Indirection Systems for Shingled-Recording Disk Drives

Yuval Cassuto*, Marco A. A. Sanvido*, Cyril Guyot*, David R. Hall† and Zvonimir Z. Bandic*
Hitachi Global Storage Technologies
* San Jose Research Center
3403 Yerba Buena Road
San Jose, California 95135 USA
† Advanced Magnetic Recording Laboratory
3605 Hwy 52 N
Rochester, Minnesota 55901 USA
Email: {*yuval.cassuto, marco.sanvido, cyril.guyot, david.hall, zvonimir.bandic*}*@hitachigst.com*

Abstract—**Shingled magnetic recording is a promising technology to increase the capacity of hard-disk drives with no significant cost impact. Its main drawback is that random-write access to the disk is restricted due to overlap in the layout of data tracks. For computing and storage systems to enjoy the increased capacity, it is necessary to mitigate these access restrictions, and present a storage device that serves unrestricted read/write requests with adequate performance. This paper proposes two different indirection systems to mask access restrictions and optimize performance. The first one is a disk-cache based architecture that provides unrestricted access with manageable drop in performance. A second, more complex indirection system, utilizes a new storage unit called S-block. It is shown that the S-block architecture allows good sustained random-write performance, a point where the disk-cache architecture fails. The organization and algorithms of both architectures are specified in detail. Each was implemented and simulated as a discrete-event simulation, mimicking its operation on real storage devices. For the performance evaluation both synthetic workloads and traces from real workloads were used.**

I. Introduction

Magnetic hard disk drives have been around for more than half a century, an unusually long time for an information-age technology. Their impressive survival as ubiquitous devices in various markets is attributed to the wondrous areal density scaling they have sustained. Each leap in areal density required significant novelty and effort across the many employed disciplines, but from the computing and storage systems' perspective, the scaling challenges remained latent deep behind the interface connector. Moreover, each increase of linear bit density between drive generations has led to higher read/write data rates, and to better performance observed by the host. A common wisdom in the disk drive industry is that the physical scaling processes used so far will soon fail to deliver significant additional capacity gains [1]. Newer magnetic storage technologies, such as bit-patterned media (BPM) [2] and heat-assisted magnetic recording (HAMR) [3], hold the promise to scale beyond conventional media, but neither has yet matured to product-level functionality. A different path toward higher recording densities is offered by the technology of *shingled magnetic recording* (SMR). Without a dramatic rework of the media, the SMR technology enables the usage of write heads with stronger fields and higher tolerances, thereby allowing increasing the grain density without compromising the stability of the written bits [4, Sec. 3]. The main caveat of the SMR technology is that it introduces interference between adjacent tracks. More concretely, because of the wider signature of the shingled head, data tracks are laid out with overlap, such that writes to a given track affect bits previously written in proximate overlapping tracks (hence the adjective *shingled* refers to roof shingles, commonly laid out in overlapping rows.). Consequently, writing to a randomly chosen physical block cannot be performed without care to affected neighboring blocks. Shingled magnetic recording allows even higher bit densities when combined with two-dimensional magnetic recording (TDMR) [4], though this paper treats the case of shingled recording in conjunction with conventional, random-access read. The details of the interference model and access limitations of SMR are provided in section II.

The prospect of a double-digit, between 15% to 40% depending on head and media design, percentage increase of storage capacity with shingled recording motivates a study on introducing SMR devices into computing and storage systems. The premier challenge, from the systems point of view, is the mitigation of the write-access restrictions, as imposed by the physical characteristics of the shingled-recording modules. Depending on the specific storage system/application, the restrictions on random-write physical access may be a significant or minor issue. But in any case, there is a need for a solution to guarantee the integrity of the written data without assuming

978-1-4244-7153-9/10/$26.00 © 2010 IEEE

that the logical write access from the host will be appropriately restricted. The observation that the advent of SMR devices warrants a system study is not a new one. In a high-level survey of system ramifications from shingled recording (and TDMR) [5], Gibson and Polte point to the need to introduce a software layer to mask access restrictions from the host system, in a similar way that the flash translation layer (FTL) masks access restrictions in solid-state storage devices [6]. Such a software layer, specifically designed and evaluated for magnetic storage devices, is the subject of the current paper. The general method by which shingled-access restrictions can be mitigated is by implementing an *indirection system*, defined in section III. The indirection system can be implemented either inside the storage device or at a higher layer on the host (similarly to the JFFS2/UBIFS file systems for flash storage), though a preferable design would be one that is implemented by the storage device itself, and will thus have a complete knowledge of the instantaneous physical layout of data blocks on disk.

The focus of study in this paper is on achieving good read/write performance in the presence of shingled-recording constraints. The fact that SMR drives are likely substitutes of traditional magnetic disk drives, sets the perfomance of the latter as the baseline for evaluating the former. The standard performance criteria for hard-disk drives are mentioned in subsection III-A, and are used as a guide for the design and evaluation of the shingled-recording architectures. The two proposed indirection architectures: shingled set-associative disk cache (section IV), and circular buffers with S-blocks (section V), both use known computer-science concepts that have appeared before in a variety of applications. Caching schemes of different structures and flavors are discussed in [7, Part 1], and circular buffers are commonly used in log-based file systems [8]. Nevertheless, the manner in which these concepts are utilized in the proposed indirection systems is unique to the properties and constraints of shingled magnetic recording. For each of the two proposed indirection systems, the discussion consists of architectural details, algorithms, and simulated-performance results, followed by a short discussion. With a moderate increase in complexity, the S-blocks architecture is shown to solve key issues of the disk cache architecture, such as sustained random-write performance, cache bypass for sequential writes, and data consistency after power failure. Beyond the introduction and analysis of two specific indirection architectures, the objective of this paper is to provide design insight on the general problem of shingled access in performance-sensitive storage systems.

II. Shingled Recording Access Fundamentals

A. Interference profiles

The pivotal manifestation of shingled recording is the "squeezing" of the tracks below the width of the write head, hence magnetic field from a write to a given track may introduce errors to sectors in neighboring tracks. The resulting interference regime imposes access restrictions upon the shingled-recording device, causing it to lose the block random-access property of standard magnetic disk drives. The *interference profile* of a shingled-recording device depends on the particular head and media components that it employs, and the purpose of this section is to characterize that interference and propose access restrictions that would prevent data loss in its presence. The following defines the physical interference profile of a shingled-recording device.

Definition 1 (Physical Interference Profile) *A physical interference profile is defined as the quadruple* $\mathcal{I}_p = [s_{\text{id}}, s_{\text{od}}, \phi_{\text{cw}}, \phi_{\text{ccw}}]$*, that specifies, relative to a given write location, the area affected by the write.* s_{id} *and* s_{od} *are the number of tracks affected in the direction of the inner-diameter (ID) and outer-diameter (OD), respectively,* $\underline{\textit{including}}$ *the currently written track.* ϕ_{cw} *and* ϕ_{ccw} *are the circumferential-direction boundaries of the affected area in the neighboring tracks.*

An example of a physical interference model is provided in Figure 1, with $s_{\text{od}} = 3$ and $s_{\text{id}} = 1$. Notes:

1) In a workable shingled device either $s_{\text{id}} > 1$ or $s_{\text{od}} > 1$, but not both. If both are greater than one, then it is not possible to utilize all the tracks, even with access restrictions.
2) The number of affected tracks include the current track because s_{id} or s_{od} are sometimes referred to as the *head width* in tracks.
3) If $s_{\text{id}} > 1$ we say that the *shingling direction* is from OD to ID. If $s_{\text{od}} > 1$ we say that the shingling direction is from ID to OD.
4) A physical interference profile is not constant across the drive. For example, with a proper head design, the drive can be formatted such that $s_{\text{id}} > 1$ in one part and $s_{\text{od}} > 1$ in another, and then the shingling direction changes as well.
5) A non-shingled drive, as a special case, has $s_{\text{id}} = s_{\text{od}} = 1$.

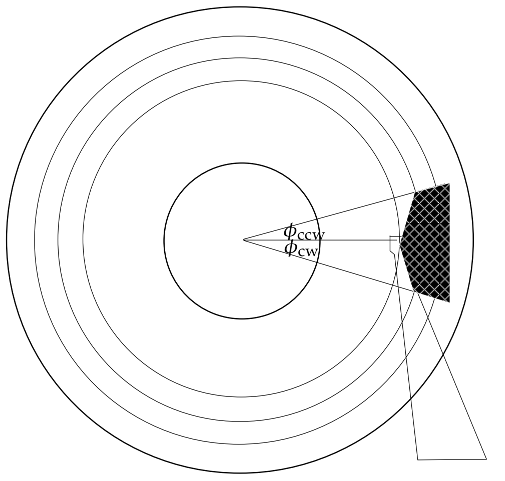

Fig. 1. Sample physical interference profile of a shingled drive: $\mathcal{I}_p = [1, 3, \phi_{\text{cw}}, \phi_{\text{ccw}}]$

The main thrust of this article is to study drive architectures that mitigate performance degradation due to restricted access in a *general* shingled drive. Therefore, it is most convenient to abstract out the detailed physical properties of the drive, and adopt a more "logical" viewpoint. The logical viewpoint assumes that the physical data blocks are numbered as a one-dimensional sequence of Physical Block Addresses (PBAs). Moving from two-dimensional layout of data to one dimension allows for a succinct specification of shingled-recording constraints. Hence the architectures and algorithms proposed herein will enjoy sufficient generality to apply to a wide variety of physical interference profiles. The definition of the (one-dimensional) *block interference profile* now follows.

Definition 2 (Block Interference Profile) *A block interference profile $\mathcal{I}_b$ is defined as the integer number of PBA addresses, starting from the write location address, that are affected by the current write.*

Figure 2 illustrates the block interference model for the example of $\mathcal{I}_b = 6$. The block interference profile implicitly

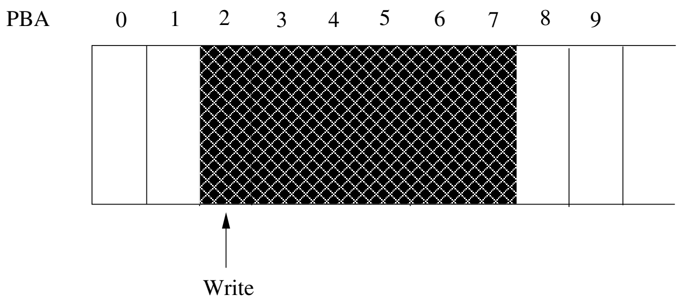

Fig. 2. Sample block interference profile of a shingled drive: $\mathcal{I}_b = 6$

assumes that the shingling direction is in the direction of increasing physical block addresses. A linear ordering of blocks *can* capture the parameters s_{id} or s_{od} by setting $\mathcal{I}_b$ to be the number of blocks in the adjacent s_{id} or s_{od} tracks. However, this ordering *fails* to capture the locality offered by the parameters ϕ_{cw} and ϕ_{ccw}. That said, utilizing the angular locality for better architectures is non-trivial, if one wants to obtain sequential read/write performance comparable to a non-shingled drive (employing angular-selective writing implies non-sequentiality of the physical writing and reading.). In addition, for some write-PBAs the block interference profile also adds subsequent blocks on the *same track* to the interference profile (while in a real shingled device only blocks on adjacent tracks are affected). Nevertheless, alleviating the above over-strengthening of the interference profile introduces an unrealistic requirement of having a different interference profile for each PBA, since the true interference profile changes, for example, between the first and last blocks of a track.

B. Constrained access in shingled devices

Suppose that a host command requests to modify a small chunk of data on the shingled drive. Given the interference profiles above, writing this small chunk of data will erase subsequent blocks of valid data. So a fundamental limitation of shingled devices is their inability to carry out simple update-in-place write operations. The challenge of designing a storage device that employs shingled recording is then to allow an unrestricted write access from the host's perspective, despite the restrictions on the physical write process.

Read-Modify-Write
A rudimentary solution to the write-access problem above is the *read-modify-write* operation. As its name implies, the read-modify-write operation first reads a portion of data from the disk, then modifies part of that portion with the host-provided write data, and finally writes the whole portion back to disk. The portion size of data used by the read-modify-write operation is chosen to be the smallest that will not erase valid data blocks outside the portion. Note that in the worst case, when all the disk currently stores valid data, read-modify-writes may require to operate on a full disk surface. The performance penalty due to large read-modify-write operations can be bounded by the following introduction of the *shingled region* concept.

Shingled Regions
The size of read-modify-write operations can be bounded by partitioning the disk surface to independent shingled regions. A shingled region (also called *region* for short) is a group of tracks that is separated from neighboring shingled regions by a guard band. The purpose of the guard band is to prevent a write in a given region to interfere with data written on other regions. That isolation of interference guarantees that no read-modify-write will need to go beyond a region boundary. Small shingled regions thus improve performance by limiting the read-modify-write size required for small host writes. On the other hand, small shingled regions imply large disk capacity devoted to guard bands instead of data. This tradeoff between capacity and performance is not unique to read-modify-write access, but will be a common thread in all architectures discussed later in the paper.

The read-modify-write approach, even with the implementation of guarded shingled regions, does not provide a satisfactory solution to the write restrictions due to shingled recording. This is because the large ratio between the shingled-region size and the smallest-write size implies significant decrease in write performance. Consequently, more sophisticated access architectures are required to mitigate the performance hindrances associated with shingled recording.

III. Shingled-Recording Architectures with Indirection

The reason for the poor performance of read-modify-write is that unless the write request is to a block at the end of the shingled region, many blocks (on average half) need to be read and re-written. What then if we could organize matters such that *every* write will go to the end of the shingled region? Having no valid data blocks beyond the write address is obviously desirable, since no additional reading and writing are needed in addition to the actual write request. Tempting as it be, given that host writes are completely unrestricted, the physical layout of data blocks will need to be made dependent on the write workload into the storage device. No longer can a host-side Logical Block Address (LBA) be mapped to a fixed[1], statically computable, PBA. Rather, the relationship between an LBA and PBA becomes indirect. To maintain the mapping between LBAs and PBAs, an *indirection system* is implemented, whose definition now follows.

Definition 3 (Indirection System) *An indirection system is a collection of data structures and algorithms that* assigns *physical locations to logical block addresses and* retrieves *physical locations of logical block addresses.*

All indirection systems provide the same simple functionality above, but each one is designed to meet specific system-level objectives, under resource consumption constraints (e.g. memory used for data structures, time required to map between logical and physical addresses). In the case of shingled recording, the indirection system is to be designed to provide good read/write performance for a wide variety of natural workloads. More detailed discussion of performance objectives follows in sub-section III-A. A key component of indirection systems is *garbage collection*. This term refers to operations invoked by the indirection system to reclaim resources, or transition the system to a more desirable state.

While new to magnetic disk drives, indirection systems are commonly used in NAND Flash based storage devices to level device wear, and to enforce sequential page writes within Flash blocks [6].

A. Design objectives

The ultimate goal of implementing an indirection system for shingled recording is to present to the host a storage device that has essentially the same performance behavior as a non-shingled drive. If this can be done without a significant added cost (due to resources required for the indirection system), then the storage-capacity gain thanks to shingled recording is deemed attractive to the device user. Unfortunately, the seemingly innocuous term "performance behavior" turns out to be very hard to define, as storage devices are used in different types of systems, under different usage conditions, and with different performance expectations. With that difficulty in mind, we turn to itemize a list of conditions under which the shingled recording architecture ought to provide good performance. This by no means is an exhaustive list that captures all drive usages, nor we claim that strictly all of these conditions should be considered. This list merely reflects the authors' experience with the interaction between storage devices and different computing systems. Choosing the right evaluation criteria is a known challenge in the storage community [9], but for the sake of completeness, we detail the main criteria we considered for the work reported here.

Random Read/Write

Random reads/writes, whereby the sequence of read/write requests from the host has no predictable pattern, is an important workload to evaluate the performance of a shingled drive. Different sizes (in blocks) of read/write requests (also called I/Os) should be considered, and the performance of the drive will be measured in the average number of host I/Os Per Second (IOPS) that the drive serves. Two particular random workloads of interest are one that is localized in space, and one that is localized in time. A workload *localized in space* means that the LBAs in the request sequence are taken from a sub-range of the drive LBA range. The space locality is motivated by the large ratio between the total storage device capacity and the amount of storage accessed by a typical host application (or even by multiple applications in multi-threaded or multi-port systems). A workload *localized in time* means that a continuous random workload that is spread across the full LBA range does not last a long time before it changes to a more structured workload. It is admittedly possible to envision random workloads that are localized neither in space nor in time, but such extreme workloads are in the scope of a very small slice of the storage market that is not the natural target of shingled-recording drives.

Sequential Read/Write

Sequential reads/writes, whereby the sequence of read/write requests from the host are addressed to a contiguous sequence of LBAs, are key workloads to evaluate a shingled drive. For magnetic disk drives in general, the sequential access mode provides the best read/write throughput, since it obviates the need to carry out time-consuming head seeks between requests. Therefore, any indirection system for a shingled drive should provide similarly high read/write throughput for the sequential access mode.

Workloads from Traces

The two previous access modes are used to evaluate the drive performance using synthetic workloads with well-defined properties. A complementary way to evaluate performance is to run "natural" workloads using traces collected from real-system usage. The advantage of the trace workloads is that they provide a mix of random and sequential access modes, as well as less structured access modes. Another benefit from traces is that they include information on the timing of requests to the drive, and this can be utilized to optimize the scheduling of background operations performed by the indirection system. The main challenge with trace-based performance evaluation is selecting the right traces to run, and arguing that these traces are good representatives of real-life workloads.

B. Experimental evaluation

An experimental study of the proposed indirection architectures comes with two objectives.

1) Design: Assist the choice of an architecture and its parameters.
2) Analysis: Predict the behavior of real shingled-recording devices under variable workload conditions.

For the experimental results that follow, we use natural read/write workloads collected from traces, as well as synthetic random and sequential workloads, to aim at the two research

[1]Standard non-shingled disk drives can change physical location of LBAs due to defects, but such re-mappings are rare, small-scale events.

objectives above. Processing a sequence of read/write commands from these workloads, the resulting physical read/write commands issued by the proposed indirection systems are simulated and logged. The experimental method is *discrete-event simulation*, whereby the simulator mimics the operation of the specified indirection system by maintaining its state and following its actions at all times during the progression of the workload. A block diagram of the implemented simulator is given in Figure 3. It comprises two main blocks: the

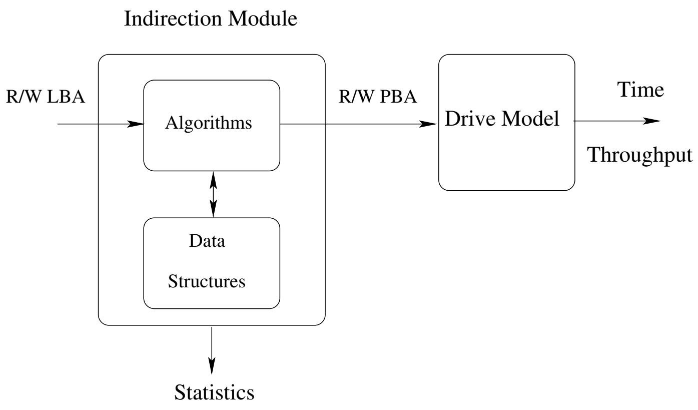

Fig. 3. A block diagram of the simulator used for evaluation of shingled-recording architectures.

core indirection module mapping LBAs to PBAs (left), and a physical-drive model to provide access-time information for the resulting physical commands (right). To compare the performance of the shingled architecture to a standard non-shingled drive, the same drive model is also used with LBA accesses as inputs. The outputs of the simulator are logs of PBA access, the time those accesses took to execute (or the average R/W throughput for the workload), and also internal statistics on the indirection system itself (frequency and size of garbage-collection operations, memory usage and other state data). For the quantification of PBA access, three units are counted, each for read and write. These counts are compared to the same counts for the LBA access.

1) Number of read/write commands of an arbitrary size in blocks.
2) Number of block read/writes: the total number of blocks that are read/written for the workload.
3) Number of adjusted read/writes. Each adjusted read/write is a read/write of a 1000 or less consecutive blocks.

Read/write command counter (number 1 above) treats a consecutive read/write of an arbitrary length as a single event. On the other extreme, the block read/write counter (number 2) only counts block accesses without regard to their relative locations. Neither of these measurements provides a good insight on the overall excess load of writing to a shingled architecture, compared to a non-shingled one. On one hand, the block read/write counter ignores the fact that in magnetic recording the time cost of accessing sequential blocks is much smaller than accessing the same number of arbitrarily located blocks. On the other hand, counting only the number of commands introduces bias in the other direction, since very long read/write commands (e.g. a full shingle region) require significantly more time than single-block commands. The *adjusted read/write* counters (number 3), on the other hand, do capture the performance cost of the workload, hence they are used to define the *write overloading* evaluation criterion defined below. The 1000 blocks/count limit is chosen since the time to read/write 1000 blocks is in the same order as a seek time between non-adjacent blocks. The advantage of the write overloading quantifier is that it can serve as a rough performance predictor without complex assumptions on seek times, physical placement of cache regions, and other physical properties of the disk-drive.

Definition 4 (Write Overloading) *For a given architecture and workload, the write overloading is calculated as*

$$\mathrm{write\_overloading} =$$

$$\frac{\mathrm{adj\_PBA\_write} + \mathrm{adj\_PBA\_read} - \mathrm{adj\_LBA\_read}}{\mathrm{adj\_LBA\_write}}$$

To the number of adjusted PBA writes we add the difference between the number of adjusted PBA reads and the number of adjusted LBA reads. This difference is an estimate to the excess of adjusted reads used in garbage collection operations. By counting all the excess reads as outcome of write operations, we implicitly assume that the read performance of the shingled architecture is identical to a non-shingled drive. The overall ratio is thus the number of physical writes and reads resulting from host write commands divided by the number of host writes. The write overloading factor is similar to the *write amplification* factor used for the wear-analysis of Flash storage [10], only that here we also count the excess reads in addition to the excess writes.
In addition to PBA statistics, we also use a drive model to measure the total simulated physical access time (seek+read/write) of the full workload, and in addition the average throughput in different-size time windows. It is important to note that the purpose of this initial experimental study of shingled-recording is to primarily understand first-order phenomena in shingled access. For that reason, the reader's emphasis should not be the quoted absolute performance values achievable by shingled drives, but more their performance behaviors and design tradeoffs.

IV. Shingled Access with Set-Associative Disk Cache

The objective, stated in the opening of section III, to write incoming requests beyond valid data in the shingled region, lends itself well to a disk-cache based solution. In such a solution, each logical block has a fixed native physical location in a shingled region, and a chunk of contiguous LBAs is mapped to a chunk of contiguous PBAs in a native region. Additionally, a small number of the shingled regions are provisioned to be used as a cache for incoming writes. In each cache region, writes will be addressed to consecutive physical blocks starting from the first block of the region, with no need for auxiliary reads/writes until the region is filled. Upon filling of the cache region, the cached blocks are written to their native physical locations, by invoking read-modify-write events to the regions that contain their respective native locations. At that point all the blocks in cache are invalidated, and the cache is vacant again for new incoming writes. The indirection system tracks the valid locations of logical blocks, and serves reads from cache/native locations accordingly. Data caching in general is a ubiquitous and highly-effective performance boosting method, implemented in a variety of computing systems [7]. However, *disk-cache* solutions, whereby data is cached onto the magnetic media itself has received far less attention in the literature [11],[12]. Subsequent sub-sections aim to study and optimize disk-caching for shingled disk drives.

A. Architectural details

To get from the concept of disk-caching to a working indirection system for shingled drives, one needs to make some architectural decisions. The first issue to resolve is how to assign LBAs to cache regions[2]. First, since a filling of a cache-region results in a read-modify-write of a *full* native region, all LBAs in a native region should be assigned to the same cache region. Second, since the total cache size is much smaller than the drive capacity, while the cache region is of similar size to a native region, then multiple native regions should be assigned to each cache region. To exploit the spatial locality of access to the drive, the group of native regions assigned to a cache region span logical addresses from the whole LBA space of the drive. More precisely, if the ordering of native regions by their LBA ranges is $[N_0, N_1, \dots, N_{n-1}]$, then native region N_i is mapped to cache region $i \bmod M$, where M is the number of cache regions. This assignment allows a workload on a local LBA range to utilize the entirety of cache space. A diagram of the set-associative cache layout is provided in Figure 4, for an example with $M = 2$ cache regions.

Once the assignment policy of native regions to cache regions is decided, the two parameters that need to be set are the percentage of disk space provisioned to cache, and the size of the shingled regions. Both these parameters offer trade-offs between capacity and performance. A high percentage of cache provisioning reduces the device capacity, but allows fewer and less frequent read-modify-write events due to cache fills. Similarly, a small region size implies shorter read-modify-write events, but consumes more of the disk surface for guard bands between regions. To this end, the cache allocation percentage and region size are left as optimization parameters of the architecture. In later sub-sections, the performance of shingled drives will be evaluated under different choices of these parameters.

[2]Pre-assigning native blocks to a subset of cache locations is called in the literature *set-associative caching*.

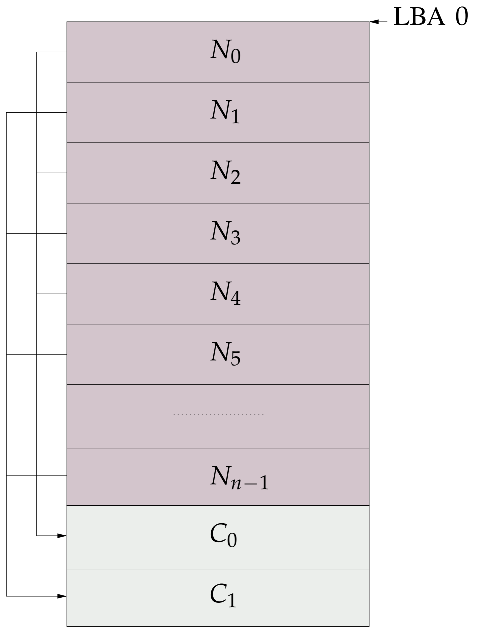

Fig. 4. Disk cache layout and association of native regions to cache regions. Even-numbered native regions are mapped to C_0 and odd-numbered ones are mapped to C_1.

B. Algorithms

Given the organization of the indirection system described in the previous sub-section, the way it handles read/write requests is now detailed. The simplicity of the set-associative disk cache architecture contributes to the simplicity of the algorithms it employs. The basic algorithms needed to serve read/write workloads now follow. For a write operation,

```
Algorithm IV.1: write_block
  Input: LBA
  $i$ = native_region(LBA) $\bmod M$
  if is_full($C_i$) then
      garbage_collect($C_i$)
  end
  Append(LBA,$C_i$)
```

described in Algorithm IV.1, if the cache associated with the region of the input LBA is full, a garbage_collect operation (specified in Algorithm IV.2) is invoked before appending the data block at the end of the cache region. Garbage collection in this architecture is the invocation of read-modify-write operations on all the native regions that have valid LBAs present in the cache. After completing the read-modify-write operations, the cache is cleaned by invalidating all of its blocks.

Read operations are served by the Algorithm IV.3. The function cache_lookup returns the cache physical block address of LBA if it is valid in the cache, and NOT_IN_CACHE otherwise. native(LBA) returns the (fixed) native physical address of block LBA.

```
Algorithm IV.2: garbage_collect
  Input: $C$
  foreach $N_i$ present in $C$ do
      read_region($N_i$)
      modify_with_cached_blocks($N_i$)
      write_region($N_i$)
  end
  Invalidate_all_blocks($C$)
```

```
Algorithm IV.3: read_block
  Input: LBA
  $i$ = native_region(LBA) $\bmod M$
  PBA_c = cache_lookup(LBA,$C_i$)
  if PBA_c $\neq$ NOT_IN_CACHE then
      read(PBA_c)
  end
  else
      read(native(LBA))
  end
```

C. Simulation results

We start the experimental study of the set-associative disk cache architecture with running a workload collected from 24 operation hours of a personal computer serving an individual employee in a corporate environment. Initially, we provide sample results for a fixed set of parameters to exemplify the different performance evaluation criteria introduced in subsection III-B. Subsequently, we use the evaluation criteria to compare the performance of the architecture under different choices of parameters.

The chosen parameters for the initial results are a shingled region size of 50000 blocks (25 MB) and 1% of the storage space used as disk-cache. The total number of physical-storage (512 Byte) blocks is $6 \cdot 10^8$, amounting to a storage capacity of 307 GB. The chosen capacity does not necessarily represent the capacity of real SMR products, but rather, is used to match the footprint of the simulated benchmarks. With those parameters, the results are summarized in Table I. The results of the shingled set-associative disk cache architecture are listed in the right-most column, in comparison to the results in the center column, of a standard non-shingled device running the same workload. As the results show, employing the shingled architecture with these parameters results in a write overloading factor of 3.8 and total-time slowdown[3] of factor 3.48. This reasonable performance degradation is in stark favor compared to a pure read-modify-write architecture that results in 3-4 orders of magnitude degradation for the same region size.

Measurement	non-shingled	set-associative disk cache
# reads	435,285	438,758
# writes	75,615	77,713
# block reads	8,394,244	112,343,297
# block writes	2,889,969	105,787,800
# adjusted reads	435,565	542,968
# adjusted writes	75,615	180,613
total time [sec]	623.24	2172.81

TABLE I
COMPARISON BETWEEN SHINGLED SET-ASSOCIATIVE 1% DISK CACHE AND NON-SHINGLED ARCHITECTURES FOR A SAMPLE PARAMETER SET.

We next wish to examine the effect of the total size of the shingled disk cache on read/write performance. For that we take a similar workload, which is more write intensive than the one used for Table I's results. We fix the architecture parameters as above (drive size, region size), and vary the percentage of storage used as disk cache from 1% to 10% in steps of 1%. We expect the write overloading and slowdown factors to be smaller for larger cache sizes, since read-modify-writes of native regions are amortized over more LBA writes. The results of that experiment are given in Figure 5, where both the write overloading (solid line) and the total-time slowdown (dashed line) are plotted. The two curves in Figure 5

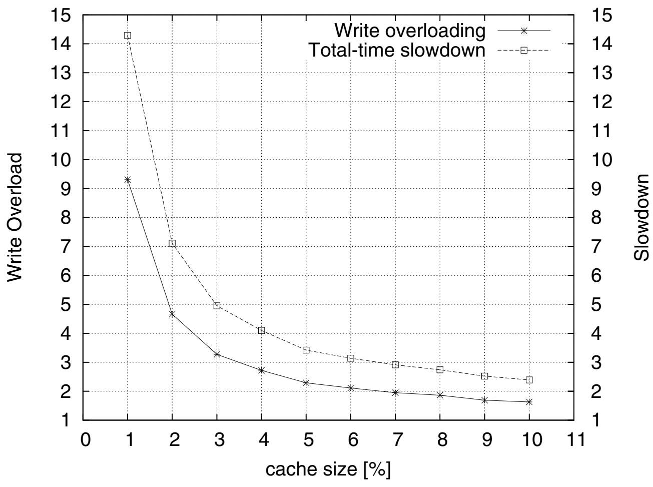

Fig. 5. Write overload (solid) and total-time slowdown (dashed), as a function of cache size.

show that both measures of performance degradation have a similar monotone decrease as the cache size increases. Moreover, the convexity of the curves testifies that as we keep increasing the cache size, the return we achieve in performance improvement diminishes. Note that it is possible to make the curves closer to each other by changing the threshold for adjusted read/writes from 1000 to a lower value. However, we refrain from doing so to avoid the danger of fitting to specific seek model or workload.

Moving to analyze the random-write behavior, we fix the cache size to 1% and issue a workload of 4 KB ($8\times$ 512 Byte blocks) write requests, to randomly chosen LBAs uniformly distributed across the LBA space. The request size of 4 KB is chosen as a typical access unit, with the expectation that a different unit would change the absolute results, but not

[3]Note that comparing the total execution time assumes a continuous workload with no idle time (which is not true in reality knowing that the workload amounts to 24 hours of drive access)

their core characteristics. We measure the average number of 4 KB IO (write) operations per second (IOPS) served by the shingled drive. The IOPS values are measured in windows of 10[sec] (all time units are in workload time and not simulation time). As the plot in Figure 6 shows, the first 780 seconds of the workload enjoy a very high write throughput, until the cache regions fill up, at which point the performance drops dramatically to values under 50 IOPS. Worse yet, there are many 10-second intervals with 0 IOPS values, meaning that no host writes were served at these time intervals. When the indirection system "catches up" with garbage collection operations, the original performance is regained, until the next drop. The reason for the high variation in performance is that

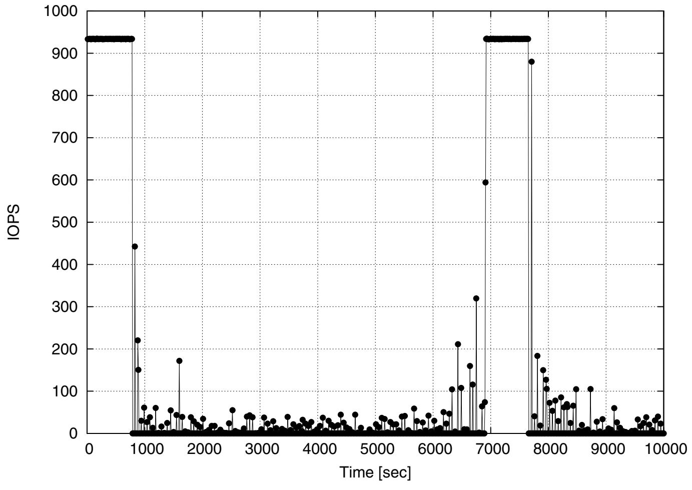

Fig. 6. Random write 4 KB IOPS in 10[sec] intervals.

in a random workload, the cache regions are likely to have contributions from *all* of the associated native regions (in the case of 1% cache allocation, each cache region is shared by 99 native regions), and upon cache fill-up, very many read-modify-write operations need to be performed prior to acceptance of new write commands. This problem can be partially mitigated by reducing the region size, hence shortening the time of individual garbage collection operations. To see the effect of smaller region size, the same experiment is repeated with the shingled region size changed from 50000 to 10000. The difference between the two region sizes can be seen by comparing Figure 7 to Figure 8. The results in Figure 7 are the same as Figure 6, zoomed on the low IOPS range. In summary, reducing the region size does not improve the overall average IOPS for the total test time, but improves the drive response by eliminating instances of 0 IOPS.

Our next goal is to examine the performance of read, and more specifically, sequential read workloads. Recall that the shingled-recording model considered in this paper assumes no restrictions on physical reads. However, the read performance may still depend on the indirection system's physical organization of data. This issue is epitomized by sequential read workloads, which exhibit different behavior in the presence of disk cache. To study this issue, we pre-filled the cache

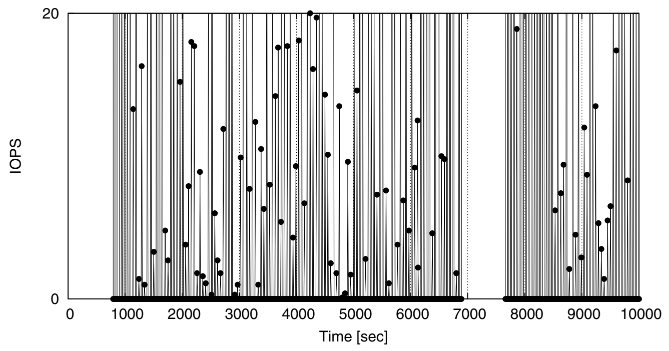

Fig. 7. Low IOPS with region size of 50000.

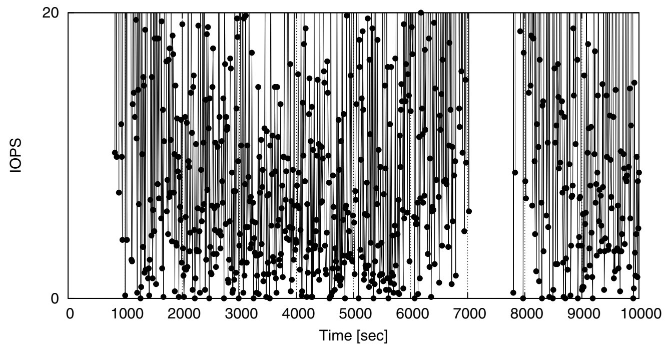

Fig. 8. Low IOPS with region size of 10000.

regions with *randomly* written LBAs, and subsequently issued a pure sequential read workload. The average read throughput in [MB/s] was measured, and the experiment was repeated for 6 different amounts of cache fill between 0% (empty) and 100% (full). The results, found in Figure 9, indicate that sequential read performance suffers from the cached random writes, even for fairly low percentage of cache fill. Admittedly, the scenario of pure sequential read after pure random write is not a highly realistic one in practice. Therefore, the results of Figure 9 should be taken as an extreme worst case not representing the behavior under typical workloads. We note that if the test is performed after *sequential* pre-filling of the caches, then the performance is similar to a non-shingled drive, with negligible dependence on the mount of cache fill.

D. Discussion

The main claim of this section is that employing a simple indirection architecture can move shingled recording from the impractical realm of read-modify-write operations, to a more manageable regime of trading off (a small amount of)

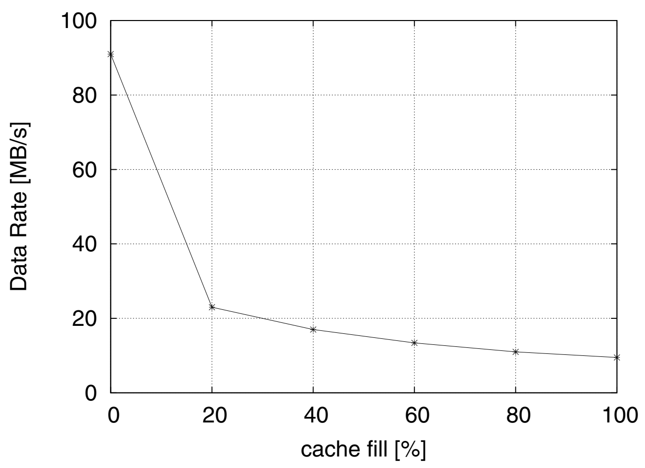

Fig. 9. Sequential read after random write with different amounts of cache fill.

disk-cache space with performance. Going beyond this initial study of shingled disk-cache indirection architectures, many interesting research problems lie ahead. For example, it is possible to improve performance by departing from the fixed associativity of the caches, and allowing a dynamic, workload dependent, association of native regions to cache regions. There is also a plethora of practical issues not addressed herein, such as protecting the consistency of data in cases of power failure. A read-modify-write of a native region may result in inconsistent data if interrupted prematurely.

V. Shingled Access with S-blocks

The disk-cache based architecture proposed in section IV has the merits of conceptual clarity and implementation simplicity. However, for workloads that are not spatially local, it suffers the shortcoming of requiring many full-region read-modify-write operations upon filling of a cache region. This implies that the drive will not be able to serve an additional write command before completing all the read-modify-write operations for the destination cache region. Shortening the outage time by invoking read-modify-write on only some of the regions does not solve the problem, since blocks from different native regions are interspersed in the cache. This issue raises the need to devise an architecture that exhibits more graceful garbage collection events. The ungraceful behavior of the shingled disk-cache architecture arguably stems from the large gap between the storage unit in the disk cache (single blocks) and the region size (order of 50K blocks). This gap limits the flexibility of the indirection system and results in abrupt transitions from very good performance to very bad one, or even complete irresponsiveness. The solution to this problem is to add an intermediate storage layer in between single blocks and full regions, which we call *S-blocks* – short for Shingled Blocks. S-blocks store multiple blocks with consecutive LBA addresses on a continuous range of PBA addresses. The mapping between the start LBA and start PBA of an S-block is *not* fixed, and is managed by the indirection system to optimize performance. Like other parameters in the system, the size of the S-blocks is set as a performance vs. resource tradeoff. A small S-block allows more flexibility, and potentially better performance, but consumes more memory resources for the indirection system to track S-block locations. S-block size of order 1000 blocks may serve good balance between flexibility and resource parsimony.

On a high level, the proposed S-block architecture is not conceptually dissimilar to the shingled disk-cache architecture of section IV. Similarly it stores data blocks either in a cache area of the disk, or in their native location, which is now an S-block. So the main difference is that in the S-blocks architecture the native physical locations of logical blocks *can* change, as part of the full S-block unit. On a lower level, when considering the mapping of S-blocks and cache areas to physical shingled regions, the S-blocks architecture departs considerably from the shingled disk-cache architecture. In order to reap the promised flexibility and tunability of S-block usage, S-blocks and cache blocks need to be laid out onto shingled regions in a way that will allow their efficient relocation while maintaining the access constraints of shingled recording. For that purpose, the shingled regions will no longer be a pure sequentially-written entities, and will instead be managed as circular buffers with maintained guard bands. The details of the S-blocks architecture, including description of circular-buffer use are given in the next sub-section.

A. Architectural details

As noted earlier in the section, organizing data in S-blocks is beneficial if S-blocks can be relocated without compromising other S-blocks due to shingled-recording interference. The plain sequential shingled region used in the disk-cache architecture of section IV is too rigid to allow individual-S-block relocation without a read-modify-write operation. Therefore, to mitigate the rigidity of the sequential shingled region, we instead manage it as a circular buffer with maintained guard band, as in the following definition.

Definition 5 *A circular buffer with guard band Λ is a contiguous range of M blocks with pointers head and tail that satisfy*

$$(tail - head) \bmod M \geq \Lambda$$

As its name suggests, a circular buffer is most conveniently depicted as a circle, shown in Figure 10. The shaded blocks represent blocks in use and the white blocks represent vacant ones. The empty shaded blocks are blocks that are in use and *valid*. The shaded blocks crossed with an $\times$ are used but *invalid* blocks. Used invalid blocks are ones that are not needed (since, for example, a newer version of the block is stored elsewhere), but cannot be reused without a proper manipulation of the *head* and *tail* pointers. Addition to the circular buffer is done at the *head* pointer, and removal is done from the *tail* pointer. Both pointers move in the clockwise direction: *head* upon addition and *tail* upon removal of blocks. Because of the guard-band requirement, once *head* is Λ blocks

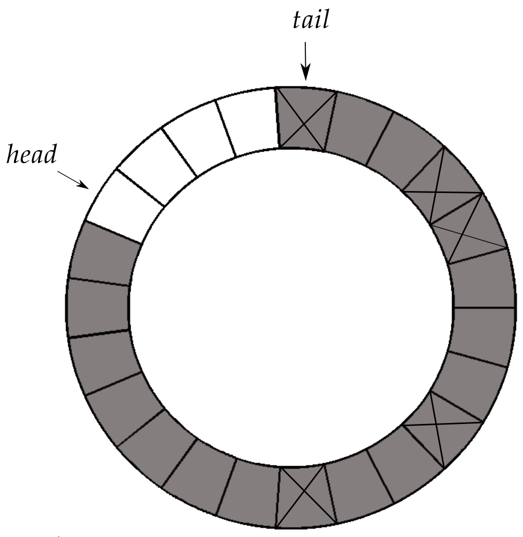

Fig. 10. A circular buffer has head and tail pointers. New blocks are written at the head, old blocks are removed from the tail.

away from *tail* in the counter-clockwise direction, no new blocks can be written before removal of used blocks from the tail. If Λ is set to be at least the block interference profile $\mathcal{I}_b$ (from section II), then managing a shingled region as a circular buffer clearly guarantees that written blocks do not affect the integrity of existing blocks in the region.

A key element in the proposed S-blocks architecture is to maintain two *separate* circular buffers: one for block writes (cache) and one for S-blocks (native locations). The orders of magnitude size difference between blocks and S-blocks suggests that different policies and algorithms may be appropriate for one over the other. Hence separating them yields a layered design that is flexible to optimize the circular-buffer management separately for blocks and S-blocks. Another architectural decision is to divide the full LBA range of the drive into a number of sub-ranges, called *sections*, each of which independently manages a pair of block and S-block circular buffers. Such a division can be seen as treating the full storage device as multiple independent devices, and has the advantage of saving memory resources, and bounding the severity of performance drops for worst-case workloads that are relatively local in space. As illustrated in Figure 11, the section's two circular buffers are the *cache buffer* that stores block writes, and the *S-block buffer* that stores native data in S-blocks. The S-block buffer is initialized to sequentially store all the section's range of LBAs in units of S-blocks. The cache buffer is initialized empty. All the white areas in Figure 11 represent physical storage that is provisioned beyond the reported device capacity to serve the indirection system. Over-provisioned storage in the S-block architecture comes in two forms: space to store additional copies of S-blocks in the S-block buffer, and the space used for the cache buffer. Each of the two buffers is assumed to be managed independently

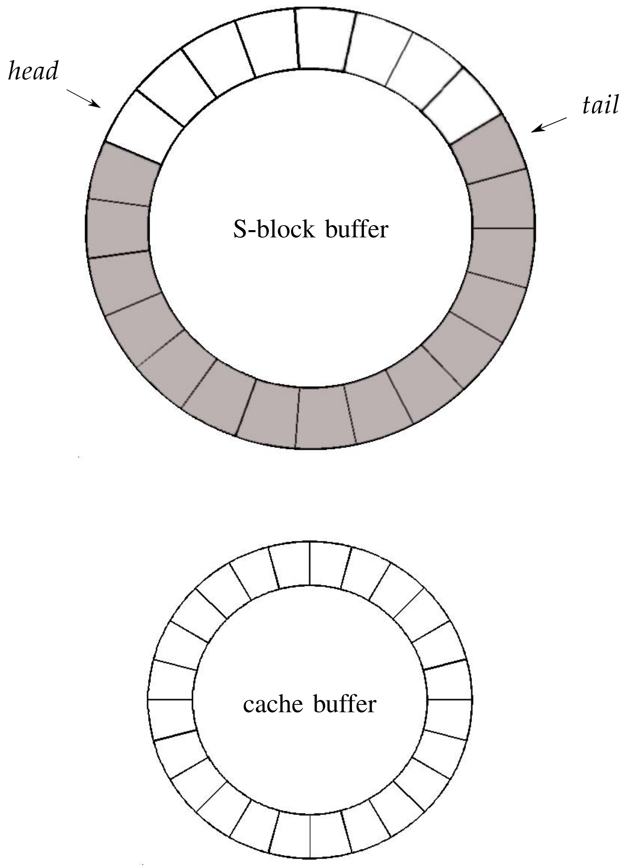

Fig. 11. A drive section with two circular buffers: S-block buffer for native data stored in S-blocks, cache buffer for caching individual-block writes.

from the other, and also from buffers of other sections. Hence every buffer shall be separated from others with a guard band.

The parameters that govern the implementation of the S-blocks architecture are now listed.

- Total physical storage
- Percent of total storage used for over-provisioned S-blocks
- Percent of total storage used for cache buffers
- Number of sections
- Size of S-block

These 5 parameters define a specific instance of the architecture. Different assignments to these parameters result in different storage capacities and different performance profiles. The problem of choosing the parameters is discussed with an experimental lens in sub-section V-D.

B. Algorithms

The purpose of introducing more flexible data organization in the S-blocks architecture is to allow its management to be tailored to a variety of workload conditions. This implies that a wider space of indirection-handling algorithms can – and should – be explored in the design. The response to read/write commands, as well as to internal need of garbage collection, can be made dependent on the type of workload we wish to optimize for. The behavior can also change and evolve to fit an instantaneous workload whose properties are not known a priori. Before delving into the details of specific algorithmic decisions, we present the general operation of the S-blocks indirection system.

Cache Buffer

Incoming writes are added to the *head* of the cache buffer. At some point in time, a process is invoked to invalidate used blocks in the cache buffer by re-writing them, with their full S-block unit, in the S-block buffer. This process is referred to as *group destage*. On or before the instance when no more writes can be added to the cache buffer due to *head-tail* proximity, the cache buffer invokes a process called *buffer defrag*. The purpose of buffer defrag is to push the *tail* pointer farther clockwise from the *head* pointer by removing used blocks from the buffer. When *tail* points to a valid block, this block is copied to the head and both *head* and *tail* are progressed clockwise. Space is actually freed up when *tail* reaches an invalid block, which allows progressing the *tail* pointer without copying a block and moving the *head* pointer.

S-block Buffer

An S-block is added to the S-block buffer as a result of a group destage operation in the cache buffer. On or before the instance when no more S-blocks can be added to the S-block buffer due to *head-tail* proximity, the S-block buffer invokes a buffer defrag operation, similarly to the cache buffer. The difference is that in the S-block buffer there is no explicit destage operation, since re-writing an S-block automatically invalidates the existing copy of the S-block.

With the understanding of the general functionality of the cache and S-block buffers, it is now possible to provide a detailed description of the indirection system's algorithms. In the presentation below, Algorithms V.1-V.2 describe operations in the cache buffer. Algorithms V.4-V.5 describe operations in the S-block buffer, and Algorithm V.3 describes moving data from the cache buffer to the S-block buffer.

Algorithm V.1: write_block
```
Input: LBA
$i$ = section(LBA)
if is_full(C_buff$_i$) then
    cache_buffer_defrag(C_buff$_i$)
end
Add(LBA,C_buff$_i$)
```

For a write operation shown in Algorithm V.1, if C_buff$_i$, the cache buffer of the section, is full (due to *head-tail* proximity), a cache_buffer_defrag operation (specified in Algorithm V.2) is invoked before adding the block to the cache buffer. During defrag of the cache buffer in Algorithm V.2, if there are no invalid blocks in the cache buffer, a group destage is first run on a chosen S-block. When there are

Algorithm V.2: cache_buffer_defrag
```
Input: buff
if num_invalid(buff)==0 then
    S_Blk = choose_S_block_to_destage()
    group_destage(S_Blk)
end
while is_valid(tail) do
    blk = read(tail)
    write(head,blk)
    $tail = tail + 1$
    $head = head + 1$
end
$tail = tail + 1$
```

invalid blocks, valid blocks at the tail are copied to the head until an invalid block is encountered, at which time the tail pointer is incremented without copying. The implementation of group_destage is illustrated in Algorithm V.3. The S-block

Algorithm V.3: group_destage
```
Input: S_Blk
read(S_Blk)
foreach block of S_Blk present in cache buffer do
    read(block)
    invalidate(block)
end
modify_with_cached_blocks(S_Blk)
write_S_block(S_Blk)
```

is read, as well as all blocks in the cache that belong to it. Then the S-block is modified in memory with the contents of the cached blocks and written to the S-block buffer. Writing an S-block to the S-block buffer, detailed in Algorithm V.4, is essentially the same as writing blocks to the cache buffer in Algorithm V.1. The defrag operation on the S-block buffer

Algorithm V.4: write_s_block
```
Input: S_Blk
$i$ = section(S_Blk)
if is_full(S_buff$_i$) then
    S_block_buffer_defrag(S_buff$_i$)
end
Add(S_Blk,S_buff$_i$)
```

in Algorithm V.5 is similar to the defrag of the cache buffer in Algorithm V.2, only without the need to check for the existence of invalid S-blocks (All S-blocks natively reside in the S-block buffer so the written S-block automatically invalidates an existing S-block). Note that all the units in Algorithm V.5 are S-blocks, hence $tail + 1$ points to the next S-block (and not next block) in the clockwise direction.

Finally, read operations are served by Algorithm V.6, which is very similar to the read operation in Algorithm IV.3 of the previous section.

Algorithm V.5: S_block_buffer_defrag

```
Input: S_buff
while is_valid(tail) do
    S_blk = read(tail)
    write(head,S_blk)
    tail = tail + 1
    head = head + 1
end
tail = tail + 1
```

Algorithm V.6: read_block

```
Input: LBA
$i$ = section(LBA)
PBA_c = cache_lookup(LBA,C_buff$_i$)
if PBA_c $\neq$ NOT_IN_CACHE then
    read(PBA_c)
end
else
    read_from_S_block(LBA)
end
```

The inclusion of detailed pseudo-code above clearly helps elucidating the operation principles of the S-block architecture's indirection system. However, to obtain a working system, which serves real read/write commands, there is a need to fill some important implementation decisions that are missing from the provided pseudo-code. Some of the decisions will have significant impact on performance, and are the subject of the next sub-section.

C. Destage and defrag policies

Given an outstanding write request and a state of the cache and S-block buffers, the indirection system needs to decide what actions to take to satisfy the write request. If the cache buffer is too full to contain the request, then defrag, and potentially destage operations are needed. In particular, the following questions should be decided to optimize access time.

1) How many S-blocks to destage before cache-buffer defrag?
2) How to choose the S-block for group destage?
3) How many invalid blocks/S-blocks to reclaim during buffer defrag?

We now attempt to answer the questions above, or at least reason on how to answer them.

1) A destage will not be necessary at all if there are sufficient used-invalid blocks in the cache buffer. However, destaging more than the required minimum may invalidate blocks close to the tail of the cache buffer, thereby reducing the overall write time.
2) Some options: the one with the most blocks in the cache buffer (best amortization of S-block write over invalidated blocks), one that contains blocks close to the tail of the cache buffer (most efficient cache buffer defrag), one that is close to the tail of the S-block buffer (most efficient S-block buffer defrag). A combination of these criteria can also be sought for optimal write time.
3) Unlike question 1 above, here there is little motivation to continue the defrag beyond the required minimum. In fact, if we postpone defrags as much as possible, more blocks may be invalidated in the meantime, and the defrag will be more efficient.

D. Simulation results

The S-blocks indirection architecture shares some of the performance behaviors of the disk-cache architecture of section IV. Both store block writes in temporary (cache) locations, which results in a varying performance that depends on the instantaneous state of the caches. Splitting the data blocks between cache and native locations also implies some degradation in sequential-read performance (consecutive LBAs no longer map to consecutive PBAs), as was illustrated in Figure 9. For concern to the presentation efficiency, we focus in this sub-section on results that demonstrate the unique behavior of the S-blocks architecture. In particular, it is shown that the S-blocks architecture solves the cardinal performance issue of the disk-cache architecture, and achieves good sustained random-write throughput.

The sample parameters with which the experimental results were obtained are now provided. The size of an S-block was set to 2000 blocks. The total amount of over-provisioned storage is 2% of the physical storage space, divided to 1% for cache buffers and 1% for extra S-blocks in S-block buffers. The 4 KB random-write throughput was measured for a workload duration of 1000[sec], in intervals of 1[sec]. The measured IOPS values are shown in Figure 12. Starting with

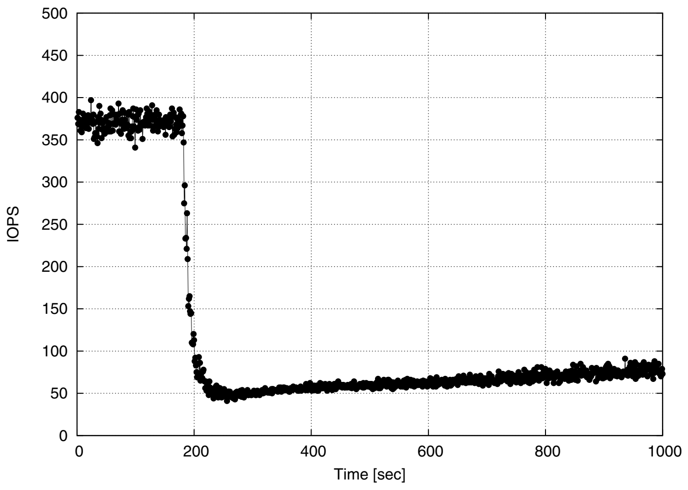

Fig. 12. Random write 4 KB IOPS in 1[sec] intervals.

empty cache buffers, the initial performance is high and flat. Then, when cache buffers fill up, a drop occurs – but unlike before – the lower performance still serves around 50×4 KB requests per second, with no 1[sec] intervals with 0 IOPS. From the drop point on, the performance slowly improves, until it reaches a steady average of around 80 IOPS. So at all times during the continuous workload, the drive is able to respond to write requests at a reasonable rate. The slow convergence to the steady value attests to the time it takes to accumulate enough invalid blocks in the cache buffer to allow for more efficient cache-buffer defrags. When garbage collection (defrags) starts, the invalid blocks in the cache buffer are on average half-way between head and tail, and thus require relatively long defrag time. When garbage collection progresses, the probability of finding invalid blocks near the tail increases, and the defrag is faster. A quicker convergence can be obtained by destaging more than the necessary number of S-blocks, at a price of a deeper temporary drop.

For the final set of results, we wish to explore how the choice of S-block for destage affects the random-write performance. For this purpose, we take two of the suggested policies in sub-section V-C, and compare them in terms of their command-completion-time histograms. The first policy, also used for the results of Figure 12, is to choose the S-block with cached blocks that is closest to the tail of the S-block buffer. The second policy is to choose the S-block with the largest number of cached blocks. For a random workload, we logged the time to complete each 4 KB write command, and plotted the histogram. We repeated the test two times, once for each of the S-block selection policies. The results for the closest-to-tail policy are found in Figure 13. The results for the largest-number-of-cached-blocks policy are found in Figure 14. Note that the time axis used in the figures is not linear, but rather consists of four different time scales, separated by grid lines: milli-seconds, hundredths, tenths, and full seconds. Comparing the results reveals a much more favorable behavior of the closest-to-tail policy, since the longest it took to serve a 4 KB command is under 50[msec]. On the other hand, the largest-number-of-cached-blocks policy exhibits a very long tail, showing completion times of more than 7 full seconds. The reason for the long tail is that if the S-block with the largest number of cached blocks is close to the head of the S-block buffer, then the defrag of the S-block buffer will require a significant amount of time. Although for random workloads the closest-to-tail policy is a clear winner, for different, less balanced workloads, considering the number of cached blocks in the S-block choice may improve performance.

E. Discussion

The S-blocks architecture proposed and studied in this section offers a substantial improvement over the more naïve disk-cache architecture of the previous section. This is achieved with a moderate amount of additional complexity. Anther advantage of the S-blocks architecture is that it allows bypassing the cache buffer altogether for sequential writes, by writing large (but not necessarily very large) chunks of data directly as S-blocks. Moreover, since the S-blocks architecture does not update data "in place", but always appends to the buffer head, terminating writes prematurely does not cause a data consistency problem. It is clear that the optimization avenues explored here are only a tiny part of the complete optimization space. More studies are needed to refine the architecture and capture its best operation modes for different workload types.

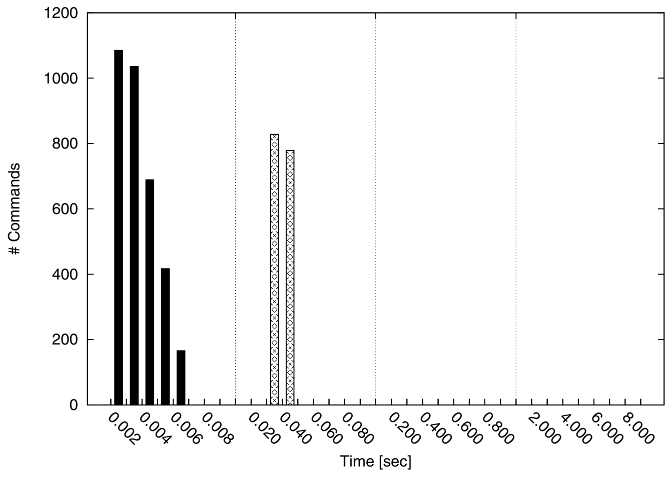

Fig. 13. Command completion time histogram for closest-to-tail destage policy.

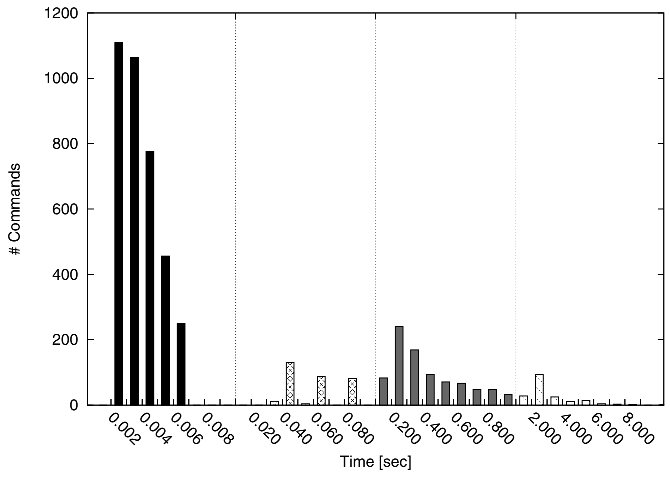

Fig. 14. Command completion time histogram for largest-number-of-cached-blocks destage policy.

References

[1] D. Thompson and J. Best, "The future of magnetic data storage technology," *IBM Journal of Research and Development*, vol. 44, no. 3, pp. 311–322, May 2000.

[2] E. Dobisz, Z. Bandic, T. Wu, and T. Albrecht, "Patterned media: nanofabrication challenges of future disk drives," *Proceedings of the IEEE: Advances in Magnetic Data Storage Technologies*, vol. 96, no. 11, pp. 1836–1846, 2008.

[3] M. Kryder, E. Gage, T. McDaniel, W. Challener, R. Rottmayer, J. Ganping, H. Yiao-Tee, and M. Erden, "Heat assisted magnetic recording," *Proceedings of the IEEE: Advances in Magnetic Data Storage Technologies*, vol. 96, no. 11, pp. 1810–1835, 2008.

[4] R. Wood, M. Williams, A. Kavcic, and J. Miles, "The feasibility of magnetic recording at 10 terabits per square inch on conventional media," *IEEE Transactions on Magnetics*, vol. 45, no. 2, pp. 917–923, Feb. 2009.

[5] G. Gibson and M. Polte, "Directions for shingled-write and two-dimensional magnetic recording system architectures: synergies with solid-state disks," Carnegie Mellon University Parallel Data Lab Technical Report, Tech. Rep. CMU-PDL-09-104, 2009.

[6] E. Gal and S. Toledo, "Mapping structures for flash memories: techniques and open problems," in *Proc. of the IEEE International Conference on Software, Science, Technology and Engineering*, 2005, pp. 83–92.

[7] B. Jacob, S. Ng, and D. Wang, *Memory systems. Cache, DRAM, Disk*. Burlington, MA, USA: Morgan Kaufmann Publishers, 2008.

[8] M. Rosenblum and J. Ousterhout, "The design and implementation of a log-structured file system," *ACM Transactions on Computer Systems*, vol. 10, no. 1, 1992.

[9] A. Traeger, N. Joukov, C. Wright, and E. Zadok, "A nine year study of file system and storage benchmarking," *ACM Transactions on Storage (TOS)*, vol. 4, no. 2, pp. 25–80, May 2008.

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

[11] Y. Hu and Q. Yang, "DCD—Disk Caching Disk: a new approach for boosting I/O performance," in *The 23rd Annual International Symposium on Computer Architecture*, Philadelphia, PA, May 1996.

[12] D. Hall, "Write-twice method of fail-safe write caching," *United States Patent 6,378,037*, 2002.