block storage systems which write data in temporal order, rather using e.g. LBA or other spatial information to determine physical location. They are widely used as translation layers for flash and now Shingled Magnetic Recording (SMR) disk, performing the out-of-place write function needed for media where direct overwriting of data is not possible.**

Although certain aspects of log-structured storage system performance (i.e. write amplification) have been extensively studied, few or no efforts have examined the effect of seek overhead on workloads in this context. We examine seek overheads due to log-structured writing across a range of well-known and more recent block I/O traces, demonstrating that while some workloads are relatively unaffected, others suffer significantly.

In addition we propose two novel mechanisms, *opportunistic defragmentation* and *translation aware selective caching*, for seek reduction in log-structured systems, as well as an application of existing disk read-ahead techniques (which we name *translation aware look-ahead-behind prefetching*), and demonstrate that these mechanisms greatly reduce or even eliminate increases in read seek cost in almost all workloads examined. We evaluate our proposed techniques using a metric we define, seek amplification factor. Results show up to 4x, 4x and 18x improvement of seek amplification factor among the studied workloads for *opportunistic defragmentation*, *translation aware selective caching* and *translation aware selective caching* respectively.

I. Introduction

The term "log-structured" is typically used to refer to storage mechanisms where data is written sequentially in a new location, replacing an older version rather than over-writing it. The idea originated with file systems for write-once optical drives [1], as well as the pioneering work of Rosenblum, Ousterhout and others [2], [3], however for the most part early interest in this approach faded as the magnitude of cleaning cost and difficulty in reducing it became apparent [4], [5], and interest did not revive until the introduction of NAND flash years later. As a result although much effort has been spent analyzing the copying overhead (write amplification factor or WAF) of log-structured systems [6], [7], [8], [9], few works have examined disk seek behavior in these same systems. However the recent development of Shingled Magnetic Recording (SMR) [10], [11], a disk organization requiring out-of-place writes to avoid data loss, has made seek behavior in log-structured systems relevant again.

The original log-structured file system was motivated by the goal of reducing *write seeks*—disk seeks due to non-sequential writes. As an example, creation of a small file in a non-journaled file system might require writing six blocks in different parts of the disk; on disks of the era might take a few milliseconds in transfer time, but hundreds of ms in seek time. By writing these blocks sequentially, write seek overhead was almost eliminated and performance was greatly improved. But what about seeks for read operations?

Early work on log-structured systems assumed that ever-increasing RAM sizes would allow caching to virtually eliminate the need to read from disk, eliminating read seeks as a performance consideration; however it was soon recognized that ever-increasing disk sizes canceled out those increasing RAM sizes, and that disk read performance would remain a significant performance factor. A few systems [12] attempted to reduce read seek overhead[1], but this concern became irrelevant when researchers' focus turned to flash-based log-structured systems. Although flash performance is affected by how evenly requests are distributed across planes and channels [13], there is nothing comparable to the 1000:1 performance disparity between sequential and random disk accesses. With the advent of SMR, however, which combines the no-overwrite constraint of flash with the random I/O performance penalty of disk, the question of read seek overhead due to log structured writing has become relevant again.

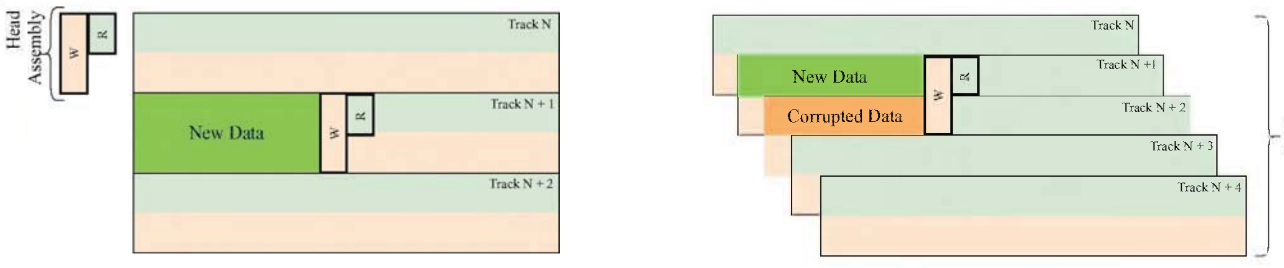
So, what is the impact of read seeks on the performance of log-structured, disk-based storage systems? One can readily construct scenarios where log-structured writes cause no increase in read seeks; conversely other artificial scenarios result in huge performance losses relative to update-in-place. But what about real-world workloads? Are the I/O patterns which perform poorly on log-structured disk rare ones, or commonplace? And when they occur, are there methods we can use to reduce their impact?

We address these questions in the context of block translation layers[2]: device- or host-resident algorithms which provide a conventional rewritable block abstraction layer on top of an underlying log-structured organization. We perform an extensive series of trace measurements and workload analysis, using block traces from production Linux and Windows servers. We measure the read seek overhead of log-structured writes in a simple infinite-disk model, examine the results in detail, and present and evaluate several mechanisms for improving performance.

In particular, the contributions of this paper are:

[1]It is likely that this topic received attention at NetApp, a vendor of disk-based log-structured storage systems, however any results have not been published.

[2]Termed Flash Translation Layers (FTLs) or Shingling Translation Layers (STLs) as appropriate.



(a) Conventional Magnetic Recording Drive (CMR) (b) Shingled Magnetic Recording Drive (SMR)

Fig. 1: Track layout of (a) CMR drive vs. (b) SMR drive; overlapping tracks in SMR drive results in a larger number of tracks, however a random write in a track may damage data in down-stream adjacent tracks.

1) an evaluation of log-structured translation performance on a wide variety of modern traces, showing that workloads run the gamut from log-friendly (i.e. a net decrease in seeks), log-sensitive (i.e. seek amplifications of as much as 10x or more) and log-agnostic, with small or no change in seek numbers.
2) characterization of the read/write behavior seen in these traces, showing that a small number of writes are responsible for many fragmented reads, and that seek length distributions are radically different after log-structured translation.
3) two novel mechanisms for reducing seek overhead—*opportunistic defragmentation* and *translation aware selective caching*—and a third mechanism based on modifications to existing drive read-ahead algorithms.
4) evaluation of these techniques and demonstration that they reduce or eliminate seek overhead for log-structured writes in almost all the examined workloads.

We stress the importance for SMR-based systems of this last contribution. SMR disks at present carry a dual performance penalty over conventional disk, incurring both (a) cleaning overhead and (b) additional seeks. Eliminating both overheads would allow the creation of systems with the capacity advantages of SMR (as compared to non-SMR disk) without the performance penalty.

Many of today's large storage systems (where SMR drives are targeted) are *archival* systems, accumulating data and never modifying or deleting it; in these systems a log-structured translation layer need never enter cleaning (garbage collection), as writes cease just as the disk fills and runs out of room. In these systems we can reasonably claim to be able to eliminate cleaning overhead; by applying the techniques proposed in this paper it appears possible to reduce or eliminate performance overheads due to disk seeks, potentially eliminating the SMR performance penalty entirely.

II. Background

Shingled magnetic recording decreases the track width of perpendicular magnetic recording by writing overlapping tracks, storing more data at the cost of losing the ability to perform random sector updates without losing data (see Figure 1). In particular, when overwriting a sector, data in the physically adjacent track is at risk of being corrupted or over-written. Although several other methods have been proposed (e.g. Caveat Scriptor [14], SMART [15], Virtual Guard [16]), devices shipped to date organize each platter in *zones*, where zones are separated by guard tracks wide enough to prevent adjacent-track corruption, allowing each zone to be written (or re-written) sequentially and independently of other zones. In *drive-managed* SMR devices an internal translation layer emulates a fully-rewriteable block device on top of these zones, while in *host-aware* and *host-managed* devices the zone structure and write constraints are exposed to the host.

The host-managed model provide by the Zoned Block Device extensions to SCSI and SATA is almost identical to the NAND flash model: a zone (cf. erase unit) consists of a sequence of sectors (pages) which must be written sequentially, after which the write pointer may be reset (the unit may be erased), losing access to data stored in that zone and allowing it to be re-written from the beginning again.

Existing translation layers for SMR [17], [11] have typically been very simple, logging updates to a reserved region of the disk (the *media cache*), and then merging them back to *data zones*, where they are stored in logical order, similar to mechanisms used in the simplest flash translation layers [18]. As a result almost all data is stored in LBA order, resulting in little or no read seek amplification, but at the price of high cleaning overhead.

An alternate approach would be to perform log-structured writes with a full block or extent map, as is done in high-performance flash translation layers [19], reducing cleaning overhead or avoiding it entirely in some archival applications. In this case read seek amplification becomes a more significant issue, as the address space defragmentation of simpler mechanisms, is eliminated, and the cleaning overhead is reduced, increasing the relative contribution of smaller overheads.

Disk model: We examine this second approach, where data is placed on disk in a physical order matching the temporal order in which it was written, with each write being directed to a *write frontier* which advances across the disk. We assume an infinite disk, ignoring cleaning overhead; for archival workloads cleaning may never be needed, and for traditional workloads cleaning performance has been extensively examined. Finally, we ignore specific device performance characteristics or geometry. We consider a seek to occur if an I/O operation starts at a sector other than that immediately following the previous I/O operation, and term it a read or write seek according to whether the second of the two operations is a read or write. Performance is expressed as *seek amplification*: the ratio of seeks (read, write, or total) for the log-structured system to seeks incurred on a conventional drive by the workload trace.

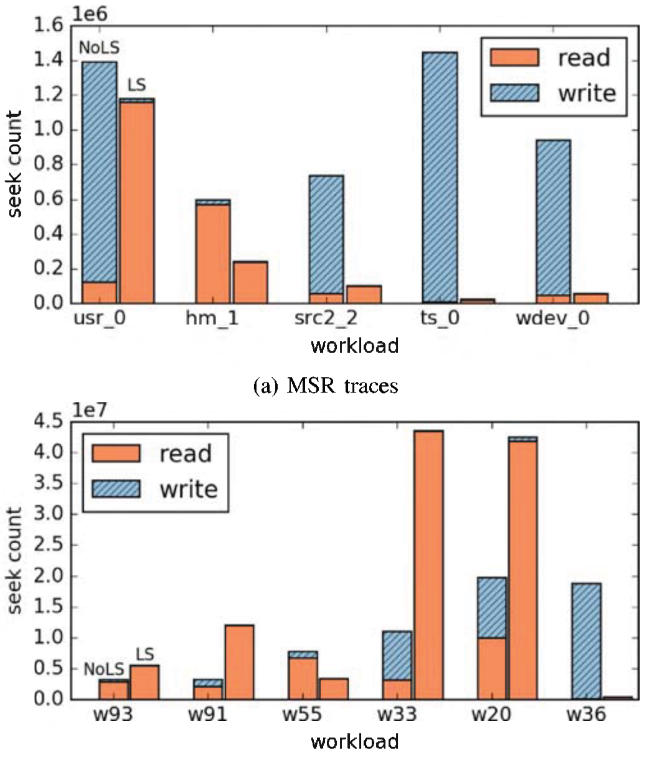
III. Seek amplification of log-structured writes

It is the workload that entirely determines the relative read seek increase (read seek amplification) due to log-structured writing under the disk model used here. As a thought experiment we can easily construct "toy" cases where read seeks are unaffected (or even reduced), and others where a large overhead is seen:

- Small file creation and access (decreased read seeks): By sequentially writing the blocks modified in creating a single-block file—block and inode bitmaps, directory and file inodes, directory and file data, and journal—it becomes possible to read all blocks needed for accessing that file (directory/file inodes and data blocks) with a single seek, rather than the four that would be typically required.
- Sequential read after random write (increased read seeks): If a large number of small random writes are performed to a large file (e.g. a database) , a single sequential read of that file will require approximately as many seeks as were saved in performing all the writes. If the file is read in its entirety N times, the net result will be an N-fold seek amplification.

We note that these cases may be characterized in terms of temporal and spatial ordering of reads and writes. In the first we have a set of operations which are not spatially sequential; however the temporal order of their writing is mimicked in the order in which they are read; in this case log structure eliminates read as well as write seeks. In the second case the write and read ordering is entirely unrelated, and (importantly) reads are ordered to improve performance. If the reads were scattered randomly then we would see a net reduction in seeks due to the near elimination of write seeks; however with sequential read, that savings is "paid back" in extra read seeks, once for each time the data is read sequentially. To explore the impact of read/write ordering on seek count in real-world workloads under log-structured writes, we perform some analysis using the the disk model explained in Section II.

Workloads: For our experiments we use two sets of block traces: the well-known MSR traces [20], from production Windows systems in the 2007-2008 time period, and a set of Linux and Windows traces collected more recently by CloudPhysics [21]. We sample the traces and select some that represent different I/O behavior in terms of read/write intensity and seek count in both log-structured and non-log-structured translation. Characteristics of these traces are shown in Table I.



(a) MSR traces

(b) CloudPhysics traces

Fig. 2: Read and write seek counts of a subset of (a) MSR and (b) CloudPhysics traces for both non-log-structured (NoLS) and log-structured (LS) translation. We consider a seek to occur if an I/O operation starts at a sector other than that immediately following the previous operation.

Results: Results may be seen in Figure 2, where we see read (orange) and write (blue) seek counts for selected traces for the log-structured (LS, right bar) and untranslated (NoLS, left bar) cases. As expected, write seeks are greatly reduced by log-structured writing, as all back-to-back writes are written sequentially regardless of LBA. For some traces (e.g. `src2_2, wdev_0, w36`) there is a modest increase in read seeks, resulting in an overall reduction in seek overhead. For others (e.g. `w91, w33, w20`) the increase in read seeks is huge, resulting in a large net seek amplification, of up to 5x in the case of `w91`. In a third group, the increase is significant but not overwhelming, as in `hm_1, w93, w55`. Overall we see that read seek amplification is highly workload-dependent, in some cases negligible or better (i.e. a gain), and in others risking significant performance degradation.

In Figure 3 we see log-structured translation overhead in terms of long ($\geq$500KB) seeks over time for a sub-set of these workloads; the values plotted are the absolute difference in seeks (log-structured minus original) for each unit of time. In order to show temporal patterns more clearly we ignore short seeks (less than +/- 500 KB), which have much noisier behavior. We see very strong temporal changes in seek amplification, many of them on a diurnal pattern (slightly

TABLE I: Workloads Characteristics.

	read count	write count	read volume (GB)	written volume (GB)	mean write size	OS (guest)
w84	655397	4158838	13.7	124.1	31.2	Red Hat Enterprise Linux 5
w95	1264721	2672520	30.3	27.7	10.8	Microsoft Windows Server 2008
w64	6434453	1023814	399.6	36.9	37.8	Microsoft Windows Server 2008 R2
w93	2928984	422470	115.7	11.4	28.3	Microsoft Windows Server 2003
w20	19652684	10189634	2353	332.8	34.25	Microsoft Windows Server 2003
w91	3147384	1169222	52.9	15.3	17.1	Microsoft Windows Server 2003
w76	258852	5817421	30.3	5.15	35.7	Microsoft Windows Server 2008 R2
w36	113090	18802536	399.6	4.02	141.8	Red Hat Enterprise Linux 5
w89	1536898	2089042	115.7	20.5	31.7	Microsoft Windows Server 2008 R2
w106	576666	2699254	2353	8.4	21.2	Microsoft Windows Server 2003 Standard
w55	7797622	1057909	35.8	18.4	18.2	Microsoft Windows Server 2008 R2
w33	7603814	8013607	238	241	31.6	Red Hat Enterprise Linux 5
usr_0	904483	1333406	35.3	13	10.2	Microsoft Windows
src2_2	350930	805955	22.7	39.2	51.1	Microsoft Windows
hm_1	580896	28415	8.2	0.5	19.9	Microsoft Windows
web_0	606487	1423458	17.3	11.6	8.5	Microsoft Windows
usr_1	41426266	3857714	2079.2	56.1	15.2	Microsoft Windows
wdev_0	229529	913732	2.7	7.1	8.2	Microsoft Windows
mds_0	143973	1067061	3.2	7.3	7.2	Microsoft Windows
rsrch_0	133625	1300030	1.3	10.8	8.7	Microsoft Windows
ts_0	316692	1485042	4.1	4.1	8	Microsoft Windows

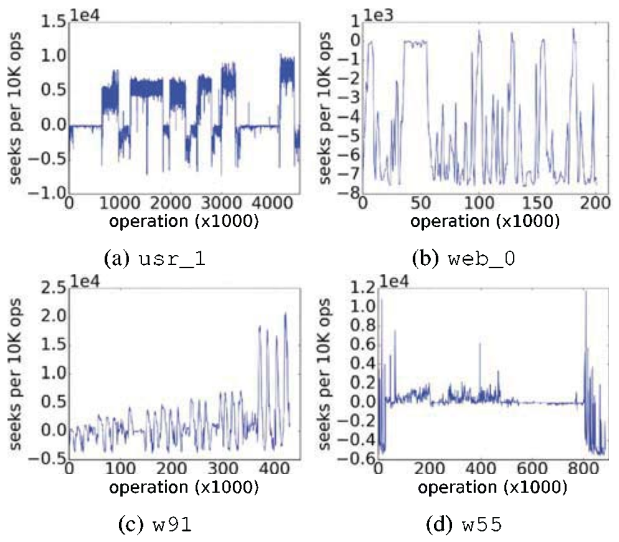


Fig. 3: Log-structured translation overhead (log-structured minus original) in terms of long (≥500KB) seeks over time for MSR ((a) usr_1 and (b) web_0) and CloudPhysics ((c) w91 and (d) w55) workloads.

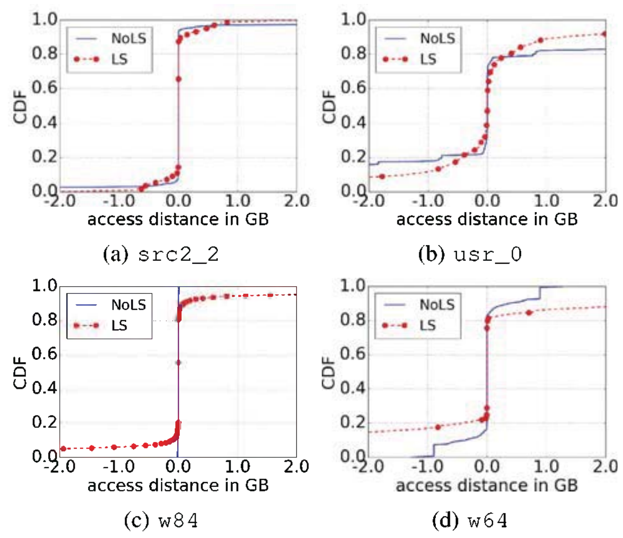


Fig. 4: CDF of access distances across (a) src2_2 and (b) usr_0 from MSR traces, and (c) w20 and (d) w36 from CloudPhysics traces for both non-log-structured (NoLS) and log-structured (LS) translation.

obscured by plotting vs. operation number rather than time). In other words, not only does seek amplification vary widely from trace to trace, but it varies widely over time within individual traces.

In considering the performance impact of seek amplification, we note that the cost of a seek varies significantly with its length. Very short seeks (e.g. 100s of KB) result only in a modest rotational delay, equivalent to the transfer time required to read the skipped sectors. Longer seeks incur both head seek and rotational delay; for seeks of substantially more than one track the rotational delay can be well-approximated by an average $1/2$ rotation (3-5ms), while the time required for head movement increases from a few ms to 25ms or more as the seek length increases.

CDFs of seek distances are shown in Figure 4. We note that there is a systematic difference between the older traces (src2_2, usr_0) and the newer ones (w84, w64): in the older traces a larger fraction of log-structured seeks (vs. non-log-structured) were between +1 GB and -1 GB, while for the newer traces less than half of the log-structured seeks fell within a range that includes virtually all of the seeks from the original block traces.

Actual seek lengths are dependent on details of how the data is written. In particular, since the traces include reads from sectors which were written before the period of trace collection, we need to assign a physical location for unwritten data. We assume this data is stored at a physical location corresponding to its LBA, and start the write frontier above the highest LBA found in the trace. As a result many seeks will be longer than they would be if all data was written log-structured from the beginning, biasing any computation of mean seek distance. However the CDFs in Figure 4 are restricted to a narrow range of LBA offsets, which would not be affected by placement of unwritten data.

From the results shown in Figures 2, 3, and 4 we can conclude the following about the effect of log-structured writing on seek behavior, at least for the traces available to the authors:

- Not all workloads incur read seek amplification from log-structured storage; however for some workloads the penalty is substantial.
- Workloads which have little or no read seek amplification when averaged over long time periods (e.g. `w55`) may have periods where they suffer significant read seek overhead.
- The effect of log-structured writes on the distribution of seek lengths varies by workload; more work is needed to fully understand it.

Finally we note that the MSR traces, which are over a decade old, exhibit substantially different behaviors than the newer CloudPhysics traces. We urge researchers to consider carefully whether results based on the MSR traces are applicable to more modern systems.

IV. Read seek reduction strategies

We next motivate and describe three techniques for reducing read seek amplification: *opportunistic degragmentation*, *translation aware look-ahead-behind prefetching*, and *translation aware selective caching*.

A. Opportunistic defragmentation

As arbitrary LBAs are written to the physical write frontier, a log-structured system becomes *fragmented*—i.e. the LBA space is represented by many non-contiguous physical extents, each corresponding to a small *fragment* or LBA range. This fragmentation is in turn responsible for read seek amplification, where a read for an LBA range may require access to multiple physical extents and thus seeks between those extents.

We can measure this fragmentation in two ways: *static* fragmentation and *dynamic* fragmentation. Static fragmentation is just a measure of how many physical extents have been created; it is also equivalent to the number of seeks which would be incurred by a sequential read of the entire LBA space. However we don't read the LBA space sequentially; some fragmentation may never effect a read operation in the workload, while other fragments may impact many read operations. By dynamic fragmentation we mean the fragmentation of a single read—i.e. the number of non-contiguous physical extents which would need to be fetched to fulfill that read request.

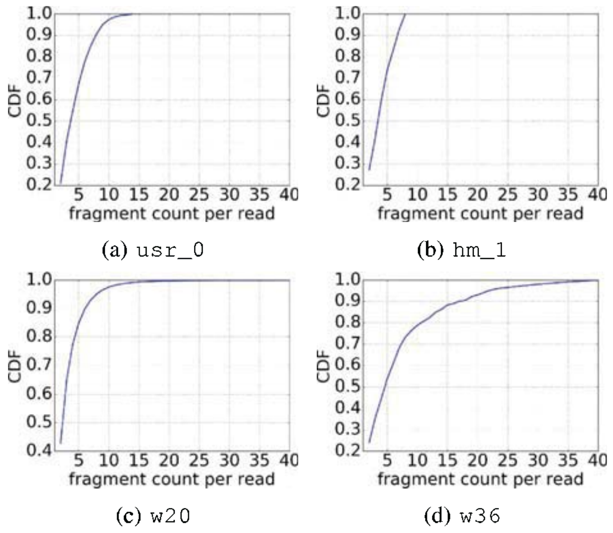


Fig. 5: CDF of fragmented reads for (a) `user_0`, (b) `hm_1`, (c) `w20` and (d) `w36`.

In Figure 5 we see CDFs for dynamic fragmented read operations (i.e. ignoring un-fragmented reads) for several traces. Rather than being evenly distributed, in each case the bulk of the fragments are found in a small fraction of the read operations: for `usr_0`, `hm_1` and `w20` over half of the fragments are found in about 20% of the operations, while for w36 the disparity is even higher.

The idea of opportunistic defragmentation (Algorithm 1) is to take advantage of the fact that read operations reorder fragmented data before returning it, allowing us to eliminate heavily-fragmented sections of data by writing back a defragmented version at the cost of only a single seek (to the write frontier) and a sequential write.

Extensive defragmentation (e.g. as performed by simple translation layers in SMR drives) would likely eliminate almost all read seek amplification, as it would in effect turn a log-structured (i.e. temporally-ordered) system back into a spatially-ordered one at run time; however the overhead of doing so would be extremely high. Opportunistic defragmentation instead focuses on fragments which actually matter - because they impact a read - and takes advantage of work which already needed to be done in order to fulfill a read request.

We see this in operation in Figure 6. The LBA range 1..6

Algorithm 1: Opportunestic defragmentation

```
while True do
    IO ← ReceiveIO()
    if IO == read then
        DoRead(IOextent)
        if FragmentedRead(IOextent) == True then
            WriteAtLogHead(IOextent);
        end
    end
end
```

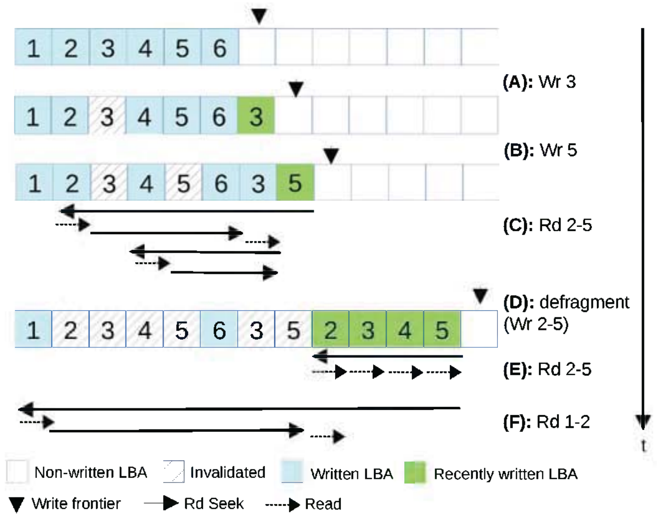


Fig. 6: State of the log along with required seeks for a sequence of read/write operations under log-structured translation showing how *opportunistic defragmentation* reduces extra seeks but also can impose overhead. t_A and t_B: LBA 3 and 5 are updated resulting in fragmentation. t_C: sequential read operation of LBA range 2..5 incurs three additional seeks. t_D: *opportunistic defragmentation* re-writes the read LBA range (2...5) to the log head. t_E: LBA range 2..5 is re-read but this time with no additional seek. t_F: A read to LBA range 1..2 causes an extra seek as a results of using *opportunistic defragmentation*.

is contiguous at the start, but is then fragmented by writes to LBAs 3 and 5. A read of the LBA range 2..5 incurs three extra seeks due to this fragmentation; however if we then defragment the range by writing the data back to the write frontier, a subsequent read of the same range incurs no extra seeks. We note that opportunistic defragmentation does not come for free; it incurs an additional seek to the write frontier and transfer time to re-write the data, and its use of free space will eventually necessitate running the cleaning algorithm with its attendant overheads. We can reduce these overheads by restricting the times when defragmentation is performed, specifically by defragmenting only regions with N or more fragments, or waiting until a fragmented range has been accessed k or more times.

B. Translation aware look-ahead-behind-prefetching

Although the file system and block layers in most operating systems go to sometimes great lengths to ensure that data is allocated contiguously, it does not always get written sequentially to disk. This can be clearly seen in a section from the MSR hm_1 trace shown in Figure 7a, where a series of contiguous LBA ranges are written in various orders: descending, ascending within small chunks with the chunks descending, interleaved ascending, etc. Examples in the CloudPhysics traces are less extreme; however in Figure 7b we see an example of small-scale randomness which is

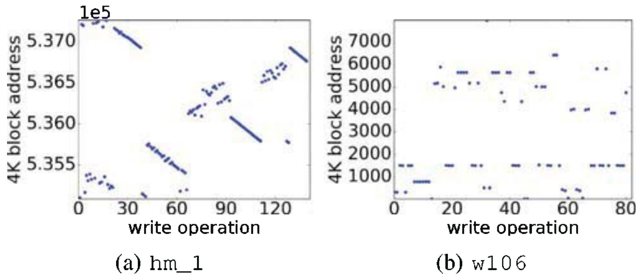


Fig. 7: Examples of highly non-sequential LBA accesses from traces (a) hm_1 and (b) w106.

more common in those workloads. In addition another source of non-sequentiality occurs when multiple sequential write streams are interleaved on their way to the disk.

In a conventional disk system, none of these write patterns would be a big deal. On further examination one finds that the sequences of descending I/Os in Figure 7a were dispatched almost simultaneously (e.g. several dozen I/Os over the course of a few microseconds), and that they actually completed in ascending LBA order. In other words, the disk subsystem was able to re-order the I/Os on the fly and then re-write them sequentially with almost no overhead.

In a simple log-structured system these I/Os would instead be written in an unwanted order, preserving it so that it could interfere with later reads of the same LBA range. Rather than long-distance seeks, the risk here is of *missed rotations*, where a read of physical location N+1 followed by a read of location N requires an entire disk rotation to "back up" to the preceding LBA. To examine how often this might occur, we measure *mis-ordered writes*, writes with LBAs sequentially following a write in the near future. ("near future" being defined as "within the next 256 KB of write operations). We see values for this metric for several traces in Figure 8; in several cases as many as one in 25 (w106) or one in 20 (src2_2) of writes are mis-ordered.

We consider the mis-ordered writes phenomenon and offer *translation aware look-ahead-behind prefetching*. Although techniques used in this method (i.e., look-ahead and look-behind prefetching [22]) are not novel, employing them

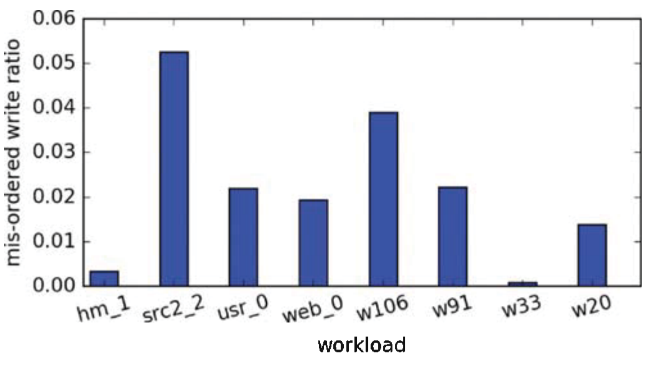


Fig. 8: Mis-ordered writes within the distance of 256KB.

upon accessing fragments could reduce future read seeks significantly. *Look-behind* prefetching is a standard caching technique used in disk drives; it is similar to *look-ahead prefetching*, which is thought to be implemented on every modern drive; look-behind prefetching is no more difficult to implement, but may be less commonly implemented due to limited utility with conventional workloads. For log-structured systems, however, we argue that look-behind prefetching serves to effectively reduce missed rotations due to out-of-order writes.

To describe look-behind, we first describe lookahead: after completing a read up through LBA N, if there is no pending request requiring seeking to another track, the drive will read LBA $N+1, N+2, \dots N+k$ into cache, in the hopes that they will be requested in the near future. This is quite likely, e.g. in the common pattern of single-threaded sequential read, where the read request for block $N+1$ is not received until after the read completion for block N has propagated up through the operating system and the next read request could be issued. In other words, lookahead consists of continuing to read for some length of time *after* all the current requests are finished.

In contrast, look-behind entails reading LBAs *before* the next requested LBA. On receiving a read request for LBA N, the disk seeks to the appropriate track; once the read head is centered, it reads data into cache up through LBA $N-1$, then reads N and completes the request. In other words, in look-behind the drive bets that a read to N will be followed by a read to $N-1$; implementing it consists of reading for some length of time *before* the next request begins. Although look-behind provides some performance improvement with conventional workloads, it is limited; however it is just the thing to prevent missed rotations due to mis-ordered writes in a log-structured system. In our trace-driven workload analysis under log-structured translation we evaluate a combination of lookahead and look-behind prefetching, as described in Algorithm 2.

In Figure 9 we see how the combination of look-behind and look-ahead prefetching can reduce the number of seeks. As depicted, at the initial state LBA 1 to LBA 6 are already written to the log. Thereafter, LBA 3, 2 and 4 are updated

Algorithm 2: Look-ahead-behind-prefetching

```
while True do
    IO ← ReceiveIO()
    if IO == read then
        for lba in IOextent do
            pba ← RetrievePBA()
            if FragmentedRead() == True then
                PreFetch(fetchRegion);
                DoRead(pba);
                PostFetch(fetchRegion);
            end
            else
                DoRead(pba);
            end
        end
    end
end
```

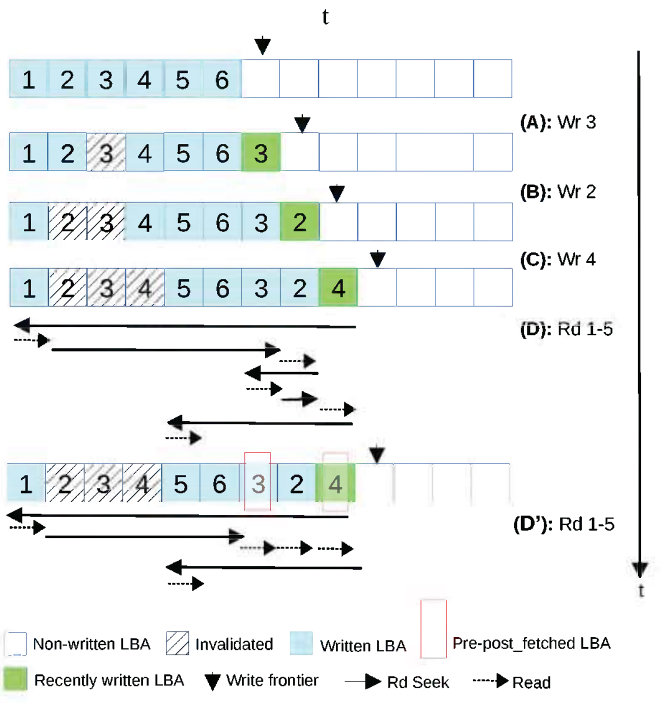


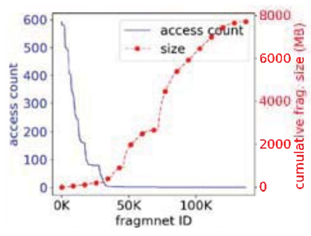
Fig. 9: State of the log along with required seeks for a sequence of read/write operations under log-structured translation showing how *translation aware look-ahead-behind prefetching* eliminates extra seeks. t_A, t_B and t_C: LBAs 3, 2 and 4 are updated. t_D: LBA range 1..5 is read resulting in four additional seeks due to fragmentation. $t_{D'}$: another read to the same LBA range (1..5) is performed yet with no additional seeks as look-ahead-behind prefetching is enabled; LBA 3 and 4 are prefetched upon reading LBA 2.

from t_A to t_C. At t_D a read operation to LBAs 1 to 5 takes place; As seen, a read (without prefetching) leads to a total of 5 seeks, of which 2 are due to fragmentation. However, a shown at $t_{D'}$ look-ahead/look-behind reduces the overall seeks to 3; in the path to read LBA 2, LBA 3 and LBA 4 are prefetched.

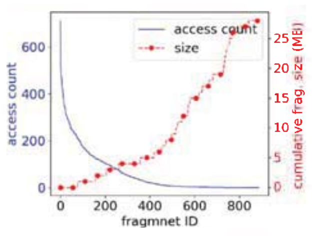
C. Translation aware selective caching

Finally, by caching small number of selected fragments in RAM, we are able to eliminate a significant portion of seeks which would not be omitted by a vastly large buffer cache. Unless done carefully this is unlikely to work, as operating systems running on modern servers typically devote gigabytes of RAM to caching on-disk data—orders of magnitude more than are available for on-drive caching. (and although more RAM is available on the host, using it would directly compete with the buffer cache.)

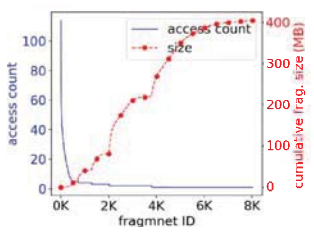
We take advantage of the fact that (for the workloads examined in this work) access to fragments on disk is highly skewed, with a small number of fragments responsible for a large number of seeks. By caching just those fragments we are able to make the best use of a small cache, eliminating



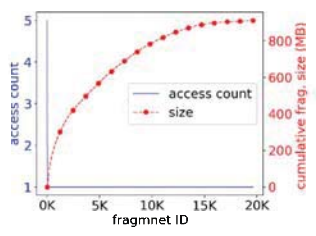
(a) usr_1



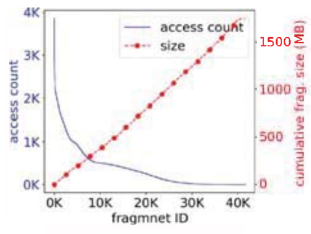
(b) hm_1



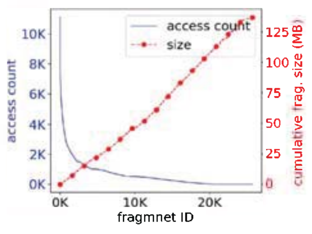
(c) web_0



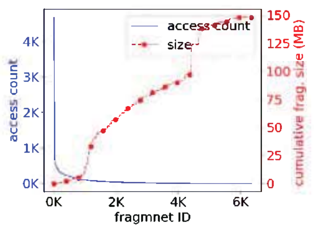
(d) src2_2



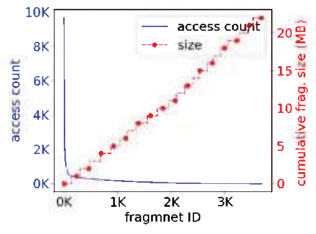
(e) w20



(f) w33



(g) w55



(h) w106

Fig. 10: X axis: fragments are sorted by read access count from the most to least popular. Blue solid line: access count of the sorted fragments. Red dashed line: CDF of required cache size to store the fragments.

large numbers of seeks while avoiding cache pollution from data which is unlikely to incur read seeks.

In Figure 10 we see fragment access statistics for a range of traces; fragments (solid blue line) are sorted by access count (y axis) from most to least popular, while the cumulative cache size needed to hold the corresponding fragments is shown in dashed red. In each case we see that the fragments responsible for a large majority of accesses (and thus additional seeks) add up to a few 10s of MB or less, well within the size that may be readily cached within an on-host driver. (although such a cache might be a bit large for current drive controllers with 128 MB DRAM, drives are starting to move to 256 MB DRAM chips for cost and supply reasons.)

Our *translation aware selective caching* algorithm is seen in Algorithm 3; data from fragmented reads is cached with LRU eviction, eliminating read seeks in those cases where the data sought may be found in cache first.

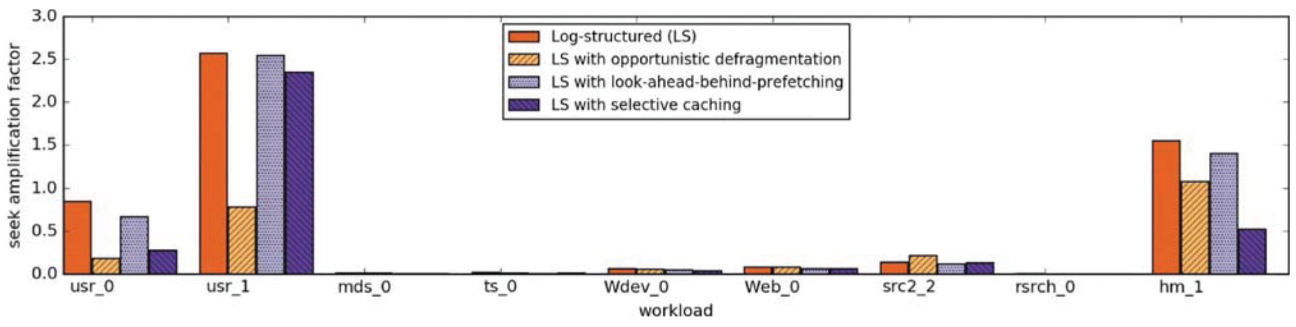
V. Evaluation

Using the disk model described in Section II, we evaluate the three proposed techniques, *opportunistic defragmentation*, *translation aware look-ahead-behind prefetching* and *translation aware selective caching*; we count the number of seeks generated in non-log-structured fashion and accordingly report the seek amplification factor (SAF) under log-structured translation as well as when such techniques are combined with it. We run our experiments against workloads sampled from both MSR [20] and CloudPhysics [21] traces; workloads were selected to represent different behaviors in terms of read/write intensity and seek count. For the experiments related to *translation aware selective caching* technique, the cache size is set to 64MB.

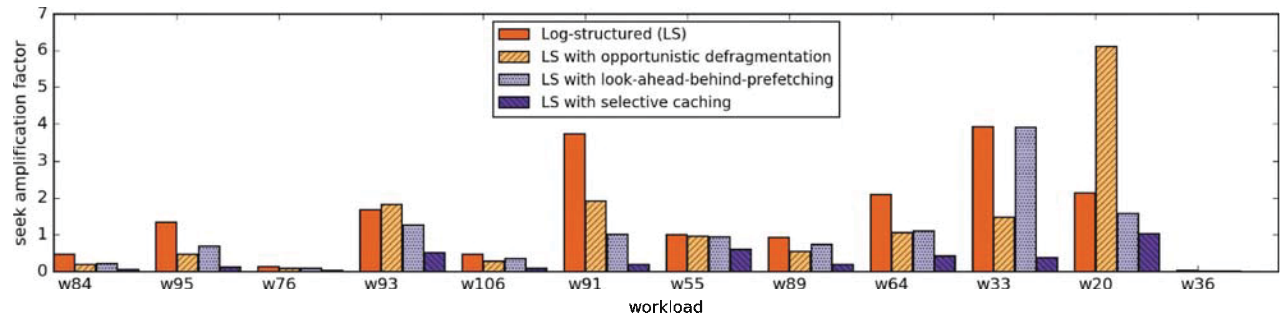
Results may be seen in 11; for MSR workloads as seen in Figure 11a, when log-structured translation is used, all studied workloads except for usr_1 and hm_1 show a seek amplification factor of less than one, meaning a reduction in seek count. Such workloads are write-intensive (see Table I) and potentially could benefit more from log-structured translation (all back-to-back writes are written sequentially requirung no seek). Contrary to MSR workloads, as observed in Figure 11b, the majority of CloudPhysics workloads suffer from a SAF of greater than one when log-structured translation is used. As reported in Table I, these workloads are not write-intensive and therefore benefit less from back-to-back write operations. As a result, additional seeks incurred by fragmented reads leads to a greater seek count compared to non-log-structured fashion.

Algorithm 3: Selective caching on reads

```
while True do
    IO ← ReceiveIO()
    if IO == read then
        if FragmentedRead() == True then
            for fragment within IOextent do
                if ChechkCache(fragment) == True then
                    ReadCache(fragment);
                end
                else
                    ReadDisk(fragment);
                    WriteCache(fragment);
                end
            end
        end
        else
            ReadDisk();
        end
    end
end
```



(a) MSR traces



(b) CloudPhysics traces

Fig. 11: Seek amplification factor of log-structured translation (left bar), log-structured translation when combined with *opportunistic defragmentation* (second bar from left), *translation aware look-ahead-behind prefetching* (third bar) and *translation-aware selective caching* (last bar) for (a) a subset of MSR and (b) CloudPhysics workloads.

All the proposed techniques except for *opportunistic defragmentation* result in SAF improvement across every workload; as stated in IV-A, *opportunistic defragmentation* comes at the cost of an additional seek which may result in greater seek amplification factor. This can be seen in Figure 11 in some of the studied workloads such as `src2_2`, `w93` and `w20`; in `w20` case we observe that SAF is worsen by 2.8x.

Depending on workload, *translation aware look-ahead-behind prefetching* has a different impact on SAF. While it shows a marginal improvement for some cases (e.g., `usr_1`, `hm_1`, `w55` and `w33` with less than 1% of improvement) it significantly improves SAF in other cases including `w84`, `w95` and `w91` (up to 3.7x improvement); these latter cases probably contain a considerable amount of *mis-ordered writes*.

Translation-aware selective caching on average, performs the best among all proposed techniques. As seen in Figure 11, it results in the lowest SAF in all workloads except for a few such as `usr_1` and `src2_2`. In the best case for the `w91` workload, it reduces the SAF caused by log-structured translation from 3.7 to 0.2.

VI. Related work

Following the original log-structured file system paper [3] a long series of works have examined the write amplification due to cleaning, both for disk-based systems [5], [4] and for flash [23], [24], [6], [7], [8], [9]. In general these efforts have focused on simplified random workload models; write amplification for workloads has primarily been addressed in the context of evaluating cleaning and translation algorithms.

Besides, a number of file systems which also write in part or totally in a log-structured fashion have been proposed for SMR disk; examples include SFS [25], HiSMRfs [26], and SMRBD [27]. Other work has examined block translation layers, including SMART [15], Virtual Guard [16] and Cassuto's early work on indirection systems for SMR disk [17].

None of the prior work, however, has addressed the inherent seek overhead (specifically read seek overhead caused by fragmentation) in log-structured storage systems, which is addressed by this work. There is only a small number of work that offers general solutions to overcome the seek overhead in log-structured systems [28], [12], [29], [30].

Zhang and Ghose proposed a combined journaling/log-structured file system for reduced seek overhead [28]. Their approach is orthogonal to ours as it is offered at the file system level while our proposed techniques try to resolve the issue at the translation layer level. This also applies to Wang's WOLF file system [12] which went to great lengths to separate hot and cold data (for cleaning cost reduction) without incurring seek overhead due to switching between write frontiers. Similar to Wolf [12], Zhang and Liu [29], leverage an external intermediate buffer at the system level to handle the defragmentation process and be able to rewrite files defragmented sectors on disk.

There are also some other work proposed to reduce seeks, yet not for log-structured file system. For instance, Jernigan and Quinn proposed a two pass defragmentation mechanism for the FAT file system [30]. In contrast to our proposed techniques which is transparent to the upper layers (OS) and is triggered automatically whenever a fragment is accessed, Jernigan's and Quinn's proposed approach requires a command from upper layers to initiate the process.

VII. Conclusion

We present the first examination of disk seek overheads in log-structured storage systems such as SMR disk translation layer, showing the effect of workloads on this issue. We show that a fraction of workloads incur large numbers of additional read seeks when writes are performed in log-structured fashion.

We describe and evaluate three techniques for addressing these overheads: two novel methods—*opportunistic defragmentation* and *translation aware selective caching*—and *translation aware look-ahead-behind prefetching*, an existing disk buffering technique, showing that read seek overheads are substantially reduced in all cases, resulting in little to no overhead for log-structured disk storage in almost all cases.

Acknowledgment

We would like to thank Irfan Ahmad and CloudPhysics for the use of their traces, and our shepherd, Hung-Wei Tseng, and the anonymous reviewers for their valuable suggestions. This work was supported in part by a NetApp Faculty Fellowship.

References

[1] R. Finlayson and D. Cheriton, *Log files: an extended file service exploiting write-once storage*. ACM, 1987, vol. 21, no. 5.

[2] J. Ousterhout and F. Douglis, "Beating the I/O bottleneck: a case for log-structured file systems," *ACM SIGOPS Operating Systems Review*, vol. 23, pp. 11–28, Jan. 1989, aCM ID: 65765.

[3] M. Rosenblum and J. K. Ousterhout, "The design and implementation of a log-structured file system," in *13th ACM symposium on Operating systems principles*. Pacific Grove, California, United States: ACM, 1991, pp. 1–15. [Online]. Available: http://dx.doi.org/10.1145/121132.121137

[4] J. T. Robinson, "Analysis of steady-state segment storage utilizations in a log-structured file system with least-utilized segment cleaning," *SIGOPS Operating Systems Review*, vol. 30, no. 4, pp. 29–32, Oct. 1996.

[5] J. Menon and L. Stockmeyer, "An age-threshold algorithm for garbage collection in log-structured arrays and file systems," in *High Performance Computing Systems and Applications*, J. Schaeffer, Ed. Springer US, 1998, vol. 478, pp. 119–132.

[6] S. Baek, J. Choi, D. Lee, and S. H. Noh, "Model and validation of block cleaning cost for flash memory," in *Int'l Conf. on Embedded Computer Systems: Architectures, Modeling, and Simulation*. Samos, Greece: Springer-Verlag, 2007, pp. 46–54.

[7] L. Xiang and B. Kurkoski, "An improved analytic expression for write amplification in NAND flash," in *2012 International Conference on Computing, Networking and Communications (ICNC)*. Maui, Hawaii: IEEE, 2012, pp. 497–501.

[8] P. Desnoyers, "Analytic Models of SSD Write Performance," *ACM Transactions on Storage*, vol. 10, no. 2, pp. 8:1–8:25, Mar. 2014.

[9] B. Van Houdt, "A Mean Field Model for a Class of Garbage Collection Algorithms in Flash-based Solid State Drives," in *Proceedings of the ACM SIGMETRICS/International Conference on Measurement and Modeling of Computer Systems*, ser. SIGMETRICS '13. New York, NY, USA: ACM, 2013, pp. 191–202.

[10] M. Shafaei, M. H. Hajkazemi, P. Desnoyers, and A. Aghayev, "Modeling drive-managed smr performance," *ACM Transactions on Storage (TOS)*, vol. 13, no. 4, p. 38, 2017.

[11] A. Aghayev, M. Shafaei, and P. Desnoyers, "Skylighta window on shingled disk operation," *ACM Transactions on Storage (TOS)*, vol. 11, no. 4, p. 16, 2015.

[12] J. Wang and Y. 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*, ser. FAST '02. Berkeley, CA, USA: USENIX Association, 2002. [Online]. Available: http://dl.acm.org/citation.cfm?id=1083323.1083329

[13] M. Jung and M. Kandemir, "An Evaluation of Different Page Allocation Strategies on High-Speed SSDs," in *Proc. of the 4th USENIX Workshop on Hot Topics in Storage and File Systems (HotStorage'12)*, Boston, MA, Jun. 2012.

[14] S. Kadekodi, S. Pimpale, and G. A. Gibson, "Caveat-scriptor: Write anywhere shingled disks." in *HotStorage*, 2015.

[15] W. He and D. H. Du, "Smart: An approach to shingled magnetic recording translation." in *FAST*, 2017, pp. 121–134.

[16] M. Shafaei and P. Desnoyers, "Virtual guard: A track-based translation layer for shingled disks," in *9th USENIX Workshop on Hot Topics in Storage and File Systems (HotStorage 17)*. USENIX Association, 2017.

[17] Y. Cassuto, M. A. Sanvido, C. Guyot, D. R. Hall, and Z. Z. Bandic, "Indirection systems for shingled-recording disk drives," in *Mass Storage Systems and Technologies (MSST), 2010 IEEE 26th Symposium on*. IEEE, 2010, pp. 1–14.

[18] S.-W. Lee, D.-J. Park, T.-S. Chung, D.-H. Lee, S. Park, and H.-J. Song, "A log buffer-based flash translation layer using fully-associative sector translation," *ACM Transactions on Embedded Computing Systems (TECS)*, vol. 6, no. 3, p. 18, 2007.

[19] A. Gupta, Y. Kim, and B. Urgaonkar, *DFTL: a flash translation layer employing demand-based selective caching of page-level address mappings*. ACM, 2009, vol. 44, no. 3.

[20] D. Narayanan, A. Donnelly, and A. Rowstron, "Write off-loading: practical power management for enterprise storage," in *Proceedings of the 6th USENIX Conference on File and Storage Technologies*. San Jose, California: USENIX Association, 2008, pp. 1–15.

[21] C. A. Waldspurger, N. Park, A. Garthwaite, and I. Ahmad, "Efficient MRC Construction with SHARDS," in *13th USENIX Conference on File and Storage Technologies (FAST 15)*. Santa Clara, CA: USENIX Association, 2015, pp. 95–110.

[22] B. Jacob, S. Ng, and D. Wang, *Memory systems: cache, DRAM, disk*. Morgan Kaufmann, 2010.

[23] S. Boboila and P. Desnoyers, "Performance models of flash-based Solid-State drives for real workloads," in *IEEE Symposium on Massive Storage Systems and Technologies*. Denver, Colorado: IEEE, Jun. 2011.

[24] X. Hu, E. Eleftheriou, R. Haas, I. Iliadis, and R. Pletka, "Write amplification analysis in flash-based solid state drives," in *SYSTOR 2009: The Israeli Experimental Systems Conference*. Haifa, Israel: ACM, 2009, pp. 1–9.

[25] D. Le Moal, Z. Bandic, and C. Guyot, "Shingled file system host-side management of shingled magnetic recording disks," in *Consumer Electronics (ICCE), 2012 IEEE International Conference on*. IEEE, 2012, pp. 425–426.

[26] C. Jin, W.-Y. Xi, Z.-Y. Ching, F. Huo, and C.-T. Lim, "Hismrfs: A high performance file system for shingled storage array," in *Mass Storage Systems and Technologies (MSST), 2014 30th Symposium on*. IEEE, 2014, pp. 1–6.

[27] R. Pitchumani, J. Hughes, and E. L. Miller, "Smrdb: key-value data store for shingled magnetic recording disks," in *Proceedings of the 8th ACM International Systems and Storage Conference*. ACM, 2015, p. 18.

[28] Z. Zhang and K. Ghose, "yfs: A journaling file system design for handling large data sets with reduced seeking," in *Proceedings of the FAST '03 Conference on File and Storage Technologies, March 31 - April 2, 2003, Cathedral Hill Hotel, San Francisco, California, USA*, 2003.

[29] J. D. Ji Zhang, Hain-Ching Liu, "Disk drive storage defragmentation system," 2006, uS Patent 11,353,370. [Online]. Available: http://www.google.it/patents/US4741207

[30] S. D. Q. Richard P. Jernigan, "Two-pass defragmentation of compressed hard disk data with a single data rewrite," 1994, uS Patent 5,574,907A. [Online]. Available: http://www.google.it/patents/US4741207