Track-based Translation Layers for Interlaced Magnetic Recording

Mohammad Hossein Hajkazemi, *Northeastern University;*
Ajay Narayan Kulkarni, *Seagate Technology;* Peter Desnoyers, *Northeastern University;*
Timothy R Feldman, *Seagate Technology*

https://www.usenix.org/conference/atc19/presentation/hajkazemi

This paper is included in the Proceedings of the
2019 USENIX Annual Technical Conference.

July 10–12, 2019 • Renton, WA, USA

ISBN 978-1-939133-03-8

Track-based Translation Layers for Interlaced Magnetic Recording

Mohammad Hossein Hajkazemi[⋆], Ajay Narayan Kulkarni[†], Peter Desnoyers[⋆], Timothy R Feldman[†]
Northeastern University[⋆], *Seagate Technology*[†]

Abstract

Interlaced magnetic recording (IMR) is a state-of-the-art recording technology for hard drives that makes use of heat-assisted magnetic recording (HAMR) and track overlap to offer higher capacity than conventional and shingled magnetic recording (CMR and SMR). It carries a set of write constraints that differ from those in SMR: "bottom" (e.g. even-numbered) tracks cannot be written without data loss on the adjoining "top" (e.g. odd-numbered) ones. Previously described algorithms for writing arbitrary (i.e. bottom) sectors on IMR are in some cases poorly characterized, and are either slow or require more memory than is available within the constrained disk controller environment.

We provide the first accurate performance analysis of the simple *read-modify-write (RMW)* approach to IMR bottom track writes, noting several inaccuracies in earlier descriptions of its performance, and evaluate it for latency, throughput and I/O amplification on real-world traces. In addition we propose three novel memory-efficient, track-based translation layers for IMR—*track flipping*, *selective track caching* and *dynamic track mapping*, which reduce bottom track writes by moving hot data to top tracks and cold data to bottom ones in different ways. We again provide a detailed performance analysis using simulations based on real-world traces.

We find that RMW performance is poor on most traces and worse on others. The proposed approaches perform much better, especially dynamic track mapping, with low write amplification and latency comparable to CMR for many traces.

1 Introduction

Magnetic recording technology has made enormous strides over the last several decades, reaching densities of about a terabit per square inch, higher than that of any but the most modern and densest solid-state storage technologies. Yet in recent years density improvements have run up against the *superparamagnetic limit* [17]—as bits get smaller, the magnetic media coercivity (resistance to being magnetized) must go up, to avoid bit flips from thermal noise, while as heads get smaller their magnetic field becomes weaker, requiring lower coercivity media. In other words, smaller bits require smaller track sizes, requiring smaller write heads, requiring lower-coercivity media, resulting in larger minimum bit sizes. When the minimum bit size becomes as large as the write head, further density improvements require new approaches. We are currently at or near this limit; increases in disk capacity in the past 5 years or more have relied more on increasing the number of platters per drive rather than increases in areal density.

New strategies allow further increases in areal density by sidestepping one or both sides of this trade-off, i.e. either breaking the link between bit size and write head size / magnetic field strength, or between bit reliability and media coercivity. Shingled Magnetic Recording (SMR) [1] overlaps adjacent tracks, reducing the effective track width without reducing the write head size. Yet, this increase in density comes at a cost: random writes are not allowed, as overwriting a sector will also overwrite the corresponding sector in the adjacent "downstream" track, and therefore the data could be lost. Heat-Assisted Magnetic Recording [12] takes advantage of the fact that the coercivity of a material goes down with temperature, and uses a laser to heat the media to near the Curie point[1] before writing. This allows use of a medium with much higher room-temperature coercivity and smaller grain size (and thus minimum bit size), and also allows an effective track width narrower than the write head by narrowing the width of the heated domain, by controlling the laser current.

Interlaced Magnetic Recording (IMR) [9] uses both heat-assisted recording and track overlap. This is in contrast to SMR, which uses track overlap but conventional room-temperature recording. As shown in Figure 1, tracks are written in an "interlaced" fashion, with a "bottom" layer of tracks written first, after which a "top" layer is written between (and partly overlapping) these bottom tracks. To avoid total overwrite of the bottom tracks, top tracks are written with a narrower width and thus slightly lower capacity, roughly 90% that of the

[1] The temperature above which the material will no longer retain its magnetic properties.

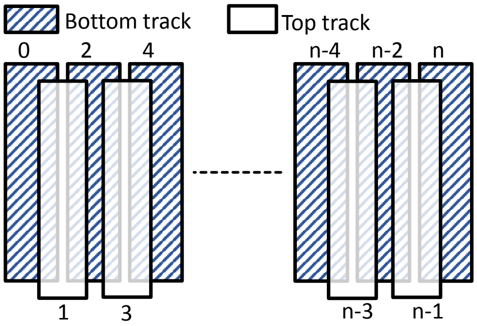


Figure 1: IMR technology: tracks are written in an interlaced fashion; top tracks are written between and over bottom tracks. Top track *i* partially overlaps bottom track *i-1* and *i+1*.

bottom tracks. The result is a drive with write constraints, but ones that are far less strict (and thus less performance-limiting) than those for SMR. Where SMR writes must be limited to a single track per zone (a group of a few hundred of tracks) to avoid data loss, in IMR nearly half of the sectors (i.e. those on the top tracks) can be re-written safely. Moreover whereas moving valid data in SMR requires reading and/or writing an entire (typically 256 MB) zone, for IMR in the worst case only two tracks (less than 5 MB) must be moved.

IMR is a very new technology, with the first descriptions of its physical feasibility dating to 2016 [4,9]; with one exception, Wu et al. [20] that is discussed in Section 5 the publications to date have focused on the magnetic and physical aspects of IMR, rather than system implications and algorithms. We provide the first thorough performance analysis of the naive *read-modify-write (RMW)* strategy described in the first IMR proposals [4], correcting several mistaken assumptions, and quantifying the performance degradation of IMR with RMW for real workloads. We offer three algorithms for improved management of IMR writes, *track flipping*, *selective track caching* and *dynamic track mapping*, all of which (unlike the approach of Wu et al.) may be readily implemented in disk firmware with limited memory and compute resources. We provide detailed performance models of these algorithms, and evaluate them in simulation on real workloads, showing substantial improvement over RMW in all cases and near-conventional-drive performance for some workloads.

In particular, the contributions of this paper are:

1. a thorough performance analysis of naive read-modify-write for IMR disk—i.e. as proposed in Hwang et al. [9]—showing a performance overhead of more than 2x that assumed by prior work,

2. three novel track-based translation layers—track flipping, selective track caching and dynamic track mapping—which mitigate most of the IMR performance penalty at a sufficiently modest cost in memory that they may be implemented within today's on-board disk controllers,

3. evaluation of conventional ("CMR") disk, RMW and the three proposed algorithms on real-world traces, demonstrating (a) significant costs to RMW vs. CMR, and (b) substantial improvements for all proposed algorithms particularly dynamic track mapping.

2 Algorithms for IMR

As with SMR, IMR write limitations maybe addressed from the host or within the device; however the complexity of the IMR track-to-track write restrictions makes it preferable to employ a device-based block translation layer rather than expose restrictions to the host.

We describe in detail four algorithms: naive read-modify-write [9], track flipping, selective track caching and dynamic track mapping. For each algorithm we estimate memory usage, describe the data copies and logging of mapping changes needed to prevent data loss in the case of a crash, and analyze the performance of the algorithm's operations.

2.1 Read-modify-write

The simplest IMR translation layer is what we term naive read-modify-write (RMW). Sectors on disk are assigned fixed logical addresses, as in conventional drives, and before performing any write to some sector S on a bottom track T, the drive (1) reads the adjacent top track sectors (S_{T-1} on $T-1$ and S_{T+1} on $T+1$) and (2) copies them to a "backup region", then (3) performs the bottom-track write, and finally (4) re-writes the adjacent top track sectors.

When numbering tracks $T-1$, T, $T+1$ we are referring to physical position, which may not directly correspond to logical block numbering. In particular IMR is expected to use a *serpentine* or *zig-zag* layout [10], where LBAs are numbered sequentially across N adjacent bottom tracks, and then across the corresponding N top tracks. The result is that sectors in physically adjacent top and bottom tracks will be separated by a distance of N track sizes either in all cases (zig-zag) or on average (serpentine).

Memory usage: Other than buffers for copying, no additional memory is required beyond that needed for standard LBA to physical location translation in a conventional drive.

Safety and crash consistency: We first note that to avoid data loss, host writes to the affected top tracks must be blocked during the RMW operation, as they would be overwritten when data is copied back to the tracks. The duration of this locking determines a phase, which is atomic with respect to user I/O, and mapping updates need only be persisted once per phase.

Copying sectors S_{T-1} and S_{T+1} to the backup region forms a single phase, and the temporary location of the sectors is logged to the backup region just before the phase completes. If a crash occurs before restoring the top tracks, a startup scan of the log will locate the saved data, which may be copied back to its proper location. The length of the log is determined by the number of simultaneous RMW operations allowed; if this is 1, then no log trimming is needed as it will just be replaced by the log from the next operation.

When logging data to the backup region, we can write additional metadata with negligible overhead, much like journal entries in a file system. Efficiently persisting the fact that S_{T-1} and S_{T+1} have been restored is more difficult, however; if this is not done, then future writes to these locations may be lost if stale backup data is copied back on restart. A straightforward way to do this is to clear the backup region; however this requires an additional seek and possibly lost rotation. Instead we clear the backup region *lazily*, if we detect a write to S_{T-1} or S_{T+1}. Since *any* RMW operation will clear the previous contents of the backup region, in most cases this lazy cleaning may be omitted, as until S_{T-1} or S_{T+1} are modified, the backup data is not stale and may be copied back safely on startup.

Timing: The performance of this approach may be analyzed by examining the steps above. We assume a random single-sector write to sector S on bottom track T, and assume as well that sectors S_{T-1} and S_{T+1} on tracks $T-1$ and $T+1$ respectively must be moved to avoid data loss. The time taken is thus at least:

1. $0.5t_{rot} + t_{seek}$ to reach and read sector S_{T-1}, where t_{rot} is the rotation time, assuming an average 0.5-rotation delay for random access.
2. A missed rotation, t_{rot}, to reach and read sector S_{T+1}.
3. $t_{seek} + 0.5t_{rot}$ to reach the backup region, plus negligible transfer time to write sector S_{T-1} and S_{T+1}.
4. $t_{seek} + 0.5t_{rot}$ (see below) to reach and write sector S on track T.
5. a missed rotation (t_{rot}) plus negligible transfer time to reach and write S_{T-1} on track $T-1$.
6. a missed rotation (t_{rot}) plus negligible transfer time to reach and write S_{T+1} on track $T+1$.

Steps 3 and 4 together will take an integral number of rotations, either 1 or 2, depending on whether the disk is able to seek to the safe track, wait until the write location passes under the head, and seek back within a single rotation (t_{rot}). Based on discussions with disk vendors we assume the two steps will take $2t_{rot}$ to complete, for a random write latency of $t_{seek} + 5.5t_{rot}$. The same operation would take $t_{seek} + 0.5t_{rot}$ on a CMR drive, for a RMW overhead of 5 rotations

Performance is even worse for sequential write, as the previous write finishes just after writing sector S, so that step 1 will require an entire missed rotation, for a total latency of $6t_{rot}$. Note that multiple writes to the same track may be coalesced into a single RMW operation, whether via command queuing or the use of write caching on the drive; however it will still take 5 or 6 rotations longer than writing a full track on a conventional drive.

We note that in our analysis, the actual performance of RMW will be significantly worse than that implied by Hwang et al. [9], where they state that writes to bottom tracks will require two rewrites, for a mean of one extra rewrite per host write request. We attribute the inaccuracy of their analysis[2] to several factors: (1) the significance of missed rotations in the rewrite process, each of which is far more costly than all but the largest write requests; (2) the need to read top-track data so that it can be re-written, and (3) the need to persist top-track data in a secondary location, to avoid data loss from failure in the middle of a RMW operation.

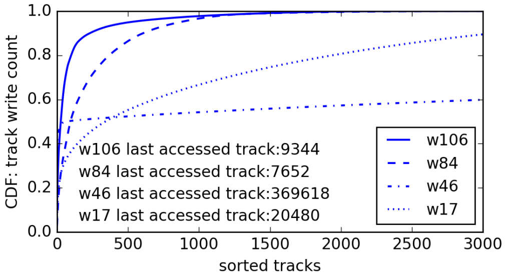


Figure 2: Track write count CDF (the first 3000 hottest tracks) for traces w17, w46, w84, w106 (See Section 3 for trace description); tracks are sorted from the hottest (i.e. track 0) to the coldest (i.e. track 3000).

2.2 Track flipping

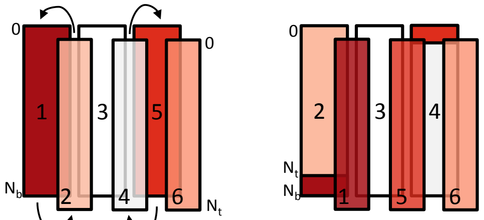
Real workloads show high locality, with typically a small number of *hot* sectors being overwritten frequently, and the remaining sectors receiving few if any writes; the same phenomena is found at the track level, as shown in Figure 2 (an illustration of the first 3000 hottest tracks in a few workloads). For instance, a significant portion of writes (80%) are received by a small number (100) of tracks in w106. We can take advantage of this locality by moving data between tracks to maximize the amount of hot data stored on re-writable top tracks. Our first algorithm, *track flipping*, locates bottom tracks containing hot sectors (i.e. *hot tracks*) and swaps them with adjacent top tracks, moving the hot data to the top, where additional writes can be performed directly, and (hopefully) moving cold data to the bottom track. In particular, we track the number of writes to each track, periodically identify candidates for flipping—i.e. hot bottom tracks which are adjacent to cold top tracks—and swap them. The actual swap of tracks T (bottom) and $T+1$ (top) is straightforward:

1. read tracks $T-1$, T, and $T+1$
2. write $T-1$ and T contents to a backup region[3]
3. write $T+1$ contents to T
4. write T contents to $T+1$
5. rewrite $T-1$

Implementation of this algorithm must take into account several real-world factors. Top and bottom tracks hold

[2] To be fair, their analysis is a minor paragraph in the middle of a magnetics paper.

[3] The backup region must accommodate at least two tracks.



(a) Before (b) After

Figure 3: Track flipping: hot bottom tracks 1 and 5 (red) are swapped with cold top tracks 2 and 4. Since top tracks (N_t sectors) are smaller than bottom tracks (N_b sectors), only the first or last N_t sectors of hot bottom tracks are moved.

differing amounts of data, with top tracks estimated to have 90% the capacity of bottom tracks. In addition, neighboring top tracks or bottom tracks may vary slightly in capacity, due to the presence of bad sectors hidden by *slip sparing* [10]—i.e. the LBA numbering skips a bad sector, resulting in a track containing fewer sectors than if it were perfect. We note that there are other variations in track capacity due to the use of zone bit recording [10] and adaptive formatting [11]; however in all but a negligible number of cases these will not result in differing capacities for adjacent tracks.

Our solution to differing track capacities is to swap *most* of the bottom track with the top track contents, as shown in Figure 3. If a bottom and adjacent top track hold N_b and N_t sectors respectively, then we can swap the first N_t sectors of the bottom track with the entire contents of the top track (tracks 4 and 5 in Figure 3). In the case where hot sectors are located at the "end" of the bottom track, we instead swap the last N_t sectors of the bottom track with the contents of the top track (tracks 1 and 2 Figure 3). Given the original location of a sector (i.e. sector position S on track T) the sector can be located precisely in the flipped configuration given knowledge of which flip (low LBA or high LBA) has been performed and the exact track sizes N_t and N_b, which are already known by the firmware as part of the LBA translation process. Note that once two tracks have been flipped, data cannot migrate any further; if tracks T and $T+1$ have been flipped, then flipping $T-1$ and T, or $T+1$ and $T+2$, is not allowed until T and $T+1$ have been flipped back.

Memory usage: The track mapping may be represented in a very concise fashion, as each bottom track T is in one of five states: (1) unmoved, (2) its low LBAs flipped with track $T-1$, (3) its high LBAs flipped with $T-1$, or (4) and (5), its high or low LBAs flipped with $T+1$. The resulting track map requires 3 bits per bottom track, or 1.5 bits per track; assuming a mean track size of 1.5 MB, this would require a map of about 2.5 MB for a 20 TB drive.

Memory requirements for hot track detection can be modest, as well. The total number of tracks is large, 1.3×10^7 for our 20 TB drive; however the number of tracks written in the period between iterations of the track flipping algorithm is much smaller (e.g. 20K in our experiments). Logging these track numbers in memory (using data structures such as an array or a list) will take less than 0.25 MB, and they may then be sorted and counted to determine track write frequency during that interval.

Safety and crash consistency: For track flipping we need to persist not only the state of the flipping process, but also updates to the track mapping. The flip process involves one more copy than RMW, but may be handled in the same way, by keeping an update log in the backup region, and marking sectors when they are copied back to their home (or flipped) locations. Backup region metadata can include a small log of map updates which can be appended to the track map in batches. To persist changes to the track map we keep a copy of the map on disk, and a log of updates to the map in the "checkpoint location". The checkpoint can be rewritten periodically and the log recycled, resulting in a negligible amortized cost for persisting map changes.

Timing: Assuming the head starts in an arbitrary location, the time required to flip bottom track T and top track T+1 will be:

1. t_{seek} to reach track T
2. $3t_{rot}$ to read tracks $T-1$, T and $T+1$[4]
3. t_{seek} to reach a backup region
4. $2t_{rot}$ to write backup copies of track $T-1$ and T
5. t_{seek} to return to track T
6. t_{rot} to write the contents of $T+1$ into track T
7. t_{rot} to write the contents of track T into $T+1$
8. t_{rot} to rewrite the contents of track T-1

for a total cost of $3t_{seek} + 8t_{rot}$. Since all accesses are to entire tracks, we assume that existing disk scheduling and buffering mechanisms allow reading or writing to begin immediately after reaching a track, rather than incurring additional rotational delay. Note, however, that track flipping is a background operation, and can be interrupted at any point in time—resuming an interrupted flip is very similar to the crash recovery scenario, except that in-memory state is still available. The primary performance impacts of track flipping are thus a reduction in overall throughput, from the background flipping process, in combination with RMW latency for those bottom-track writes to tracks which have not been flipped.

For track flipping to be effective, hot bottom tracks must be paired with neighboring cold top tracks, as there would be no advantage to flipping the two tracks of the same "temperature". Although one can easily construct synthetic workloads (e.g. uniform random) which lack neighboring hot/cold track pairs, we wish to determine whether they are found in real-world workloads. To address this question we analyze one of our experimental workloads (w17, described in the Section 3 below), assuming a constant track size of 2 MB. In Figure 4 we see write counts for the 20 hottest tracks and their neighbors.

[4] Seeks due to track switches as well as short seeks from track $T-1$ to track $T+1$ and vice versa are not included in our calculations.

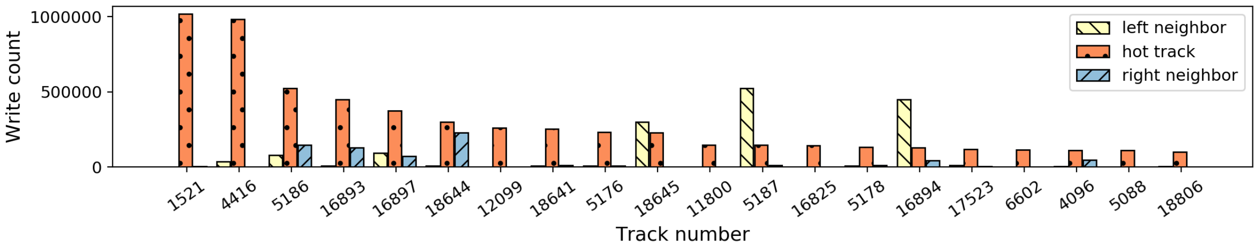


Figure 4: Write count of 20 hottest tracks and their neighbors, trace w17. This workload is seen to be "track flipping-friendly".

In only a few cases (e.g. track 5187) do hot tracks have a hot neighbor; however even in those cases the other neighbor is cold. Similar results are seen in many—but not all—other workload traces. However we note that results may vary with file systems other than the ones found in our traces, i.e. ext4 and NTFS, and will certainly vary with differing track sizes.

Real-world workloads are time-varying, with the identity of hot locations changing over time. We see this in Figure 5, which shows the write frequency over time for a range of 4 tracks. Not only does write frequency to a given track vary, but relative write frequency between tracks changes as well: e.g. track 3854 is much hotter than 3857 for a significant period, while later in the trace track 3857 is hotter. By using time-limited write counts, which are periodically reset after each search for hot tracks to flip, we are able to adapt to these changes in access frequency. In Algorithm 1 we see the full track-flipping algorithm: every N writes (e.g. 20,000) we select the hottest *k* bottom tracks over the last interval and, if possible, switch them with cold neighbors.

Algorithm 1: track flipping

```
parameter : updateFrequency, flipThreshold, maxFlips
variable  : ioDirection (read/write), trackPosition
            (bottom/top), trackNumber, writeCount,
            flipCount, trackIdLog [], trackCounts []
ioDirection, trackPosition, trackNumber $\leftarrow$ ReceiveIO()
if ioDirection $==$ write then
    writeCount ++
    trackIdLog.append(trackNumber)
    if writeCount mod updateFrequency $==$ 0 then
        trackCounts [] $\leftarrow$ Count (trackIdLog)
        for every track in Hottest (trackCounts) do
            flipCount ++
            middleCounter = trackCounts [track ]
            leftCounter = trackCounts [track-1]
            rightCounter = trackCounts [track +1]
            selected = Min (leftCounter, rightCounter)
            temperatureDiff = middleCounter- selected
            if temperatureDiff $>$ flipThreshold then
                TrackFlip (track, selected) maxFlips ++
            end
            if flipCount $>=$ maxFlips then
                Break ()
            end
        end
    end
end
```

2.3 Selective track caching

With track flipping—as with RMW—every track on the disk except for the two "backup region" tracks is filled with user data, requiring significant "data shuffling" to move data. If we instead reserve a small number of tracks for translation layer use, we can achieve additional gains in performance. *Selective track caching* does precisely this, reserving a small range of non-interlaced bottom-only tracks as a persistent cache for holding data from hot bottom tracks. Whereas track flipping is not able to move a hot bottom track if both of its neighbors are also hot or if both of its neighbors are already flipped with another bottom track, selective track caching is able to move any hot bottom track, at any time.

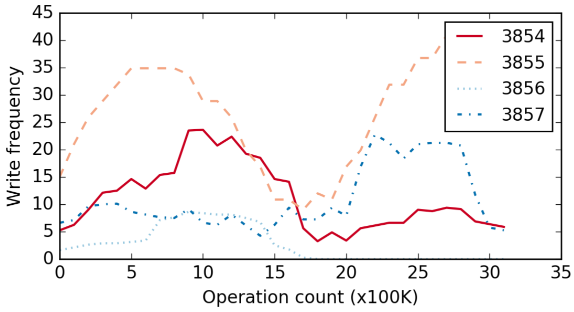


Figure 5: Track write frequency for tracks 3054-3057, trace w106. Y axis is the number of writes to a track out of 100,000 total writes.

More specifically, we reserve *k* bottom tracks as a random-write region (much like an SMR persistent cache), either at the outer diameter to maximize track size and transfer rate, or distributed in smaller groups across the disk to minimize seek time to the nearest cache. We note that tracks in the persistent cache will not be precisely the size of tracks that are cached there; instead we allocate "logical tracks" within the cache, where each logical track is a range of LBAs long enough to hold a full track and a metadata header. As seen in Algorithm 2, we again monitor track write counts, and periodically select the hottest bottom tracks to be moved to persistent cache, while moving the coldest cached tracks back to their home location.

Memory usage: For this algorithm, memory is needed for monitoring track write counts and for keeping a map of the cached tracks. The requirements for track write monitoring are the same as they are for track flipping, and thus the same approaches may be used with identical memory usage: hundreds of KB for logging the track numbers written in the period between iterations, or negligible usage if write tracking already performed by the drive is adequate. If the cache map is structured as a look-aside list of exceptions to the standard map, then its memory usage is proportional to the size of the persistent cache, not the drive itself. In our experiments a cache of 100 tracks was used, requiring a trivial amount of memory; however for a cache of several tens of thousands of tracks, memory usage should still remain in the range of a few MB.

Safety and crash consistency: The same approach may be used for maintaining a consistent copy of the map as for track flipping: updates are logged, and a full checkpoint written periodically. Alternately if the cache size is sufficiently small we can exhaustively scan the cache and rebuild the map on startup; however this requires t_{rot} per track, and starts to become impractical at cache sizes of less than 100 tracks.

Timing: The actual data movement portion of this algorithm is straightforward. Promotion of a bottom track to the cache merely requires seeking to it (t_{seek}), reading it (t_{rot}), seeking to the cache (t_{seek}), and writing it (t_{rot}), for a total of $2t_{seek}+2t_{rot}$. However track eviction takes longer as it requires a full-track RMW operation; the time taken will be:

1. t_{seek} to reach track C in the cache
2. t_{rot} to read track C
3. t_{seek} to reach track $T-1$
4. $2t_{rot}$ to read track $T-1$ and $T+1$
5. t_{seek} to reach to the backup region
6. $2t_{rot}$ to write backup copies of track $T-1$ and $T+1$
7. t_{seek} to seek back to track T
8. t_{rot} to write the contents of track C into T
9. $2t_{rot}$ to re-write track $T-1$ and $T+1$

for a total cost of $4t_{seek}$+$8t_{rot}$. In steady state one track will be evicted for every track promoted, for a total cost of $7t_{seek}+8t_{rot}$ per track promoted. Batching of promotions and evictions may remove several seek times from this total, but will not make great improvements due to the scattered locations of tracks being promoted or evicted. Again we note that promotions and evictions are interruptible background operations; the impact on host I/O will be a loss of throughput due to these operations, plus RMW latency for writes to non-promoted tracks.

Algorithm 2: selective track caching

```
parameter : updateFrequency, cacheSize
variable  : ioDirection, trackPosition,
            trackNumber, writeCount, trackIdLog
            [], cachedTracks [], trackCounts [], victim
ioDirection, trackPosition, trackNumber ← ReceiveIO()
if ioDirection == write then
    writeCount ++
    trackIdLog.append(trackNumber)
    if writeCount mod updateFrequency == 0 then
        trackCounts [] ← Count (trackIdLog)
        for every track in Hottest (trackCounts) do
            if track not in cache then
                victim ← Coldest (cachedTracks)
                TrackSwap (track, victim)
            end
        end
    end
end
```

2.4 Dynamic track mapping

Dynamic track mapping is another strategy for addressing the track flipping key limitations: (1) only neighboring tracks could be switched and (2) a small portion of bottom track must remain unflipped. It achieves this by allowing arbitrary permutations of tracks within *zones* (groups of small numbers of tracks).

To address unequal track sizes, in dynamic mapping we concatenate all bottom-track LBAs and group them in fixed-sized pseudo-tracks of approximately one physical track in size (except for the last pseudo-track). We similarly group all top-track LBAs into pseudo-tracks of the same size. These equal-sized pseudo-tracks may then be arbitrarily switched with each other. Algorithm 3 describes dynamic track mapping in more detail; as seen, we periodically check for hot bottom pseudo-tracks and swap the hottest bottom with the coldest top tracks.

Memory usage: If zones are sized to hold 256 pseudo-tracks, only 8 bits are needed for each map entry; for our 20 TB drive with almost 13M tracks this would require about 12.5 MB of memory: more than that needed for track flipping, but still modest.

Safety and crash consistency: Similar to track flipping and track caching, to persist the changes, dynamic track mapping logs the updates to the map and also writes a full checkpoint periodically.

Timing: The time required to swap a hot bottom track T with a cold top track T' in dynamic track mapping is very similar to that of track flipping. However, since pseudo-track size is not equal to physical track size, it is possible that a track cannot be read immediately after the head is placed resulting in a half rotation on average. The time required for a single swap will be:

1. $t_{seek}+0.5t_{rot}$ to reach track T
2. $3t_{rot}$ to read tracks $T-1$, T, and $T+1$
3. t_{seek} to reach a backup region

4. $3t_{rot}$ to write backup copies of track $T-1$, T and $T+1$
5. $t_{seek}+0.5t_{rot}$ to reach track T'
6. t_{rot} to read track T'
7. $t_{seek}+0.5t_{rot}$ to return to track T
8. $3t_{rot}$ to write the contents of T' into track T and write back the contents of $T-1$ and $T+1$
9. $t_{seek}+0.5t_{rot}$ to return to track T'
10. t_{rot} to write the contents of T into track T'

Thus, the expected total cost is $5t_{seek}$+$13t_{rot}$.

3 Methodology

We evaluate the four IMR translation algorithms—naive read-modify-write (RMW), track flipping, selective track caching and dynamic track mapping— in addition to conventional disk (CMR) via trace-driven simulation. We use the CloudPhysics traces [18], a recent set of block traces from virtual machines running Linux and Windows with modern file systems and large storage volumes. The LBA ranges covered in the traces varied from tens of gigabytes to 1.5TB.

Prior work [6] has shown that older traces (e.g. the widely-used MSR Cambridge traces [13] from c. 2007) display fine-grained behavior which is very different from that exhibited by modern file systems; although this difference in behavior may not be significant in block-level systems (e.g. FTLs) that ignore spatial locality, it has been demonstrated to result in significant differences in the performance of disk-based systems. The CloudPhysics corpus comprises 106 different traces; we sampled this set and selected a collection that represents different levels of read/write intensity and spatial locality. A summary of the selected workloads is shown in Table 1. The workloads range in size from about 3 to 44 million I/Os, and range from read-heavy (w08, 09% writes) to very write-heavy (w39, 95% writes).

Algorithm 3: dynamic track mapping

```
parameter : updateFrequency, swapThreshod
variable  : ioDirection (read/write), trackPosition,
            trackNumber, writeCount, trackIdLog
            [], trackCounts [], cldstTrk, hotstTrk
ioDirection, trackPosition, trackNumber ← ReceiveIO()
Function RemapTracks() is
    trackCounts [] ← Count (trackIdLog)
    for every cldstTrk in Coldest (trackCounts)
      and hotstTrk in Hottest (trackCounts) do
        temperatureGap = hotstTrkCntr- cldstTrkCntr
        if temperatureGap > swapThreshod then
            TrackSwap (cldstTrk, hotstTrk)
        end
    end
end
if ioDirection == write then
    writeCount ++
    trackIdLog.append(trackNumber)
    if writeCount mod updateFrequency == 0 then
        for every zone do
            RemapTracks()
        end
    end
end
```

Disk model: Our simulation assumes a 6000 RPM disk ($T_{rot}=10$ms) with equal-sized 2 MB tracks; although crude, we argue that this model is fairly accurate in the absence of real IMR disks for comparison. Based on current trends, 2 MB is a reasonable estimate of the mean track size for a next-generation drive; however real disks have decreasing track sizes towards the inner radius of the platter, with roughly a factor of two difference between the largest and smallest tracks. Since the majority of sectors lie in the larger outer tracks, the actual variance from the mean is less than this factor of two would imply, and as modern file systems (ext4 and NTFS) do not consider track location (as opposed to locality) in placement, errors in either direction are expected to cancel out.

The primary inaccuracy introduced by this model is in track flipping: the model assumes that top and bottom tracks are of equal size, so that no remainder of the bottom track is left behind after flipping. If top tracks in a real drive have 90% the capacity of bottom tracks, this would result in up to 10% of writes to this track being classified by the simulator as top-track writes, rather than bottom-track RMW writes. Since the colder end of the track is left behind, we expect that the misclassification rate be less than 10%. However, if both track ends are equally hot, the misclassification rate will be higher. Our observation of hot-track LBA access patterns suggests that either one end is extremely hot or all LBAs are accessed evenly. At present we neither know the actual ratio of top to bottom track size in a specific real IMR drive, nor the practical range of this parameter for feasible drives; therefore this 90% figure is highly speculative. Given this uncertainty, the choice of a uniform track size is simple and not unreasonable.

Trace playback: The traces used were collected in a virtualized environment, with a high-performance multi-disk (or SSD) back-end storage system. The resulting I/O rates may be seen in the inter-arrival time CDFs in Figure 6, where half of inter-arrival times for one trace (w84) are in the 100 μS range (up to 10,000 IOPS), while 80% of writes for another trace (w35) are below 500 μS (2000 IOPS). Several approaches may be taken in adapting such a trace to a single-disk simulation. One method is to run the simulation "flat-out", ignoring inter-arrival times and launching (or queuing) each I/O as soon as possible. However since our IMR translation algorithms include background work, the resulting behavior would not be representative of system behavior for real applications.

Our goal instead is to simulate what system behavior would be if the application that produced the original trace were run against the simulated (and much slower) I/O device.

Table 1: Statistical summary of selected workloads.

workload	w08	w09	w17	w24	w26	w28	w31	w34	w39	w43	w46	w48	w56	w61	w84	w87	w106
I/O count (M)	44.3	49.6	31.3	27.1	26.5	19.7	21.1	19.5	17.9	15.6	11.5	14	10.8	9.8	4.8	3.7	3.2
write ratio	0.09	0.55	0.78	0.11	0.58	0.33	0.16	0.23	0.97	0.51	0.62	0.42	0.95	0.50	0.86	0.79	0.82

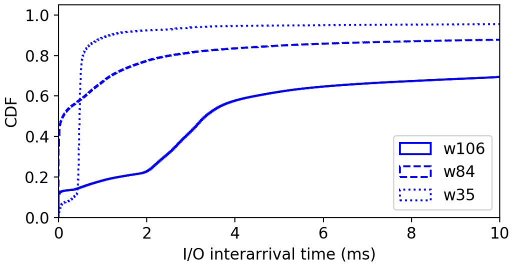


Figure 6: CDF of I/O inter-interval times in a few workloads.

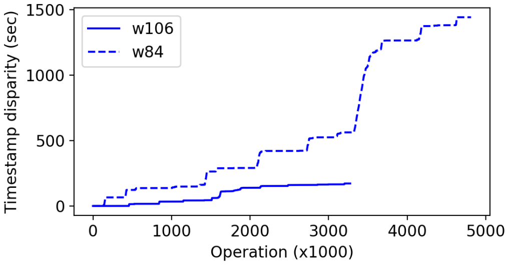


Figure 7: “Stretching” of original trace timestamps for IMR simulation.

Accurate simulation of such behavior requires information on application CPU usage and I/O dependencies which are not available in the original traces. Although recent work [5] provides mechanisms for inferring I/O dependencies given a trace containing both initiation and completion times for I/Os, very few of the traces we used contained this information.

Instead we consider a fixed-queue model, with a queue size of 64; based on inspection of the traces we believe this is an intermediate point between the lower effective queue depths seen with many application-initiated I/Os, and the very high degree of parallelism with which I/Os are initiated by the virtual memory system. I/O inter-arrival times are preserved when inserting items into the queue until it is filled, and then further I/Os are blocked until a position in the queue opens; in other words, the inter-arrival time between I/Os N and N+1 is the maximum of the original trace inter-arrival time and the time for the queue to drain by 1. The resulting I/O performance reflects a combination of individual I/Os and queuing delays due to device throughput limitations. As an example, in Figure 7 we see this “time dilation” for two workloads with naive RMW—in one case trace completion is delayed by about 150s, and in the other by over 1440s.

3.1 Disk model details

I/O latency: In our simulation, I/O latency includes host and device queuing, seek time, rotational delay and transfer time. Seek time (T_{seek}) is calculated from the source and destination track locations using the following equation proposed by Shafaei [16], assuming minimum and maximum seek times of 2 and 20 ms and calculating α accordingly:

$$t_{seek}(trk_{src},trk_{des}) = \alpha * \sqrt{|trk_{src} - trk_{des}|} + t_{seek_{min}} \qquad (1)$$

Rotational delay for I/Os on different tracks is assumed to be uniformly distributed between 0 and 1 rotation ($T_{rotation}$); for deterministic simulation we assume a constant value of a half rotation, giving a total I/O latency of:

$$t_{I/O} = t_{seek} + \frac{1}{2} t_{rotation} + t_{transfer} \qquad (2)$$

We note that IOs are split at boundaries in case they touch more than a track. The full list of drive specifications and experiment configurations is shown in Table 2.

4 Evaluation

In this section we evaluate five alternatives—conventional disk (CMR), IMR with naive read-modify-write (RMW), IMR with track flipping, IMR with selective track caching and IMR with dynamic track mapping—by measuring I/O latencies and write amplification factor (WAF). Summary results for all traces and algorithms are shown in Figure 8 (write amplification) and Figure 9 (latency).

All track flipping, selective track caching and dynamic track mapping are seen to give substantial improvements in write

Table 2: Experimental parameters and drive specification.

	drive specification	
track size	drive cache size	rotation delay
2MB	100MB	10 ms
	dynamic track mapping and flipping configuration	
update frequency	hot/cold threshold	max flips
20K write ops	50	50
	selective track caching configuration	
update frequency	track cache size	cache location
20K write ops	100 tracks	OD

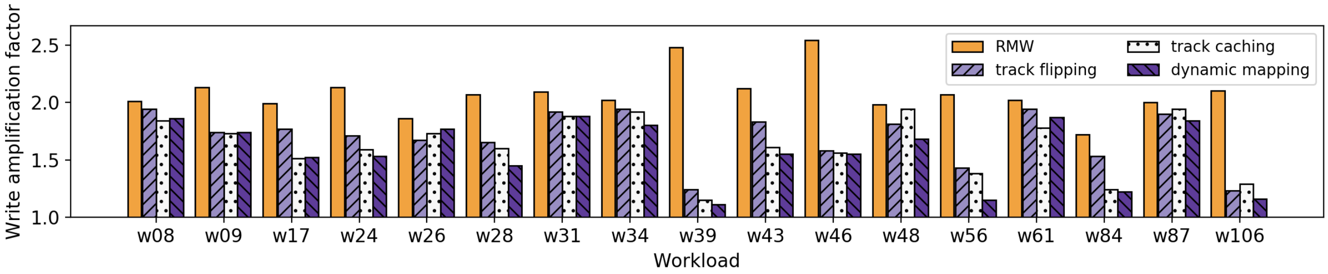
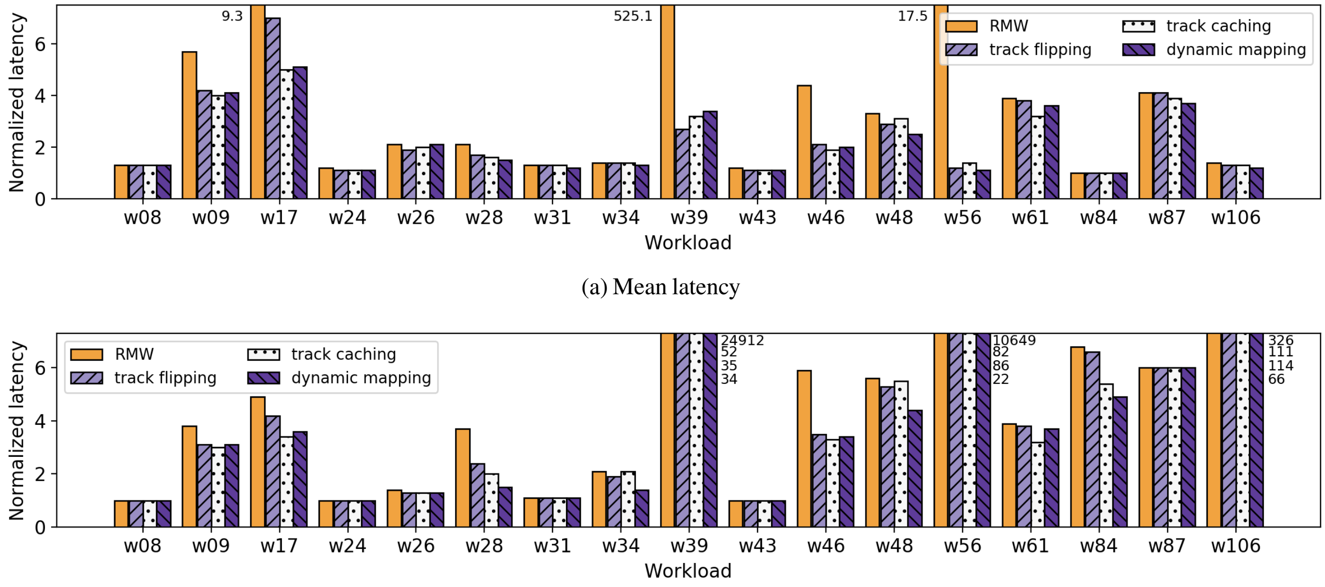


Figure 8: IMR write amplification by workload and translation layer.



(a) Mean latency

(b) 90th percentile latency

Figure 9: Mean and tail latency for 17 workloads: CMR, RMW, track flipping, track caching and dynamic track mapping.

amplification when compared to naive read-modify-write, by a factor of 2 or more in over half of the cases; in no case is performance degraded. As expected, dynamic track mapping shows the best performance in most cases as it has the minimum limitations among the proposed approaches. For several workloads it reduces the write amplification by a factor of 2 or more; in some cases (e.g., w56, w39 and w106) write amplification is nearly eliminated.

Results for mean latency are more mixed. In several cases (e.g., w09, w17, w39, w46, w56, w61 and w87) IMR read-modify-write latencies were noticeably higher than for the conventional drive. For w56 this excess latency was virtually eliminated by track flipping, track caching or dynamic track mapping. For some others (e.g., w17, w39, w46) at least one of the approaches gave a significant reduction (more than 2x) in IMR latency, while still remaining about twice that of conventional. For w09 and w87, however, latency improvements from dynamic track mapping were modest, with performance still significantly worse than CMR.

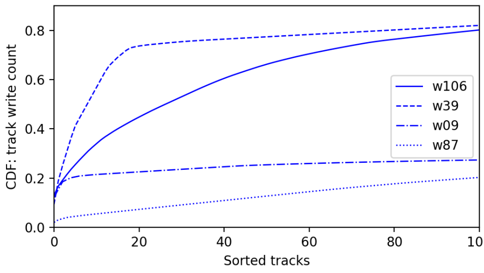


Figure 10: CDF of track write counts for workloads with highest (w106 and w39) and lowest (w87 and w87) WAF improvement with track flipping and/or selective track caching.

In Figure 10 we explore one possible reason for lower improvement for traces w09 and w87: the overall number of hot tracks. We compare these traces with w106 and w39, the traces that show the greatest improvements in write amplification when track flipping or track caching is used (see

Figure 8). We see that a high fraction of writes in w106 are to the hottest 100 tracks, and that half of the writes in w39 are to a handful of tracks. In contrast the "working set" of hot tracks for w09 and w87 appears to be very large, with only around 20% and 15% of writes going to the hottest 100 tracks, respectively. The track cache used in our experiments (100 tracks) would do almost nothing to help in this case, and if many writes are to tracks which are written only a few times at most, then the gain from flipping them will not outweigh the cost.

As with mean latency, results for tail latency (Figure 9b) are also mixed, although worse. In a few cases (e.g., w39 and w56) RMW increases tail latency by a factor of tens of thousands. That is because with CMR most writes in these cases appear to complete in the write cache which results in a mean latency of about 0.1ms, and there is just enough idle time for them to be flushed to disk; however with RMW the disk cannot keep up during idle time, so the queue fills and stays that way causing mean latency of about 2.4s. Overall tail latency is increased by more than 4x across majority of the workloads. In the worst cases (w39, w56) the proposed algorithms limit the relative increase in tail latency to double digits, but is still very high. In about half the cases tail latency with dynamic track mapping is similar to that of CMR; however in the remainder it is significantly worse.

For both mean and tail latency, there are a few workloads (w08, w24, w31 and w43) that are not affected by IMR and accordingly no improvement is observed when the proposed algorithms are applied. Besides w43 with almost 51% of write operations, the rest of the workloads are read-heavy traces and therefore are less prone to significant performance penalty due to IMR.

To examine further, in Figure 12 we see CDFs of I/O latencies for traces w46 and w28. In each case IMR read-modify-write results in high-latency I/Os due to a combination of operation latency and queuing delays due to reduced throughput; roughly 2/3 of writes were slowed in both cases. We note that there are some other cases e.g., w56 with fewer writes (roughly 10%) being affected. Track flipping, track caching and dynamic track mapping are all able to improve the w28 and w46 performance considerably, but a significant fraction of writes (roughly 25%) still suffer excessive latency. Our observations show that for the case w56 the three improved algorithms are all able to eliminate the excess latencies, resulting in performance comparable to a conventional drive.

The impact of IMR overhead on throughput can be approximated by looking at the issue time expansion during the simulation run; this indicates the periods at which the device was unable to keep up with the I/O trace, and by how much. In Figure 13 we see issue time disparity vs. I/O count for traces w46 and w28. We note that for w46 all dynamic track mapping, and to a slightly lesser extent track caching and track flipping, result in significantly improved throughput. For w28, dynamic track mapping shows a considerable throughput improvement; track caching and track flipping show a smaller improvement. We also note that throughput of the simulated algorithms would increase with larger I/Os, as read-modify-writes would be amortized over larger I/O sizes.

Track flipping only works if hot tracks are adjacent to cold tracks; if hot regions on the disk are substantially larger than a track, this might not be the case. In Figure 4 of Section 2.2 we saw that this was the case for trace 17; however in Figure 11 we see the same analysis for w87; although the hottest few tracks stand alone, many tracks with very high write counts are surrounded by tracks of similar hotness.

5 Related work

Interlaced Magnetic Recording [9] is a new storage technology using HAMR (Heat assisted magnetic recording) [4, 12] and track overlap (the technique on which SMR [19] is based) to achieve higher areal density than possible with either approach alone [3]. Numerous works have characterized [1] and modeled [15, 16] SMR performance; however due to fundamental differences in track layout and write constraints such work is not directly applicable to IMR.

While only a limited number of translation layers and data management techniques have been proposed for IMR in the two years since the original work became public [20], a wide range of file systems and translation layers have been proposed for SMR, such as Cassuto's indirection system [2], SMaRT [8] from He and Du, Shafaei's Virtual Guard [14], and FSTL [7]. Cassuto et al. propose a set associative persistent cache to hold updated sectors. SMaRT [8] proposes using a track-based dynamic mapping. Shafaei et al. propose a track-based static mapping translation layer which caches tracks containing at-risk data, rather than the track targeted by the I/O. Hajkazemi et al. [7] propose an LBA-based translation layer based on dynamic mapping. Since the write restrictions in SMR are a strict superset of IMR restrictions, SMR translation layers could in fact be applied to IMR; however this would ignore the performance improvements possible due to lessened write restrictions.

To the best of our knowledge, the data management design introduced by Wu et al. [20] is the only published work on IMR translation layers to date. The authors propose *Top-Buffer*, a technique utilizing unallocated top tracks of each track-group (a small set of tracks interlaced with top tracks) as a buffer to store LBA updates corresponding to bottom tracks. Moreover they suggest *Block Swapping*, a technique to swap bottom hot LBAs with cold ones within a track-group. Our work differs in that it is targeted for in-disk implementation, in a restricted-memory environment, while the memory requirements for Wu et al.'s algorithm are beyond the capabilities of a drive controller.

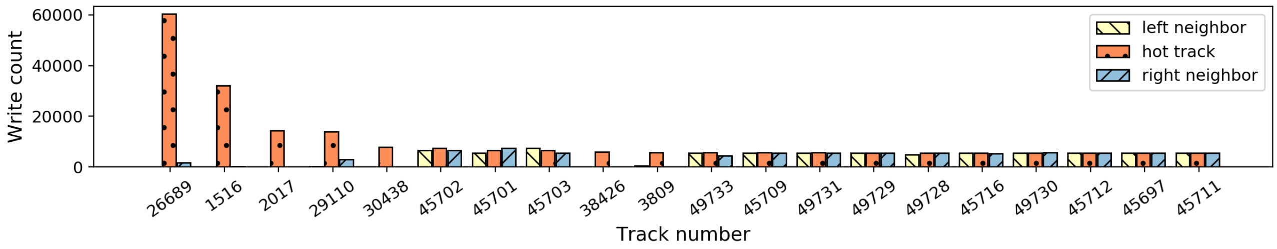
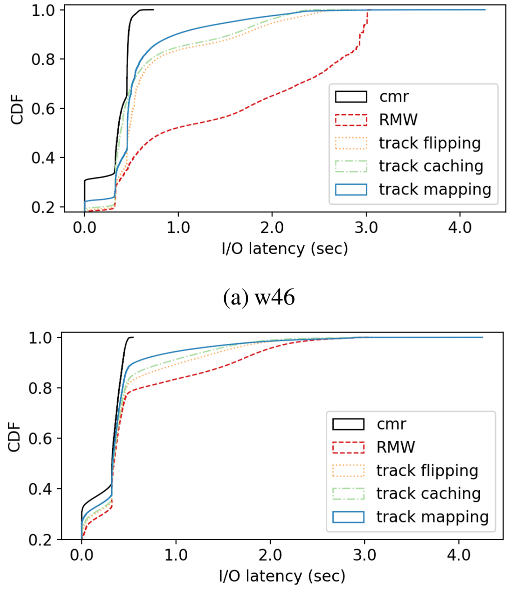


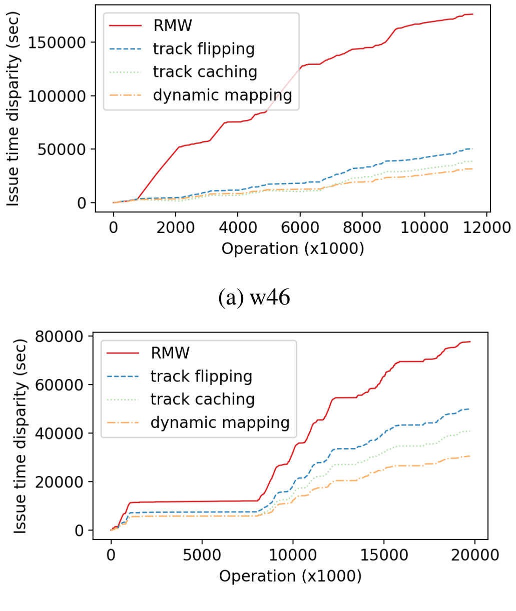
Figure 11: Write count of 20 hottest tracks and their neighbors, trace 87. This trace is seen to be "track flipping-unfriendly".



(a) w46

(b) w28

Figure 12: I/O latency distribution: workloads w46 (a) and w28 (b), for CMR, IMR/RMW, IMR/track flipping, IMR/track caching.



(a) w46

(b) w28

Figure 13: Issue time disparity (I/O issue time gap between CMR and other studied approaches) of traces w46 and w28.

6 Conclusion

Interlaced magnetic recording is still a new—or even speculative—technology, and we cannot be sure of its precise characteristics until real prototypes are available. However, when such prototypes arrive, algorithms will be needed to cope with the IMR write restrictions, and due to the track-based nature of the restrictions, those algorithms will need to run in the memory-limited environment of the drive controller.

We quantify the performance of the naive read-modify-write algorithm for IMR bottom track writes, showing that it is significantly more costly than assumed in prior work, and show via trace-driven simulations that for some workloads its performance is comparable to that of a conventional disk, but that it is worse, sometimes catastrophically so, for others. We present three algorithms to reduce the frequency of IMR bottom-track writes: *track flipping*, *selective track caching* and *dynamic track mapping*, with sufficiently modest memory requirements to be readily implemented in drive controllers. These algorithms are shown to improve I/O amplification significantly for almost all workloads examined, and to improve latency for some—but not all—of the workloads which performed poorly with IMR read-modify-write. Further research is needed to determine whether extensions of this work (e.g. track flipping+caching) will yield conventional drive-level performance for IMR with acceptable memory cost.

Acknowledgment

We would like to thank Irfan Ahmad and CloudPhysics for the use of their traces, our shepherd William Jannen, and the anonymous reviewers for their valuable suggestions.

References

[1] Aghayev, A., Shafaei, M., and Desnoyers, P. Skylight—a window on shingled disk operation. *ACM Transactions on Storage (TOS) 11*, 4 (2015), 16.

[2] Cassuto, Y., Sanvido, M. A. A., Guyot, C., Hall, D. R., and Bandic, Z. Z. Indirection systems for shingled-recording disk drives. In *Proceedings of the 2010 IEEE 26th Symposium on Mass Storage Systems and Technologies (MSST)* (Washington, DC, USA, 2010), MSST '10, IEEE Computer Society, pp. 1–14.

[3] Granz, S., Jury, J., Rea, C., Ju, G., Thiele, J., Rausch, T., and Gage, E. C. Areal density comparison between conventional, shingled, and interlaced heat-assisted magnetic recording with multiple sensor magnetic recording. *IEEE Transactions on Magnetics 55*, 3 (March 2019), 1–3.

[4] Granz, S., Zhu, W., Seng, E. C. S., Kan, U. H., Rea, C., Ju, G., Thiele, J.-U., Rausch, T., and Gage, E. C. Heat-assisted interlaced magnetic recording. *IEEE Transactions on Magnetics 54*, 2 (2018), 1–4.

[5] Haghdoost, A., He, W., Fredin, J., and Du, D. H. C. On the Accuracy and Scalability of Intensive I/O Workload Replay. In *15th USENIX Conference on File and Storage Technologies (FAST 17)* (Santa Clara, CA, Feb. 2017), USENIX Association, pp. 315–328.

[6] Hajkazemi, M. H., Abdi, M., and Desnoyers, P. Minimizing read seeks for smr drives. In *Proceedings of the 2018 IEEE International Symposium on Workload Characterization* (2018), IEEE.

[7] Hajkazemi, M. H., Abdi, M., Shafaei, M., and Desnoyers, P. Fstl: A framework to design and explore shingled magnetic recording translation layers. In *Proceedings of the 26th IEEE International Symposium on the Modeling, Analysis, and Simulation of Computer and Telecommunication Systems (MASCOTS '18)* (Oct. 2018), IEEE.

[8] He, W., and Du, D. H. SMaRT: An approach to shingled magnetic recording translation. In *Proceedings of the 6th USENIX Conference on File and Storage Technologies* (Santa Clara, CA, 2017), USENIX Association, pp. 121–134.

[9] Hwang, E., Park, J., Rauschmayer, R., and Wilson, B. Interlaced magnetic recording. *IEEE Transactions on Magnetics 53*, 4 (2017), 1–7.

[10] Jacob, B., Ng, S., and Wang, D. *Memory Systems: Cache, DRAM, Disk*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2007.

[11] Krevat, E., Tucek, J., and Ganger, G. R. Disks Are Like Snowflakes: No Two Are Alike. In *Proceedings of the 13th USENIX Conference on Hot Topics in Operating Systems* (Berkeley, CA, USA, 2011), HotOS XIII, USENIX Association, pp. 14–14.

[12] Kryder, M. H., Gage, E. C., McDaniel, T. W., Challener, W. A., Rottmayer, R. E., Ju, G., Hsia, Y.-T., and Erden, M. F. Heat assisted magnetic recording. *Proceedings of the IEEE 96*, 11 (2008), 1810–1835.

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

[14] Shafaei, M., and Desnoyers, P. Virtual Guard: A Track-Based Translation Layer for Shingled Disks. In *9th USENIX Workshop on Hot Topics in Storage and File Systems (HotStorage 17)* (Santa Clara, CA, 2017), USENIX Association.

[15] Shafaei, M., Hajkazemi, M. H., Desnoyers, P., and Aghayev, A. Modeling smr drive performance. In *Proceedings of the 2016 ACM SIGMETRICS International Conference on Measurement and Modeling of Computer Science* (New York, NY, USA, 2016), SIGMETRICS '16, ACM, pp. 389–390.

[16] Shafaei, M., Hajkazemi, M. H., Desnoyers, P., and Aghayev, A. Modeling drive-managed smr performance. *ACM Transactions on Storage (TOS) 13*, 4 (2017), 38.

[17] Thompson, D., and Best, J. The future of magnetic data storage techology. *IBM Journal of Research and Development 44*, 3 (May 2000), 311–322.

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

[19] Wood, R., Williams, M., Kavcic, A., and Miles, J. The feasibility of magnetic recording at 10 terabits per square inch on conventional media. *IEEE Transactions on Magnetics 45*, 2 (2009), 917–923.

[20] Wu, F., Zhang, B., Cao, Z., Wen, H., Li, B., Diehl, J., Wang, G., and Du, D. H. C. Data Management Design for Interlaced Magnetic Recording. In *10th USENIX Workshop on Hot Topics in Storage and File Systems (HotStorage 18)* (Boston, MA, Feb. 2018), USENIX Association.