Classifying Data to Reduce Long-Term Data Movement in Shingled Write Disks

STEPHANIE N. JONES, University of California, Santa Cruz
AHMED AMER, Santa Clara University
ETHAN L. MILLER, DARRELL D. E. LONG, REKHA PITCHUMANI, and CHRISTINA R. STRONG, University of California, Santa Cruz

Shingled magnetic recording (SMR) is a means of increasing the density of hard drives that brings a new set of challenges. Due to the nature of SMR disks, updating in place is not an option. Holes left by invalidated data can only be filled if the entire band is reclaimed, and a poor band compaction algorithm could result in spending a lot of time moving blocks over the lifetime of the device. We propose using write frequency to separate blocks to reduce data movement and develop a band compaction algorithm that implements this heuristic. We demonstrate how our algorithm results in improved data management, resulting in an up to 45% reduction in required data movements when compared to naive approaches to band management.

CCS Concepts: • **Information systems** → **Magnetic disks**; • **Hardware** → **Memory and dense storage**; • **Computing methodologies** → Modeling and simulation

Additional Key Words and Phrases: Storage, shingled write disks, shingled magnetic recording drives, data placement

ACM Reference Format:
Stephanie N. Jones, Ahmed Amer, Ethan L. Miller, Darrell D. E. Long, Rekha Pitchumani, and Christina R. Strong. 2016. Classifying data to reduce long-term data movement in shingled write disks. ACM Trans. Storage 12, 1, Article 2 (February 2016), 17 pages.
DOI: http://dx.doi.org/10.1145/2851505

1. INTRODUCTION

Shingled magnetic recording (SMR) disks are devices that increase the storage density of traditional disk media by writing overlapping wide tracks of data, resulting in a shingle-like arrangement of the tracks. This presents us with a problem when overwriting previously written data, as it cannot be done without overwriting adjacent tracks. Overlapping tracks are therefore arranged into bands, and space reclamation is done at a band level.

SMR disks therefore must employ band compaction to reclaim bands containing overwritten data, a concept similar to log-structured file system (LFS) cleaning. A

This work is supported in part by the National Science Foundation under award IIP-1266400 and industrial members of the Center for Research in Storage Systems. A. Amer and D. D. E. Long were supported in part by the National Science Foundation under awards CCF-1219163 and CCF-1217648, by the Department of Energy under award DE-FC02-10ER26017/DE-SC0005417, by the industrial members of the Storage Systems Research Center, and by a gift from Wells Fargo.
Authors' addresses: S. N. Jones, A. Amer, E. L. Miller, D. D. E. Long, R. Pitchumani, and C. R. Strong, 1156 High Street, Santa Cruz, CA 95064.

DOI: http://dx.doi.org/10.1145/2851505

number of bands are read in their entirety, and the valid blocks are compacted to a fewer number of bands, creating bands of free space. However, this can easily become expensive when the band compaction strategy is frequently moving data. We propose an algorithm for band compaction specifically designed to reduce long-term data movement.

The more time a shingled disk spends in band compaction, the more it is wasting resources. In the worst case, the system may need to block for I/O until band compaction completes, making band compaction the bottleneck in the system and drastically reducing system responsiveness during that time. It is therefore necessary to develop band compaction strategies that mitigate the cost to the system. We claim that by separating blocks that are frequently updated from blocks that are less frequently (or never) updated, we can move fewer blocks in the long term. By applying this to band compaction, we have developed an algorithm that seeks to reduce long-term data movement, potentially increasing the benefits of eliminating unnecessary activity and data movement in addition to reducing the overheads of employing SMR media.

To reduce data movement, we are looking at using *write heat* as a metric to guide band compaction. For the purposes of this article, write heat is measured in the frequency of writes to an LBA. By separating incoming write data based on write heat, we can reduce the likelihood of a band containing a mixture of hot and cold data. Bands that contain a mix of hot and cold data are more expensive to compact than bands that contain only hot or only cold data. In addition, having a large number of mixed bands can result in band compaction occurring at a higher frequency.

For our purposes, we define data blocks to be "hot" if they have been overwritten at least once and "cold" otherwise. Bands that contain only cold data will be selected for compaction more frequently and have more data that must be copied. Normally, this would cause concern, but if the blocks are cold, they are expected to be long lived and less likely to leave holes in a band. Bands that contain only hot data will be compacted less often and will have less data that needs to be copied per hot band. However, these blocks are volatile and are likely to be invalidated in the future leaving holes in the bands. Classifying data as hot or cold and placing it accordingly helps to reduce both the number of bands read before performing compaction as well as the overall amount of data copied during band compaction.

There are currently three types of SMR disks: drive managed, host managed, and host aware) [Aghayev and Desnoyers 2015]. Drive-managed SMR disks present a block-based interface that is currently used in nonshingled hard drives. It employs a shingle translation layer to map the blocks to their particular band. A host-managed drive requires data sent to the drive to be written sequentially to its bands. Finally, a host-aware drive has bands to which it prefers to receive sequential writes, but it also maintains an internal shingle translation layer for any incoming nonsequential writes. The work presented in this article is best suited for drive-managed SMR disks, but it can also work for host-aware and host-managed SMR disks. To simulate SMR disks, we implemented an LFS-based translation layer with a block-based API. LFS segments and SMR bands are similar in concept, and band compaction in an SMR disk parallels segment cleaning in an LFS. However, due to the unique properties of SMR disks, directly porting an LFS is not feasible.

Within this system, we replayed traces from the MSR Cambridge dataset [Narayanan et al. 2008]. We applied our algorithm and compared it against a greedy algorithm that focuses on compacting the emptiest bands. Our algorithm gives priority to the emptiest bands but also weights how much of the band consists of cold data when selecting bands for band compaction. We have found that the optimal value for the weight on cold varies depending on the workload. We have also identified several weight values that, while not optimal, provide a reduction in data movement.

2. RELATED WORK

A log-structuring of data, possibly through an LFS or block remapping layer, seems suitable for SMR disks. However, for an LFS to work for SMR disks, data movement needs to be minimized, as writing to a band is not the same as writing to the log in the original LFS. Specifically, you cannot plug the holes in any of the bands, and you cannot update in place. The goal of the original LFS was focused on creating large continuous spaces in which to write the log. This directly conflicts with our goal of maximizing the utilization of a band while minimizing data movement (thereby reducing the overall performance overhead of employing SMR disks).

Since SMR technology was developed after the original LFS, there are characteristics of SMR disks that the original LFS cannot fully utilize. Shingled disks may have a random access zone (RAZ) that can be used to hold metadata and other frequently modified data. Metadata updates in an LFS are stored, like everything else, in the log. This can result in frequent holes in LFS segments, forcing cleaning to happen more often. Unless the segment is empty, segment cleaning results in data movement; an increase in cleaning is contradictory to our goal.

2.1. Log-Structured File Systems and Their Successors

The LFS is a file system optimized for writing [Rosenblum and Ousterhout 1992]. It assumes that main memory will satisfy more read requests as the size of main memory increases. This means that most requests that make it to disk will be write requests. The LFS appends all modifications to on-disk data to the end of the log. By appending to the end of the log, the LFS is able to maximize disk bandwidth for writing. The log is divided into segments, and a garbage collection policy is implemented to periodically clean the segments. The LFS implements a cost-benefit policy for segment cleaning, shown in the following formula:

$$\frac{\text{cost}}{\text{benefit}} = \frac{\text{free space generated} \times \text{age of data}}{\text{cost}}.$$

The free space generated is measured by how much of the segment still contains live data. The age of the data is the age of the most recently modified block in the segment, meaning that heat is calculated on a per-segment basis. The cost of cleaning the segment is measured as the cost to read the segment plus the cost of writing the live data elsewhere. A problem with the cost-benefit policy is that if there is one recently modified block but the rest of the data in the segment is very old, the age calculation can be misleading.

Although an LFS segment and an SMR disk band are similar, our definition of heat is different. The original LFS makes two major assumptions: first, that heat is related to the recency of write, and second, that blocks within a segment are written and modified together. We chose to begin by looking at heat as the frequency of a write to an LBA. This allows us to give data a value for heat that can later be changed with age rather than a static “hot” or “cold” assignment. In addition, rather than considering heat on a segment basis, we calculate heat on a per-LBA basis. In this way, we can designate heat for specific logical locations and assign blocks to hot or cold bands as they are being written.

BSD-LFS took the original design of LFS and modified it to work with UNIX FFS [Seltzer et al. 1993]. The authors evaluated BSD-LFS using three workloads, the most notable of which was the TPCB benchmark. When testing BSD-LFS using the TPCB benchmark with 85% disk utilization, the cleaner was constantly running and copying large amounts of data into new segments. Blackwell et al. [1995] developed a simple heuristic to reduce the overhead found in BSD-LFS. They showed that 97% of all cleaning in LFS during their tests could be done in the background. The authors used a simple heuristic of whether the disk had been idle for 2 seconds to signal that the segment cleaner should begin running. Such a technique is complimentary to our algorithm and could be applied to SMR disks to compact fragmented bands in the background rather than waiting for the disk to run out of free bands and compacting on demand.

Matthews et al. [1997] demonstrated how to overcome the segment cleaning problem at higher disk utilizations. They present an adaptive cleaning mechanism that chooses either full segment cleaning, as in LFS, or their technique of *hole-plugging*. Hole-plugging reads in segments to be cleaned and fills holes found in other segments with parts of the just-read segment. Traditional LFS segment cleaning provides the best performance until the disk utilization reaches 80% to 85%, at which point hole-plugging provides superior performance. It is important to note that on a standard disk, the reason hole-plugging becomes better is because at 80% to 85% utilization, traditional LFS segment cleaning drops significantly in performance, as shown in BSD-LFS. Since hole-plugging is not generally usable in SMR disks, with the possible exception of within the RAZ, this work is supplementary at best.

PROFS is a data reorganization scheme that is aimed at improving I/O performance for LFS on drives with zone-bit recording (ZBR) technology [Wang and Hu 2001]. It places “active” segments on the outer zones and inactive segments on the inner zones to optimize for faster writes. A segment’s *active ratio* is calculated using the average of the active ratio of each file in the segment. As in LFS, active is calculated by looking at recency of access, but PROFS also considers the size of the file and the last active ratio. The reorganization happens during LFS garbage collection. We will improve on the ideas behind PROFS by identifying hot and cold according to frequency at a block level and place it accordingly as well as reorganize during band compaction.

The closest LFS implementation to our work is the reordering write buffer of LFS, or WOLF [Wang and Hu 2002]. WOLF sorts the incoming write data blocks into *active* and *inactive* data buffers, again using recency of access as the metric for active. The sorting of data before it is written to disk is intended to reduce the overhead of the segment cleaner. WOLF follows Matthews’ proposed cleaning heuristic of combining the LFS cost-benefit policy with hole-plugging. In addition to using frequency of access as our metric, we continue to evaluate data during band compaction and place data in hot and cold bands appropriately.

HyLog proposes a new type of hybrid log design to address LFS’s poor cleaning performance at high disk utilizations [Wang et al. 2004]. HyLog writes hot pages using standard LFS techniques but writes cold pages in an “update-in-place” style that they call *overwrite*. Heat is measured on a per-disk-page basis, using frequency of write over an interval of time. Each disk page has a counter that is incremented every time it receives a write during the measurement interval. After this interval, the division of hot and cold pages occurs.

Segment cleaning in HyLog is adapted from Matthews’ cleaning technique with a notable change: in Matthews’ work, the cleaning choices are either cost-benefit or hole-plugging, and the ratios are calculated over every segment equally, but HyLog separates the ratios based on the heat of the segment. Specifically, HyLog compares the best ratio for hole-plugging over hot data, cost-benefit over hot data, hole-plugging for cold data, and cost-benefit for cold data. Although this is effective for LFS, hole-plugging is not viable for SMR disks, nor is updating in place.

There is often write contention found between the segment cleaner and the incoming data to be written. Gecko [Shin et al. 2013] solves this problem by chaining a small number of hard drives together into a single log. The tail of the log then is at a separate hard drive than the head of the log. By separating the head and tail of the log in this way, there is no longer write contention, as new data can be written to one drive while another is cleaning segments.

2.2. Flash

There is a rich body of work in the area of Flash storage that focuses on the separation of hot and cold data [Desnoyers 2014; Hsieh et al. 2006]. However, some of the restrictions that apply to Flash and solid state drives can be ignored for SMR disks and vice versa [Desnoyers 2013]. For instance, SMR disks do not care about wear leveling, nor is it necessary to zero a band before writing to it. In addition, bands can and will increasingly become much larger than a Flash page or erase block. When reading or writing data, there are seek penalties that must be considered for hard drives that do not exist for Flash. Therefore, any application of techniques borrowed from Flash would be in the spirit of their original work and would not be likely to work as a direct port.

2.3. Shingled Disk

Amer et al. originally introduced the potential for SMR disks and the changes that would be required for their adoption [Amer et al. 2010]. Pitchumani et al. emulated a shingled write disk on a traditional hard drive. While informative, it is not necessary to fully emulate a shingled write disk to measure the reduction in data movement [Pitchumani et al. 2012]. Skylight [Aghayev and Desnoyers 2015] was novel and drilled a hole into a SMR drive to understand how Seagate SMR drives work. This work will be instrumental in making decisions when testing our approach in the future.

Much of the research on SMR disks has focused on mitigating the “random update problem” [Hall et al. 2012; He and Du 2014; Cassuto et al. 2010; Lin et al. 2012]. This is because due to the nature of SMR disks, random updates cannot happen. This work is complementary to ours, as we do not treat random updates any differently. Since band compaction is an inevitability, our work focuses on the proper selection of bands for compaction.

There has been some work on developing file systems for SMR drives [Le Moal et al. 2012; Jin et al. 2014], which is also complementary to our work. If we have semantic information from the file system, we can make better decisions on which bands have data that are likely to stay cool and which are likely to heat up. HiSMRfs [Jin et al. 2014] is capable of handling raw SMR drives, which would make providing information to the algorithms to select the proper bands for compaction more transparent.

3. ALGORITHMS

To use shingled disks effectively, we look at algorithms for reducing data movement during band compaction. Our approach is to look at the nature of the data blocks, classifying them appropriately. Techniques like this have been used in the past to produce self-optimizing data storage systems [Bhadkamkar et al. 2009] and to reduce energy consumption of massive storage systems and devices [Essary and Amer 2008]. To that end, we have developed a mechanism for classifying the hotness and coldness of data blocks that can be employed as part of the band compaction process. We compare our cold-weight algorithm against the more traditional baseline ranking that does not attempt to classify the blocks.

3.1. Empty

Empty is our greedy naive algorithm, which will always try to find an empty band. If there are no completely empty bands, empty will pick the bands with the least amount of live data to compact together. When compacting, empty will only find the number of bands it needs. For example, in two-band compaction, empty will stop after finding the two bands with the least amount of live data; in four-band compaction, empty will stop after four.

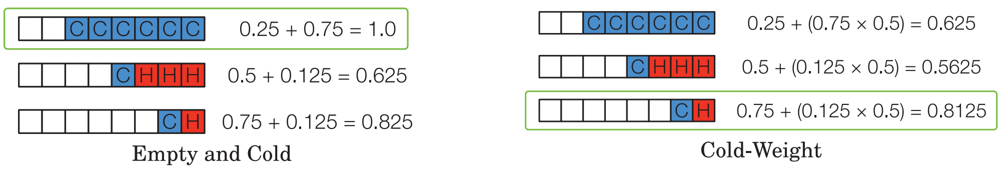


Fig. 1. Example of why we need the weights in cold-weight. In (a), we show the formula for cold-weight without the weight. Without the weight, it picks the segment that is 75% full of cold blocks. In (b), we show that cold-weight in the same example with a weight of 50% will pick the band that is mostly empty with one cold and one hot block.

3.2. Cold-Weight

The cold-weight algorithm combines a band's freeness, or the amount of free space, with the heat (or the lack thereof) of the blocks contained within the band. The formula our algorithm uses to select bands is shown in Equation (1). F is the fraction of the band that is free; it is calculated as the number of dead blocks in a band divided by the number of blocks in the band. C is the fraction of the band that contains cold data; it is calculated as the number of cold blocks in a band divided by the number of blocks in a band. H is the fraction of the band that contains hot data; it is calculated as the number of hot blocks divided by the number of blocks in a band. These three variables will add up to 1: $F + C + H = 1$. w_{cold} is the weight on cold, and w_{hot} is the weight on hot. All variables, fractions, and weights are expressed in decimal form.

$$F + (w_{\text{cold}} \times C) + (w_{\text{hot}} \times H), \tag{1}$$

where H, the hot fraction, can be measured in terms of free and cold.

$$H = 1 - F - C \tag{2}$$

Thus, our algorithm can be simplified to look only at the free space and the cold data. Equation (3) contains the final form of the formula used by our algorithm to select bands for compaction and shows how everything can be represented as a weighted value on cold. In the derivation of Equation (3), the addition of the term w_{hot} is dropped because it is a constant value and does not change on a per-experiment basis.

$$\begin{gathered} F + (w_{\text{cold}} \times C) + (w_{\text{hot}} \times (1 - F - C)) \\ F + (w_{\text{cold}} \times C) + w_{\text{hot}} - (w_{\text{hot}} \times F) - (w_{\text{hot}} \times C) \\ F \times (1 - w_{\text{hot}}) + C \times (w_{\text{cold}} - w_{\text{hot}}) \\ F + C \times \left(\frac{w_{\text{cold}} - w_{\text{hot}}}{1 - w_{\text{hot}}} \right) \end{gathered} \tag{3}$$

The weight on cold can increase or decrease the importance of compacting bands that contain mostly cold data versus mostly hot data. For the workloads examined, we have found that a smaller weight on cold data produces the best results. Using the formula in Equation (3), we calculate a value for each band and select the bands with the highest values to use for compaction.

Figure 1 shows why weighting is important. In Figure 1(a), we see that without weighting, we will pick the band with that is mostly full of cold blocks. This is because without weighting, we have assigned equal importance to cold blocks and free blocks. This is the point of putting the weight on cold blocks. In Figure 1(b), we use a weight of 50% on the percentage of the band that is cold. We pick the band that is mostly free with one cold and hot block. This is preferable because it is better to take the chance on the single hot block than to continuously move blocks from mostly full bands to new locations.

Table I. Important Statistics for the Project and Source Control 1 Servers

	Project	Source 1
Number of write requests	2,496,935	2,170,271
Total data written	26GB	31GB
Trace footprint	9.5GB	4.4GB
Percentage of hot LBAs	20.45	19.45
Percentage of cold LBAs	79.55	81.55
Percentage of trace that is hot	64.68	88.81
Percentage of trace that is cold	35.32	11.19

4. EXPERIMENTAL SETUP

In this section, we cover all of the components involved in running our experiments. We introduce the input datasets that were used to test the band compaction algorithms and discuss the methods used to stress the algorithms. We also discuss a technique to better classify incoming blocks. Finally, we outline our code flow and describe all possible paths an incoming write block can take, along with the events that can be triggered. All of the experiments were kept in memory, and "writing to disk" was simulated.

4.1. Datasets

We use two traces from the MSR Cambridge dataset that was introduced in FAST 2008 [Narayanan et al. 2008]. Specifically, we use the largest data volumes from the project (proj) and source code (src1) servers. The traces were gathered over a 1-week period, are block traces, and are stored on servers running NTFS. Table I shows some informative statistics about the traces we used. Both traces were chosen for their large number of write requests. We specifically chose the largest of the data volumes for the project and source servers from the MSR Cambridge traces because the data volumes will be more similar to a user workload than the traces of the system volumes.

4.2. Prepopulation

Since the benefits of this technique should accumulate over time, we are interested in behavior over ever-longer periods of time and ever-larger datasets. To that end, in addition to studying the basic MSR traces, we extended the workloads by prepopulating our system with a random ordering of the same trace we would be playing. We cut the trace into chunks of 10 timesteps, where a timestep is considered to be 1 second. This means that 10 timesteps is at least 10 seconds but could be longer if there is a period of inactivity in the trace. These chunks were randomly reordered and written out to a file. We chose to write it to a file because we can recreate the same on-disk state for each run for fair comparison. We tested both algorithms without prepopulation; prepopulating with a random selection of 25%, 50%, and 75% of the write requests in the trace; and prepopulating with a random ordering of all of the writes in each trace file.

4.3. Write Buffer

To model a more realistic workload, we implemented a write buffer, which also afforded the opportunity to better classify incoming blocks as hot or cold. The size of the write buffer varies proportionally to the size of the band; it is always double the band size. When the write buffer is filled, it will pick an empty band to write out some of the data. If there is more hot data than cold data in the buffer, it will write out the hottest of the data blocks. If there is more cold data than hot, the write buffer will write out cold data blocks. When there are no empty bands to write to, the write buffer will invoke band compaction. When band compaction completes, the write buffer will fill the current active band designated by the simulation with the data of the more prevalent temperature until it fills the currently active band.

4.4. Code Flow

When a write request comes in, the request is divided into blocks. For each block written, a check is issued to see if it is already "on disk." If it is, the heat of the block is incremented, and the current location on disk is marked as invalid. The block is then added to the write buffer. If it does not exist on disk, the block is marked as cold and immediately added to the write buffer. Currently, heat information is kept with the mapping of an LBA to the physical location in our simulation. In a real system, the heat information could be conveyed using a single bit in the shingle translation layer.

When the write buffer is full, it selects a band to fill from any available empty bands. If there are no empty bands, the write buffer will invoke band compaction. Band compaction returns the location of a band to write to, and the write buffer fills the band with either the hottest or coldest data in the buffer, as was described in Section 4.3. When the trace ends, band compaction is called as many times as necessary to flush the write buffer to disk. Currently, band compaction is only invoked in these two situations, and in both situations it is invoked by the write buffer.

When band compaction is invoked, the first step is to calculate values for each of the bands in the system using one of the algorithms described in Section 3.2. The bands with the highest values are selected based on however many bands are being compacted. As mentioned previously, the entire band has to be read, so all live data is read from the selected bands and sorted by their current write heat. At this point, a cooling step occurs: all of the heat counters are reset, making all of the data previously read cold. This is the only point at which data cooling currently happens. All of the bands that were read from are now marked as empty and are free to use. The live blocks that were read are written back to a free band. If we fill up the band before running out of live blocks, another recently freed band is selected and writing continues. When all live blocks have been written back to disk, the band that is currently being written to is the one that will receive future writes. In the case where the last block we wrote to is the last location in a band, another band is selected from the set of recently freed bands.

5. RESULTS

We tested weights on cold in our cold-weight algorithm in 10% increments. These weights yielded both positive and negative results. We have chosen to present the graphs for the optimal weights for both the empty algorithm and the cold-weight algorithm for the sake of brevity. However, we have also provided tables for all of the weights across all experiments. Tables II, III, IV, V, and VI show the blocks moved and the difference between cold-weight and empty in percentage form for each of the runs using a band size of 40MB for 0%, 25%, 50%, 75%, and 100% prepopulation, respectively. The results for a band size of 80MB are similar and have been omitted due to space constraints. For a band size of 40MB, the project data set experiments use 279 total bands, and the source dataset experiments use 128 bands.

In any band, there can be hot, cold, and free blocks. In addition to moving fewer blocks during band compaction, we also want to minimize the number of bands that have a mix of all three types of blocks, as this will give us a better separation of hot and cold blocks. The graphs show the number of blocks in each band that are hot (in red), cold (in blue), and free (in green); the bands are then sorted by heat.

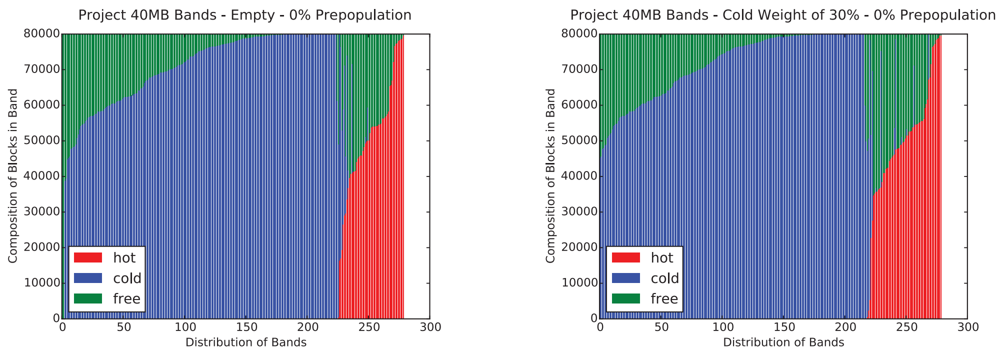


Fig. 2. Comparison of the empty and cold-weight algorithms for project without prepopulation using 40MB bands. Green represents free blocks, blue represents cold blocks, and red represents hot blocks.

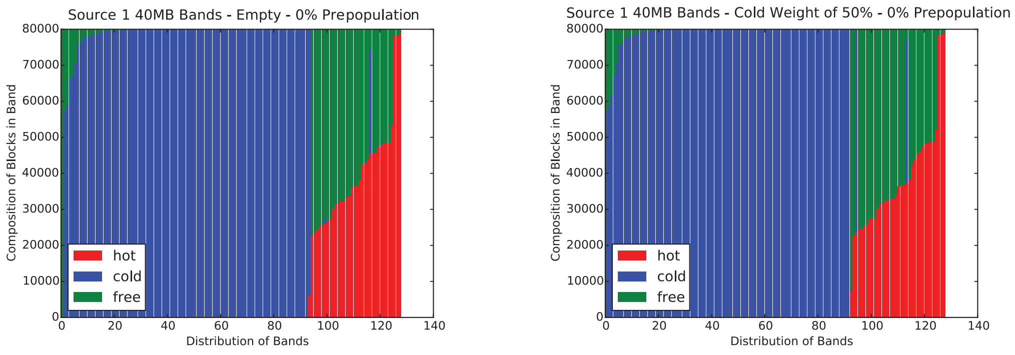


Fig. 3. Comparison of the empty and cold-weight algorithms for Source 1 without prepopulation using 40MB bands. Green represents free blocks, blue represents cold blocks, and red represents hot blocks.

5.1. No Prepopulation

Figures 2 and 3 show the distribution of hot and cold blocks over all bands in the experiments that were run without prepopulation. Due to the write buffer discussed in Section 4.3, we achieve a very good separation of hot and cold data.

5.1.1. Project. Figure 2 shows the project dataset experiments. In both the empty and cold-weight experiments, the write buffer kept the number of bands with a mix of hot, cold, and free blocks under 20. As seen in Table II, using cold-weight with a 30% weight produced the best reduction in data movement, a 12% improvement over empty.

5.1.2. Source 1. Figure 3 shows the source dataset experiments. In these experiments, the number of bands with a mix of blocks was kept under 10 by the write buffer. It is interesting to note that for this workload, the optimal weight for the cold-weight algorithm was 50%, producing close to 3% improvement.

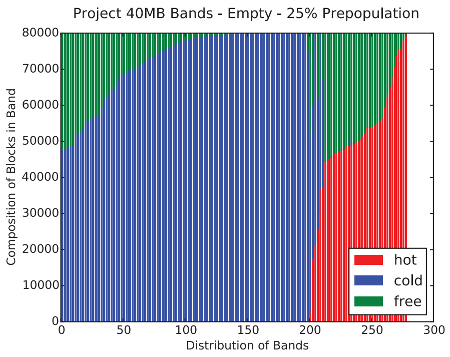
5.2. 25% Prepopulation

We use a random selection of 25% of the write requests from the original traces to prepopulate the system, as described in Section 4.2. Because we are using the same trace data to prepopulate the system as we are to run the experiments and cooling only happens during band compaction, having more hot blocks at the end of the trace means that fewer data blocks were moved.

Table II. Full Results Table for 0% Prepopulation

	Project 40MB		Source 1 40MB	
Experiment	Blocks Moved	Percentage Improvement	Blocks Moved	Percentage Improvement
Empty	2,378,357	—	318,607	—
Cold-weight 10%	2,193,595	+7.77	319,525	–0.29
Cold-weight 20%	2,206,924	+7.21	313,921	+1.47
Cold-weight 30%	**2,082,264**	**+12.45**	342,592	–7.53
Cold-weight 40%	2,142,427	+9.92	336,356	–5.57
Cold-weight 50%	2,345,437	+1.38	**310,262**	**+2.62**
Cold-weight 60%	3,252,682	–36.76	310,987	+2.39
Cold-weight 70%	3,740,842	–57.29	408,510	–28.22
Cold-weight 80%	6,182,745	–159.96	6,154,620	–1,831.73
Cold-weight 90%	23,213,149	–876.02	8,924,629	–2,701.14

Note: Optimal values for each dataset are in bold.



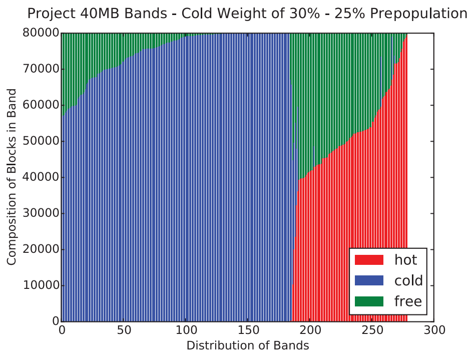


Fig. 4. Comparison of the empty and cold-weight algorithms for project with 25% prepopulation using 40MB bands. Green represents free blocks, blue represents cold blocks, and red represents hot blocks.

Table III. Full Results Table for 25% Prepopulation

	Project 40MB		Source 1 40MB	
Experiment	Blocks Moved	Percentage Improvement	Blocks Moved	Percentage Improvement
Empty	8,866,029	—	703,060	—
Cold-weight 10%	9,032,118	–1.87	658,354	+6.36
Cold-weight 20%	9,130,049	–2.98	672,584	+4.33
Cold-weight 30%	**8,510,548**	**+4.01**	682,636	+2.91
Cold-weight 40%	9,630,121	–8.62	**612,659**	**+12.86**
Cold-weight 50%	11,238,928	–26.76	703,112	–0.01
Cold-weight 60%	27,519,815	–210.40	816,907	–16.19
Cold-weight 70%	91,063,541	–927.11	906,926	–29.00
Cold-weight 80%	238,093,656	–2,585.46	8,900,399	–1,165.95
Cold-weight 90%	345,304,754	–3,794.69	9,181,743	–1,205.97

Note: Optimal values for each dataset are in bold.

5.2.1. Project. Figure 4 shows the distribution of hot and cold blocks for all bands in the project experiments. As can been seen by comparing the empty and cold-weight graphs, there is more hot data left by the cold-weight algorithm than by the empty algorithm. This is illustrated in Table III, where a 30% weight on cold shows a 4% improvement in reducing data movement. Additionally, for this workload, cold-weight

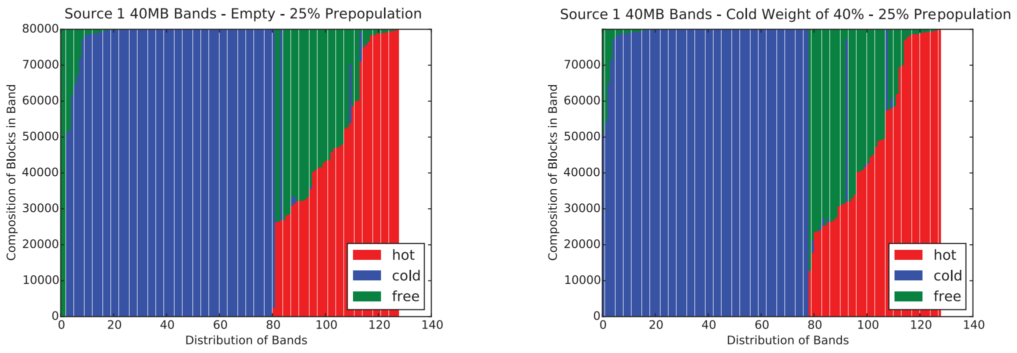


Fig. 5. Comparison of the empty and cold-weight algorithms for Source 1 with 25% prepopulation using 40MB bands. Green represents free blocks, blue represents cold blocks, and red represents hot blocks.

Table IV. Full Results Table for 50% Prepopulation

	Project 40MB		Source 1 40MB	
Experiment	Blocks Moved	Percentage Improvement	Blocks Moved	Percentage Improvement
Empty	22,411,027	—	1,608,014	—
Cold-weight 10%	20,985,569	+6.36	1,645,114	–2.31
Cold-weight 20%	**20,486,456**	**+8.59**	1,728,419	–7.49
Cold-weight 30%	22,209,083	+0.9	1,646,636	–2.40
Cold-weight 40%	26,117,640	–16.54	**1,548,803**	**+3.68**
Cold-weight 50%	52,224,438	–133.03	1,639,150	–1.94
Cold-weight 60%	207,269,171	–824.85	1,565,683	+2.63
Cold-weight 70%	305,348,799	–1,262.49	15,514,185	–32.05
Cold-weight 80%	323,949,950	–1,345.49	6,638,597	–312.84
Cold-weight 90%	374,580,433	–1,571.41	10,604,746	–559.49

Note: Optimal values for each dataset are in bold.

does a better job of separating hot and cold data, leaving only 11 bands that have a mix of hot, cold, and free compared to empty's 17 bands.

5.2.2. Source 1. Figure 5 shows the results of the source experiments. Comparing the graphs in Figure 5 shows that there are more hot bands in the cold-weight graph. This means the cooling that occurs during band compaction is happening less often and is reflected in the results shown in Table IV, where a 40% weight on cold-weight yields an almost 13% improvement.

5.3. 50% Prepopulation

We use a random selection of 50% of the write requests to prepopulate the system, as described in Section 4.2. The exact ordering in the 25% prepopulation input is not contained in the 50% prepopulation input. Since we are using the same trace data to prepopulate the system as we are to run the experiments and cooling only happens during band compaction, more hot blocks at the end of the trace means that less data was moved.

5.3.1. Project. Figure 6 shows the distribution of hot and cold blocks for all bands in the project experiments. As can been seen in the graphs, there is more hot data left by the cold-weight algorithm than by the empty algorithm. This is illustrated in Table IV, where a 20% weight on cold shows a 8.59% improvement in reducing data movement. Although there are more bands with a mix of hot, cold, and free blocks (19 bands in

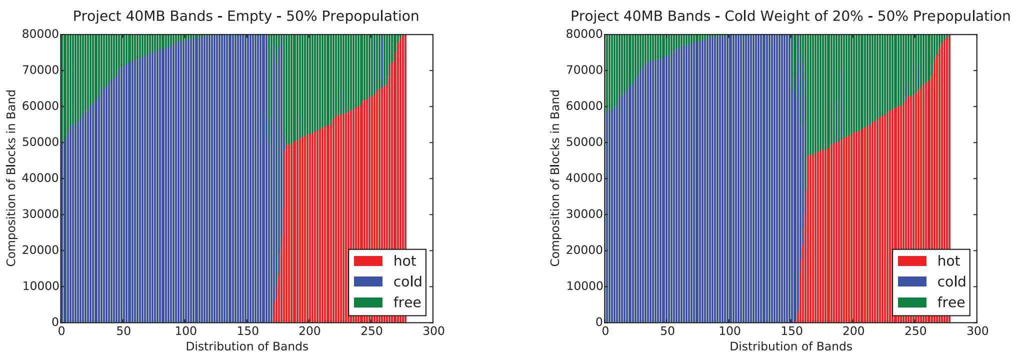


Fig. 6. Comparison of the empty and cold-weight algorithms for project with 50% prepopulation using 40MB bands. Green represents free blocks, blue represents cold blocks, and red represents hot blocks.

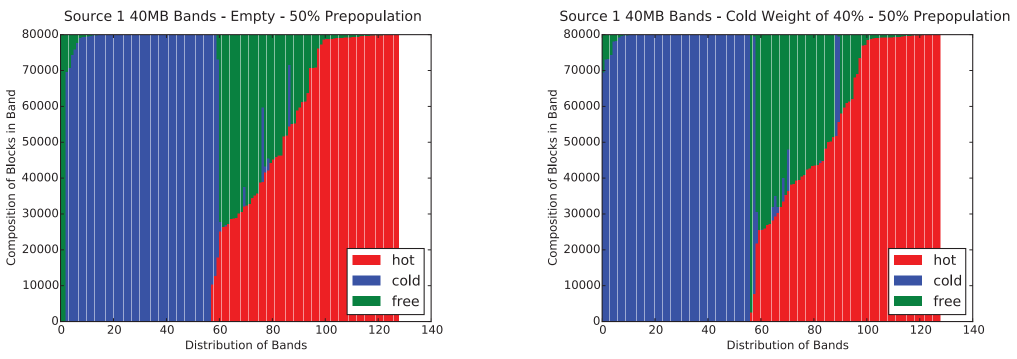


Fig. 7. Comparison of the empty and cold-weight algorithms for Source 1 with 50% prepopulation using 40MB bands. Green represents free blocks, blue represents cold blocks, and red represents hot blocks.

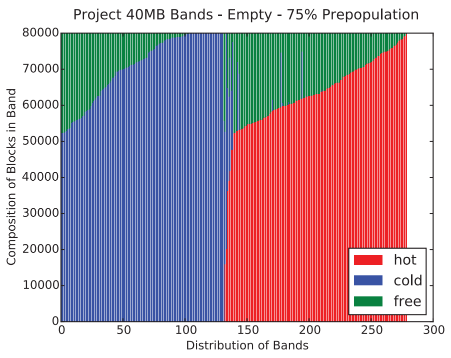
cold-weight vs. 18 in empty), it is most likely due to the behavior of the write buffer. Toward the end of the trace, the write buffer wrote hot data to a recently compacted band, resulting in mixed data. We will address this behavior in future work.

5.3.2. Source 1. Figure 7 shows the results of the source experiments. Interestingly, both cold-weight and empty have 11 bands remaining with a mix of hot, cold, and free blocks. In the cold-weight experiment, there is about one band worth of extra hot blocks compared to empty. This means that most of the segments with cold data were mostly full, so we were selecting mostly empty segments with hot data and cooling them. Table IV shows that with a 40% weight, cold-weight shows an almost 4% improvement.

5.4. 75% Prepopulation

We use a random selection of 75% of the write requests from the original traces to prepopulate the system, as described in Section 4.2. The exact ordering in the 25% prepopulation and 50% prepopulation inputs are not contained in the 75% prepopulation input. Because we are using the same trace data to prepopulate the system as we are to run the experiments and cooling only happens during band compaction, having more hot blocks at the end of the trace means that fewer data blocks were moved.

5.4.1. Project. Figure 8 shows the distribution of hot and cold blocks for all bands in the project experiments. As can been seen by comparing the empty and cold-weight graphs, there is more hot data left by the cold-weight algorithm than by the empty



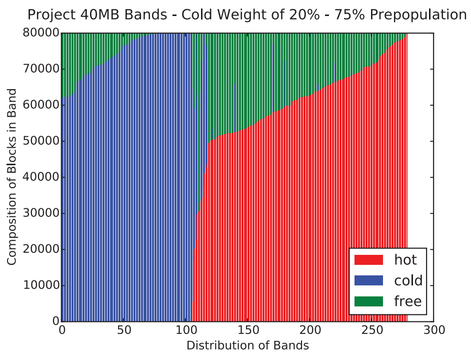
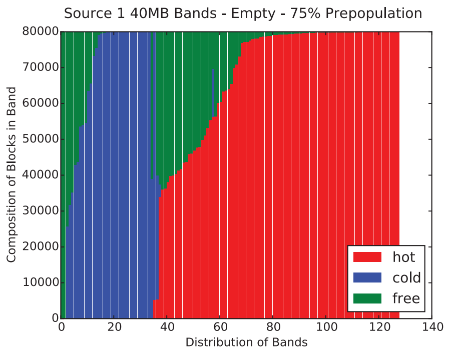


Fig. 8. Comparison of the empty and cold-weight algorithms for project with 75% prepopulation using 40MB bands. Green represents free blocks, blue represents cold blocks, and red represents hot blocks.

Table V. Full Results Table for 75% Prepopulation

	Project 40MB		Source 1 40MB	
Experiment	Blocks Moved	Percentage Improvement	Blocks Moved	Percentage Improvement
Empty	40,028,737	—	11,947,766	—
Cold-weight 10%	43,936,049	–9.76	10,085,658	+15.59
Cold-weight 20%	**39,977,633**	**+0.13**	9,869,204	+17.40
Cold-weight 30%	44,257,228	–10.56	9,649,934	+19.23
Cold-weight 40%	56,245,951	–40.51	8,963,526	+24.98
Cold-weight 50%	180,537,867	–351.02	**8,494,101**	**+28.91**
Cold-weight 60%	291,878,238	–629.17	8,760,094	+26.68
Cold-weight 70%	349,209,281	–772.4	15,514,185	–29.85
Cold-weight 80%	297,438,218	–643.06	29,341,272	–145.58
Cold-weight 90%	439,488,337	–997.93	43,572,131	–264.69

Note: Optimal values for each dataset are in bold.



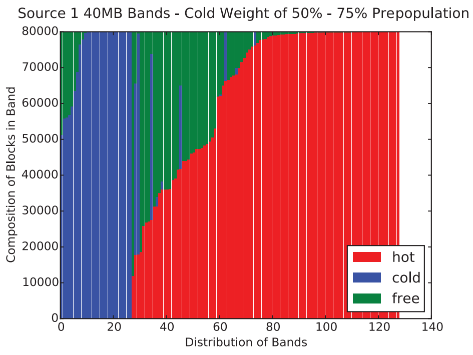
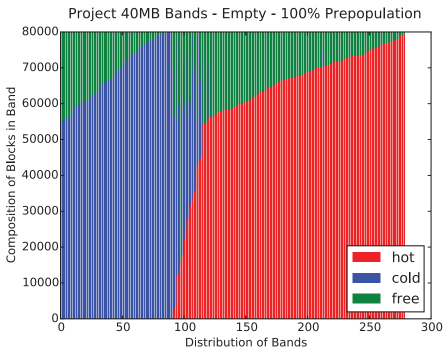


Fig. 9. Comparison of the empty and cold-weight algorithms for Source 1 with 75% prepopulation using 40MB bands. Green represents free blocks, blue represents cold blocks, and red represents hot blocks.

algorithm. This is illustrated in Table V, where a 20% weight on cold shows a 0.13% improvement in reducing data movement.

5.4.2. Source 1. Figure 9 shows the comparison of the block distributions between cold-weight and empty for the source 1 experiments. Table V shows that with a 50% weight, cold-weight shows an almost 29% improvement. Despite this massive improvement, cold-weight contains 10 bands with a mixture of hot, cold, and free blocks, whereas



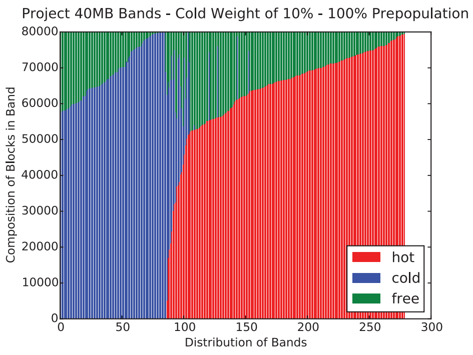


Fig. 10. Comparison of the empty and cold-weight algorithms for project with 100% prepopulation using 40MB bands. Green represents free blocks, blue represents cold blocks, and red represents hot blocks.

Table VI. Full Results Table for 100% Prepopulation

	Project 40MB		Source 1 40MB	
Experiment	Blocks Moved	Percentage Improvement	Blocks Moved	Percentage Improvement
Empty	72057,345	—	43,113,372	—
Cold-weight 10%	**66,031,282**	**+8.36**	37,270,929	+13.55
Cold-weight 20%	68,707,085	+4.65	35,571,691	+17.49
Cold-weight 30%	70,874,379	+1.64	29,727,847	+31.05
Cold-weight 40%	84,283,621	–16.97	28,067,541	+34.90
Cold-weight 50%	374,162,922	–419.26	**23,830,466**	**+44.73**
Cold-weight 60%	360,794,866	–400.71	29,280,929	+32.08
Cold-weight 70%	709,571,024	–884.73	43,549,561	–1.01
Cold-weight 80%	729,328,043	–912.15	147,499,502	–242.12
Cold-weight 90%	776,225,608	–977.23	141,843,320	–229.00

Note: Optimal values for each dataset are in bold.

empty only contains 5 bands with a mixture of hot, cold, and free blocks. This issue will be addressed in future work.

5.5. 100% Prepopulation

As in the experiments with 25%, 50%, and 75% prepopulation, more hot data at the end of the trace (as shown in the graphs) means that we are touching less data over the course of the experiment. We are prepopulating the system with 100% of the write requests from the same trace data, in random order.

5.5.1. Project. The results of the project experiments are very similar to the results with 50% prepopulation. As can be seen in Figure 10, there is more hot data remaining with the cold-weight algorithm, reflected in the 8.36% improvement shown in Table VI when using the optimal weight of 10%. Cold-weight also does a better job of separating hot and cold data, ending with 19 bands of mixed blocks compared to empty's 30 bands.

5.5.2. Source 1. The source experiments with 100% prepopulation showed the greatest improvement in reducing data movement, at 50% weight for our cold-weight algorithm. Figure 11 shows the difference in the amount of hot data left with cold-weight versus empty. Cold-weight with a weight of 50% results in an almost 45% improvement in reducing data movement. Although there are more bands with a mix of hot, cold, and free blocks (14 bands in cold-weight vs. 10 in empty), it is most likely due to the behavior of the write buffer. Toward the end of the trace, the write buffer wrote hot data to a

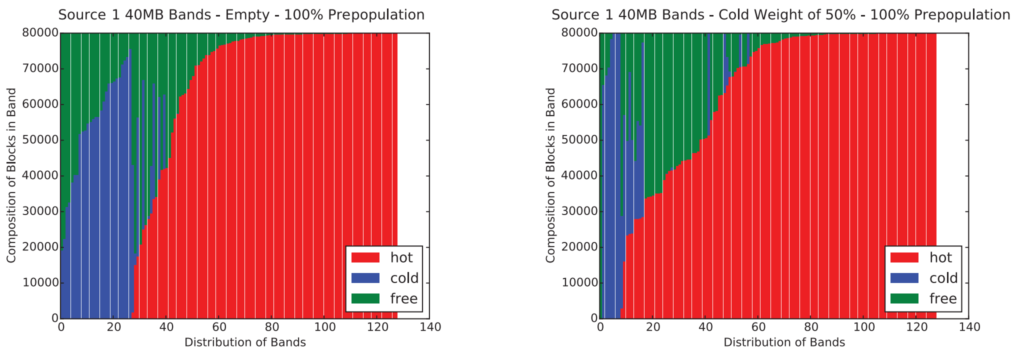


Fig. 11. Comparison of the empty and cold-weight algorithms for Source 1 with 100% prepopulation using 40MB bands. Green represents free blocks, blue represents cold blocks, and red represents hot blocks.

recently compacted band, resulting in mixed data. We will address this behavior in future work.

6. FUTURE WORK

The work presented here is only the beginning of our work to develop techniques to reduce data movement in storage systems and evaluate their usefulness to SMR disks and their applications. Additional data points will help inform more future work and will also further demonstrate the success of our work. In addition, we will implement LFS's cost-benefit policy as a band compaction algorithm and compare it to our cold-weight algorithm.

We hypothesize that the bands containing a mix of hot and cold blocks can be further reduced by modifying the band compaction algorithm, as well as improving how the write buffer flushes data. The write buffer can keep track of distinct hot and cold bands, and maintain this distinction when writing data out. The band compaction algorithm then can be modified to take advantage of these distinct hot and cold bands during band compaction. All compacted blocks will be written to the cold band, and a third band will be introduced to represent blocks that "were once hot," or were hot in the write buffer.

We will investigate other metrics beyond write frequency to reduce data movement in SMR disks. The last element in this stage of our work is to develop a dynamic band compaction algorithm that automatically tunes the values of the weights. Since workloads can, and often do, change behavior over their lifetime, it is necessary to be able to adjust the band compaction algorithm as needed to obtain the best performance.

In this article, we introduced our work and discussed the benefits that it will have for shingled drives. Skylight [Aghayev and Desnoyers 2015] has demonstrated that Seagate's shingled drives do not use LFS-style compaction to reclaim space, but it is possible that other drive manufacturers will use LFS-style compaction. We will also expand on how to adapt this work for host-managed and host-aware drives. The techniques presented in this article are not limited by the storage media. They can be employed to reduce data movement on any storage media that employs an LFS approach to storing data.

7. CONCLUSIONS

Since band compaction is a necessity for SMR disks, it is clear that improved algorithms are needed. In this article, we proposed using write frequency as a metric to separate blocks to reduce data movement. We presented a band compaction algorithm we developed that implements this heuristic and showed that our algorithm results in up to a 45% reduction in required data movements when compared to naive approaches.

Our cold-weight algorithm favors the selection of the emptiest bands but also weights how much of the band consists of cold data when selecting bands to compact. We found that the optimal value for the weight on cold varies depending on the workload. However, there is always a weight that produces an improvement in reducing data movement. We also have shown that using a write buffer provides excellent separation of hot and cold data.

We conclude that write frequency is a valid and useful metric when used in a band compaction algorithm such as our cold-weight algorithm. This kind of algorithm, combined with a write buffer, reduces the amount of data moved over the lifetime of a workload.

ACKNOWLEDGMENTS

The authors would like to thank the students of the Storage Systems Research Center for their suggested edits to this article.

REFERENCES

Abutalib Aghayev and Peter Desnoyers. 2015. Skylight-A window on shingled disk operation. In *Proceedings of the 13th USENIX Conference on File and Storage Technologies*. 135–149.

Ahmed Amer, Darrell D. E. Long, Ethan L. Miller, J.-F. Paris, and S. J. Thomas Schwarz. 2010. Design issues for a shingled write disk system. In *Proceedings of the 2010 IEEE 26th Symposium on Mass Storage Systems and Technologies (MSST'10)*. IEEE, Los Alamitos, CA, 1–12.

Medha Bhadkamkar, Jorge Guerra, Luis Useche, Sam Burnett, Jason Liptak, Raju Rangaswami, and Vagelis Hristidis. 2009. BORG: Block-reORGanization for self-optimizing storage systems. In *Proceedings of the 7th USENIX Conference on File and Storage Technologies (FAST'09)*.

Tevor Blackwell, Jeffrey Harris, and Margo Seltzer. 1995. Heuristic cleaning algorithms in log-structured file systems. In *Proceedings of the Winter 1995 USENIX Technical Conference*.

Yuval Cassuto, Marco A. A. Sanvido, Cyril Guyot, David R. Hall, and Zvonimir Z. Bandic. 2010. Indirection systems for shingled-recording disk drives. In *Proceedings of the 2010 IEEE 26th Symposium on Mass Storage Systems and Technologies (MSST'10)*. IEEE, Los Alamitos, CA, 1–14.

Peter Desnoyers. 2013. What systems researchers need to know about NAND flash. In *Proceedings of the 5th USENIX Workshop on Hot Topics in File and Storage Technologies (HotStorage'13)*.

Peter Desnoyers. 2014. Analytic models of SSD write performance. *ACM Transactions on Storage* 10, 2, 8.

David Essary and Ahmed Amer. 2008. Predictive data grouping: Defining the bounds of energy and latency reduction through predictive data grouping and replication. *ACM Transactions on Storage* 4, 1, 2.

David Hall, John H. Marcos, and Jonathan D. Coker. 2012. Data handling algorithms for autonomous shingled magnetic recording HDDs. *IEEE Transactions on Magnetics* 48, 5, 1777–1781.

Weiping He and David H. C. Du. 2014. Novel address mappings for shingled write disks. In *Proceedings of the 6th USENIX Conference on Hot Topics in Storage and File Systems*. 5.

Jen-Wei Hsieh, Tei-Wei Kuo, and Li-Pin Chang. 2006. Efficient identification of hot data for flash memory storage systems. *ACM Transactions on Storage* 2. 1, 22–240.

Chao Jin, Wei-Ya Xi, Zhi-Yong Ching, Feng Huo, and Chun-Teck Lim. 2014. HiSMRfs: A high performance file system for shingled storage array. In *Proceedings of the 2014 30th Symposium on Mass Storage Systems and Technologies (MSST'14)*. IEEE, Los Alamitos, CA, 1–6.

Damien Le Moal, Zvonimir Bandic, and Cyril Guyot. 2012. Shingled file system host-side management of shingled magnetic recording disks. In *Proceedings of the 2012 IEEE International Conference on Consumer Electronics (ICCE'12)*. IEEE, Los Alamitos, CA, 425–426.

Chung-I. Lin, Dongchul Park, Weiping He, and David Du. 2012. H-SWD: Incorporating hot data identification into shingled write disks. In *Proceedings of the 20th IEEE International Symposium on Modeling, Analysis, and Simulations of Computer and Telecommunication Systems (MASCOTS'12)*. 248–255.

Jeanna Neefe Matthews, Drew Roselli, Adam M. Costello, Randolph Y. Wang, and Thomas E. Anderson. 1997. Improving the performance of log-structured file systems with adaptive methods. In *Proceedings of the 16th ACM Symposium on Operating Systems Principles (SOSP'97)*. 238–251.

Dushyanth Narayanan, Austin Donnelly, and Antony Rowstron. 2008. Write off-loading: Practical power management for enterprise storage. In *Proceedings of the 6th USENIX Conference on File and Storage Technologies*. 17.

Rekha Pitchumani, Andy Hospodor, Ahmed Amer, Yangwook Kang, Ethan L. Miller, and Darrell D. E. Long. 2012. Emulating a shingled write disk. In *Proceedings of the 20th IEEE International Symposium on Modeling, Analysis, and Simulation of Computer and Telecommunication Systems (MASCOTS'12)*.

Mendel Rosenblum and John K. Ousterhout. 1992. The design and implementation of a log-structured file system. *ACM Transactions on Computer Systems* 10, 1, 26–52.

Margo Seltzer, Keith Bostic, M. Kirk McKusick, and Carl Staelin. 1993. An implementation of a log-structured file system for UNIX. In *Proceedings of the Winter 1993 USENIX Technical Conference*. 307–326.

Ji-Yong Shin, Mahesh Balakrishnan, Tudor Marian, and Hakim Weatherspoon. 2013. Gecko: Contention-oblivious disk arrays for cloud storage. In *Proceedings of the 11th USENIX Conference on File and Storage Technologies*. 285–298.

Jun Wang and Yiming Hu. 2001. PROFS-performance oriented data reorganization for log-structured file system on multi-zone disks. In *Proceedings of the 9th International Symposium on Modeling, Analysis, and Simulation of Computer and Telecommunication Systems (MASCOTS'01)*.

Jun Wang and Yiming Hu. 2002. WOLF—a novel reordering write buffer to boost the performance of log-structured file systems. In *Proceedings of the 1st Conference on File and Storage Technologies (FAST'02)*. 47–60.

Wenguang Wang, Yanping Zhao, and Rick Bunt. 2004. HyLog: A high performance approach to managing disk layout. In *Proceedings of the 3rd USENIX Conference on File and Storage Systems (FAST'04)*. 145–158.

Received November 2014; accepted November 2015