SOFTWARE DEVELOPER'S INTRODUCTION TO THE HGST ULTRASTAR ARCHIVE HA10 SMR DRIVES

Ryo Asai and Andrey Vladimirov

Colfax International

July 30, 2015

Abstract

In this paper we will discuss the new HGST Shingled Magnetic Recording (SMR) drives, Ultrastar Archive Ha10, which offers storage capacities of 10 TB and beyond. With their high-density storage capacities, these drives are well suited for large "active archive" applications. In an active archive application, the data is frequently read but seldom modified.

The SMR drives are host managed, meaning that the developer must manage the data storage on the drives. In this publication we introduce an open source library, libzbc, which was developed by the HGST team to assist developers who use SMR drives. The discussions cover topics from the very basics like opening a device, to more advanced topics like data padding. The goal of this paper is to give readers the necessary knowledge and tools to develop applications with libzbc.

We will present an example, and then report several benchmarks of I/O operations on the HGST SMR drives, and discuss the SMR drive's effectiveness as a active archive solution.

Table of Contents

1. INTRODUCTION

Big data is becoming increasingly important in a variety of areas. From astrophysical experiments producing large amounts of scientific data, to companies storing information about all of its customers, data storage requirements are increasing. This is especially the case for a group of applications called "active archive" applications. In active archive applications, vast amounts of data is dumped into storage devices, and this data is frequently read but seldom modified. With this need for ever larger quantities of data for active archive applications, comes a problem: how to cost-efficiently store these massive quantities of data? In theory, it is possible to store more data by simply adding more nodes to a data storage cluster. However, this low storage density approach becomes unfeasible, especially for active archive applications, as the cost of maintenance per byte increases rapidly with expanding data centers. Therefore, there is a need for a high storage density solution to accommodate the growing storage demands of modern data centers.

In order to fulfill this need by the active archive applications, hard drive capacity has made strides in the recent decade to increase the storage density. To push the storage density to the next level, a new technology called Shingled Magnetic Recording (SMR) has been developed. SMR allows for up to 2-3 trillion bits per square inch [2], and by utilizing this technology, HGST produced its first generation of new SMR drives, the Ultrastar Archive Ha10 [3], which offers a massive storage capacity of 10 TB in the standard 3.5 inch form factor. This is only the beginning in the SMR roadmap, thus data storage solutions using the SMR drives will likely become increasingly more common in the domain of active archive applications.

The SMR technology increases the storage density of drives by overlapping data tracks, in a way that resembles the shingles on a roof (hence the name Shingled Magnetic Recording). As a consequence of the overlapping data tracks, SMR drives are append-only. Therefore, applications with "write once, read many times" policy, such as active archive applications, are a natural fit for the SMR drives.

In using the Ultrastar Archive Ha10 drives, one important thing to note is that they only support what is called the "host managed" mode. Generally speaking, hard drive management can be divided into three groups of varying developer involvement;

- Drive managed - All management is done by the drives. The low-level interactions with the drive is hidden from the developer.
- Host managed - The developer manages the drives. If the commands are "incorrect", the drives are not able to handle it.
- Host aware - The developer manages the drives, however the drives are capable of handling some "incorrect" commands at the cost of unpredictable performance.

The drive managed mode will have the least developer input (high-level abstraction), and the host managed mode will have the most (low-level abstraction). Again, in the case of the HGST SMR drives, only the "host managed" mode is supported. Thus, the developer must do the work of managing the data on the drives. However, with the "host managed" mode, the developer has much finer grained control over the performance and power consumption of the SMR drives.

In order to interact directly with the SMR drives, one must use Zoned Block Commands (ZBC) for SAS drives and Zoned ATA Commands (ZAC) for SATA drives. But these commands can be too low-level of an abstraction to easily develop an application with. To assist developers for the drives, the HGST team abstracted the low level communication with ZBC into a open source library, libzbc [4], which can be used to utilize the HGST SMR drives in the "host managed" mode. The primary purpose of this paper is to introduce libzbc through a simple code example, explaining in detail how to use the library as well as constraints to look out for. After the introduction of libzbc 2.0.0, we will present several bandwidth benchmarks of the Ultrastar Archive Ha10 drives (HGST HMH7210A0ALE600) using libzbc in a variety of data management scenarios.

2. DRIVE ORGANIZATION: ZONES

In order to properly discuss libzbc, it is important to introduce some details about the hardware organization of the HGST Ultrastar Archive Ha10 drives. The SAS drives are called Zoned Block Devices (ZBD), and the SATA drives are called ZAC-enabled devices. With libzbc, the drives are organized in regions called "zones", and most libzbc operations will revolve around controlling these zones.

In the Ultrastar Archive Ha10 drives, there are total of 37256 zones (enumerated $\{0, 37255\}$ in libzbc) each with a capacity of 256 MiB. Thus, in total these drives provide $37256 * 2^{28} \approx 10$ TB of storage. Each zone has an unique range of Logical Block Address (LBA) values associated with it, which describes the location in the drive where the data resides. For the drive that is used in this paper (512e model) LBA has a size of 512 Bytes, and each 512 B block on the drive has a unique LBA value associated with it. Note however that other models of the drives may have a different size for LBA size (e.g. 4Kn model may have 4 KB sectors). Determining the LBA size for the drive is discussed in Section 3.1.

Of the 37256 zones, the first 378 zones ($\{0,377\}$) are special zones called "Conventional Zones". These zones behave like standard hard drives, and allow writes to any location. The remainder of the zones ($\{378, 37255\}$) are called "Sequential Write Required Zones". In this paper we will limit our discussion to the sequential zones, because this is the non-conventional aspect of the SMR technology that gives access to the huge capacity of these drives.

As the name suggests, these zones can only be written into sequentially. In other words, data is *append-only* in a sequential zone, and the only way to reclaim space in a sequential zone is to clear the whole zone. This is because the sequential zones are managed using "write pointers" ("wp" for short), which contains the LBA value of the next logical block that can be written in the zone. Each sequential zone has its own write pointer, and these write pointers are stored in special regions in the drive itself. When data with a size equivalent to n logical blocks (n*512 B) is written into a sequential zone, the write pointer advances by n. The sequential zones are append only because the write pointers can change in only two ways; advance or reset (see Section 3.3).

Figure 1 shows a schematic diagram of the zone structure.

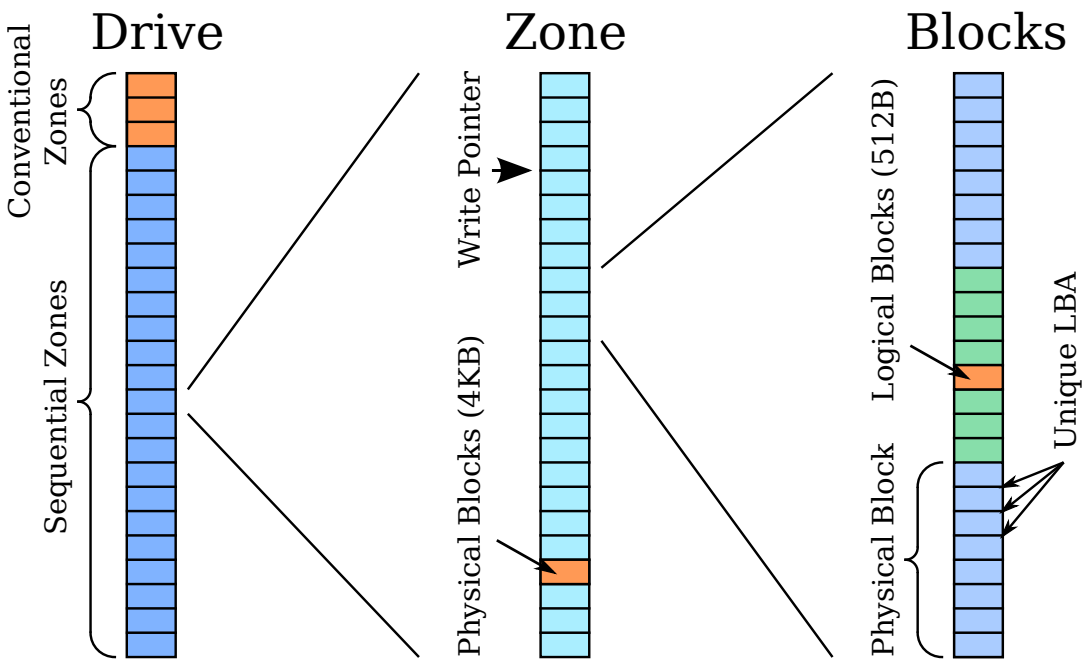

Figure 1: Zone organization of the HGST SMR drives.

Using the SMR drives with libzbc will necessarily revolve around using sequential zones. In fact, most operations in libzbc will interact with a specific zone. Therefore, it is important to have a basic understanding of how the zones function and are organized in order to take advantage of the SMR drives and libzbc.

3. USING LIBZBC

In this section we will discuss how to use libzbc to develop a basic application that utilizes the SMR drives. The general flow of using libzbc is:

- Open the device, then query information.
- Compile a list of zones.
- Write to and read from the drive.
- Close the device.

Listing 1 shows a simple application that writes some data into the SMR drives, then reads it back. In the rest of the section, we will be discussing each component of this application in depth.

```c
#include <stdio.h>
#include <stdlib.h>

#include <mm_malloc.h>
#include <libzbc/zbc.h>

// User function to initialize data
void PopulateData(void* write_buffer, int data_size);

// User function to verify read data
void VerifyData(void* read_buffer, int data_size);

int main(int argc, char* argv[]) {
  struct zbc_device_info info;
  struct zbc_device *dev = NULL;
  struct zbc_zone *zones = NULL;
  unsigned int nr_zones;

  /* Opening and getting Information (see Section 3.1) */
  char *path = argv[1];
  unsigned int ret;
  ret = zbc_open(path, O_RDWR, &dev);
  ret = zbc_get_device_info(dev, &info);

  /* Allocating I/O buffers, and populating the data to be written. (see Section 3.3.1) */
  const int data_size = 1<<20; // writing 1MB then reading 1MB
  void* write_buffer = (void*) _mm_malloc(data_size, 4096);
  void* read_buffer = (void*) _mm_malloc(data_size, 4096);
  PopulateData(write_buffer, data_size);  // Populate the data to be written

  /* Listing Zones and checking if there is enough space (see Section 3.2) */
  zbc_list_zones(dev, 0, ZBC_RO_ALL, &zones, &nr_zones);
  const int zone_id = atoi(argv[2]);
  const int rem_space_lba = zbc_zone_next_lba(&zones[zone_id]) - zbc_zone_wp_lba(&zones[zone_id]);
  const int rem_space_bytes = rem_space_lba * info.zbd_logical_block_size;
  if(rem_space_bytes < data_size || zbc_zone_full(&zones[zone_id]))
    { fprintf(stderr, "Not enough space in zone %d. Exiting\n", zone_id); exit(1); }
  printf("Space remaining in zone %d; %dB\n", zone_id, rem_space_bytes);

  /* Writing the data (see Section 3.3.2) */
  int written;
  const int lba_count = data_size / info.zbd_logical_block_size;
  written = zbc_write(dev, &zones[zone_id],(void*) write_buffer, lba_count);
  if(written != lba_count)
    { fprintf(stderr, "Write Failed. Written = %d. Exiting\n", written); exit(2); }

  /* Re-reading the data (see Section 3.3.2) */
  int read;
  const int lba_offset =
          zbc_zone_wp_lba(&zones[zone_id]) - zbc_zone_start_lba(&zones[zone_id]) - lba_count;
  read = zbc_pread(dev, &zones[zone_id],(void*) read_buffer, lba_count, lba_offset);
  if(read != lba_count)
    { fprintf(stderr, "Read Failed. Read = %d. Exiting\n", read); exit(3); }

  VerifyData(read_buffer, data_size);

  /* Closing the drive (see Section 3.1) */
  ret = zbc_close(dev);
}
```

Listing 1: A simple example application for using the HGST SMR drive. The section numbers in parentheses refers to the section where the topic is discussed. The full code will be available for download at []

3.1. Opening and Querying the Drive

In this section, we will discuss how to open the SMR drive and get information about it. Listing 1 demonstrates a basic libzbc application, which we will use in the discussion.

The first step to using the HGST SMR drive is to open it using the `zbc_open()` function, which is called in line 22 in our example.

```
1  int zbc_open(const char* filepath, int flags, struct zbc_device_t **dev);
```

Here, `filepath` is a `char` array which contains the path to the SMR drive. In our application, this path is passed in as the first command line argument. In the case of our system, this was `/dev/sg1`. The `flags` argument determines how the drive will be used. In our code example, we used `O_RDWR` to set it to read-write mode. `O_RDONLY` will set it to read-only mode, and `O_WRONLY` will set it to write-only mode. Finally, the last argument `dev` is a pointer to the memory address of a `zbc_device` structure (see line 15). After opening a device, `dev` will act as the device handle to be used in all subsequent operations on the drive.

To get various information about the open drive, use the `zbc_get_device_info()` function (line 23).

```
1  int zbc_get_device_info(struct zbc_device_t *dev, struct zbc_device_info_t *info);
```

The first argument is the device handle of the drive. Make certain that this handle was initialized with `zbc_open()`. The second argument, pointer to a `zbc_device_info` structure, will be populated with variety of information such as the device model or the logical block size. For the full listing of the available information, refer to *include/libzbc/zbc.h* located inside installation directory. In the case of our example we use `info` to determine the logical block size of the SMR drives.

After all the operations with the drive are done, close the device using the `zbc_close()` function (line 57).

```
1  int zbc_close(struct zbc_device_t *dev);
```

The argument for the `zbc_close()` is simply the device handle of the drive to be closed.

Finally, note that each function returns 0 if it was successful. We did not implement a check for completion in Listing 1 to keep the example simple, but it is generally a good practice to have the check and exit if necessary.

3.2. ZONE CONTROL

In this section, we will introduce some utility functions for working with zones. Again, we will be using Listing 1 as the example application.

To get the list of the zones, use the `zbc_list_zones()` function.

```
zbc_list_zones(struct zbc_device *dev, uint64_t start_lba, enum zbc_reporting_options ro,
               struct zbc_zone **pzones, unsigned int *nr_zones)
```

`dev` is the drive to from which the list of zones is compiled. The second argument, `start_lba` is the LBA from which the zones are listed. In our example, we simply queried the whole drive by passing in 0. Third argument, `ro`, acts as a filter for the zones. For example, it can be set to `ZBC_RO_EMPTY` to only list the zones that are empty. In our example, we simply set it to `ZBC_RO_ALL` to list all zones. For the full list of options, refer to *include/libzbc/zbc.h*. The fourth argument is the pointer for the array of `zbc_zone` structures. `zbc_list_zones()` uses an internal `malloc` to allocate the list of zones, which is then populated and `pzone` is set to this newly allocated array. This array, for example, will be used in reading and writing into zones (Section 3). The final argument, `nr_zones`, will be set to the total number of zones populated by list zones.

Again, note that `zbc_list_zones()` uses an internal `malloc` for the list of zones. If you call this function multiple times in one application, you will need to use `free()` to deallocate `zones` to avoid memory leaks. Alternatively the list of the zones can be updated without allocation using the `zbc_report_zones()` function.

```
zbc_report_zones(struct zbc_device *dev, uint64_t start_lba, enum zbc_reporting_options ro,
               struct zbc_zone *zones, unsigned int *nr_zones)
```

The argument list for this function is identical to `zbc_list_zones` except for the fourth argument, which is the array of `zbc_zones` structures to be updated.

All of the information necessary for working with zones can be gained by accessing the members of the `zbc_zone` structure. libzbc has a variety of utility accessor macros defined for this purpose. A selection of these are shown in Listing 2.

```
zbc_zone_sequential(struct zbc_zone *zone) // Returns true if the zone is sequential
zbc_zone_empty(struct zbc_zone *zone)      // Returns true if the zone is empty
zbc_zone_full(struct zbc_zone *zone)       // Returns true if the zone is full
zbc_zone_start_lba(struct zbc_zone *zone)  // Returns the starting LBA of the zone
zbc_zone_length(struct zbc_zone *zone)     // Returns the total size of the zone in units of LBA
zbc_zone_wp_lba(struct zbc_zone *zone)     // Returns the LBA of the write pointer
zbc_zone_next_lba(struct zbc_zone *zone)   // Returns the LBA of the next zone
```

Listing 2: Utility accessor macros for zones.

In Listing 1, we use these macros together with `zbc_list_zones` to determine how much space is available in the zone to write into (see lines 34 to 37). Warning: there one caveat to this technique, which is that if the zone is full, the write pointer will be at the *start* of the zone. Thus, to implement the check for availability of space, we need to add an additional check to see if the zone is full (see line 36).

3.3. I/O

In this section, we will discuss how to do read/write operations on sequential zones in an HGST SMR drive using libzbc. There are two stages to I/O operation with HGST SMR drives; data buffer preparation and read/write.

3.3.1. Data Buffer Preparation

For write operations with HGST SMR drives, the size of data must be a multiple of 4 KiB. This is because the smallest block of data that can be written is equal to the HGST SMR drive's physical block size, which is 4096 bytes. For read operations, the smallest size is the logical block size, which is 512 B. Therefore if the data is not a multiple of 4096 or 512 bytes, then a technique known as padding is required.

With padding, extra elements are added to the end of a data buffer in order to ensure that its size is of a multiple of 4096 or 512 bytes for writes and reads, respectively. Listing 3 demonstrates preparing a memory buffer for a write operation.

```
1  const int phys_blk_size = 4096/sizeof(DATA_TYPE);
2
3  // n is the number of elements. Computing the padded n.
4  n += (phys_blk_size-1) - ((n-1) % phys_blk_size);
5
6  // Allocating aligned data with mm_malloc
7  DATA_TYPE* data = (DATA_TYPE*) malloc(n*sizeof(DATA_TYPE));
```

Listing 3: Padding the data buffer to for the write operation.

The macro `DATA_TYPE` is the type of the data (e.g. int, char, double, etc.). Recall that the requirement is in bytes, not elements. So we must divide 4096 by the size of `DATA_TYPE` in order to compute the value that the number of elements must be a multiple of.

The memory buffer, `data`, in Listing 3 is a multiple of 4 KiB. Therefore it can now be used for I/O withe the HGST SMR drives. In our example in Listing 1, to keep it simple we hard-coded the size to be a multiple of 4096 B.

3.3.2. Read/Write

In this section we will discuss how to write to and read from the HGST SMR drives. Note that the drive must be open (see Section 3.1) and you must have the list of the zones (see Section 3.2) before proceeding. Again we will be using Listing 1 to demonstrate the usage of the libzbc functions.

To write into a sequential zone, use `zbc_write()`.

```
int zbc_write(struct zbc_device *dev, struct zbc_zone *zone,
              const void *buf, uint32_t lba_count);
```

The first argument, `dev`, is the device handle of the drive to write into. The next argument, `zone`, is the `zbc_zone` structure of the zone to write into. Again, refer to Section 3.2 for more information. The third argument, `buf`, is the memory buffer containing the data that will be written. Be certain that the buffer was properly prepared (see Section 3.3.1). And the final argument, `lba_count`, is the amount of data to be written in units of LBA, which is 512 B.

Recall that location to write to on the drive is determined by the write pointer; `zbc_write()` will simply write the data into the blocks following the write pointer, and advance the write pointer appropriately. If not enough space is available, the `zbc_write()` will fail at run-time with "Input/Output" error (error code -5). Thus with `zbc_write()`, it is important to ensure that there is enough available space in the zone. To determine the available space in a zone, query the write pointer address and compare it with the start address of the next zone (see Listing 1). One exception to this is when the zone is already full. In this case, there will not be a run-time error, and the data is simply not written. To be certain that the write operation completed correctly, use the return value of `zbc_write()`, which is the number of logical blocks that were written. If the returned result is equal to `lba_count`, then it completed properly.

There are two limitations to the sizes of the data that can be written. First, the write size must be a multiple of 4096 bytes, or 8 logical blocks (e.g $\texttt{lba\_count}\%8 = 0$). If this condition is not met, `zbc_write` will fail at run-time with "Input/Output" error (-5). If the size of the data is not a multiple of 4096 B, padding will need to be implemented (see 3.3.1). The second limitation is that there is a cap on the largest data size that can be written at once. For our particular system (see Section 4.1) this limit is 5.25 MiB. This is not a limitation of the drives or libzbc, but of the "scatter gather list" which is managed by the Linux operating system. If the data is larger than 5.25 MiB, then it needs to be partitioned and written using multiple `zbc_write()` operations. When `zbc_write()` tries to write more than 5.25 MiB, it will fail with "Cannot allocate memory" error (error code 12).

To read from a sequential zone, use `zbc_pread()`.

```
int zbc_pread(zbc_device_t *dev, zbc_zone_t *zone, void *buf,
              uint32_t lba_count, uint64_t lba_ofst)
```

The first argument, `dev`, is the device handle of the drive to read from. The next argument, `zone`, is the `zbc_zone` structure of the zone to read from. The third argument, `buf`, is the memory buffer to which data will be placed. Just as the write case, be certain that the buffer was properly prepared (see Section 3.3.1). The fourth argument, `lba_count`, is the amount of data to be read in units of LBA, which is 512 B. And the final argument, `lba_ofst`, determines where the data is read from in terms of LBA offset from the beginning of the zone.

Unlike `zbc_write()`, with `zbc_pread()` you are not limited to one zone; in other words, if the end of the zone is reached, the `zbc_read()` will continue to the next zone. Furthermore, the `zbc_pread()` can read from any LBA address, regardless of the write pointer location, by specifying the `lba_offset`.

However, this also means that the developer is not protected from accidentally reading an uninitialized data region. Thus, it is important to be certain that the `zbc_pread()` is reading the correct range of logical blocks. The output of the function, similarly to `zbc_write()`, is the number of logical blocks read. Again, it is recommended to implement a check to make certain the correct number of logical blocks were read.

With `zbc_pread()`, the granularity of read size is 1 logical block (512 B) because `lba_count` is an integer. Thus, make certain that the memory buffer to which data is read to is a multiple of 512 B (see Section 3.3.1). Just as `zbc_write()`, there is a limit on the largest data size that can be read at once. Again, for our system the limit is 5.25 MiB. To read more than 5.25 MiB, multiple calls to `zbc_pread()` are needed. When `zbc_pread()` tries to read more than 5.25 MiB, it will fail with "Cannot allocate memory" error (error code 12).

In order to reclaim space on the SMR drives, you must reset the write pointer of a zone. This effectively reclaims the space from the whole zone; i.e. the granularity of data reclamation on the SMR drives is individual zones. To reset a write pointer of a zone, use `zbc_reset_write_pointer()`.

```
1 int zbc_reset_write_pointer(zbc_device_t *dev, uint64_t start_lba);
```

The first argument, `dev`, is the device handle of the drive to write into. And the second argument, `start_lba` is the starting LBA of the zone to reset. To reset the whole drive, use `start_lba`$= -1$.

Note that the the programmer must manually update `zbc_zone_list` using `zbc_report_zones()` (see Section 3.2) before being able to write into the reclaimed zone. Therefore, if the application will have another write into the same zone, be certain to update the zone information first.

Although `zbc_reset_write_pointer()` does not delete the data but only resets the write pointer, the data will not be recoverable with `zbc_pread()` once the reset pointer was called. Calling `zbc_read()` to read the same LBA location will succeed, but the read value will be different.

Finally, a few notes on caching. In order to expedite small writes, many modern hard drives have temporary storage locations for the writes, known as "caches". There are typically two types of caches; the drive cache managed by the drive, and the host cache managed by the operating system. The HGST HGST SMR drives is cached by the drive cache (256 MiB), but is *not* cached by the operating system. The cache gets flushed automatically as it fills up, but it is possible to manually flush the cache by using `zbc_flush()`.

```
1 int zbc_flush(zbc_device_t *dev);
```

In our benchmarks, we have found that during continuous writes, the drive cache seems to flush at approximately 40 MiB occupied.

4. BENCHMARKING WITH LIBZBC

We developed several simple benchmark applications using libzbc, then benchmarked the HGST Ultrastar Archive Ha10 SMR drives (see Section 4.1) with them. Six different cases were benchmarked.

- Large contiguous write.
  This operation represents a workload where one is writing a large chunk of data, large enough to not fit in the drive cache. To simulate this type of workload, we will write 2 GiB (8 zones) of data sequentially with variety of values for the individual write sizes.

- Small contiguous writes.
  This operation represents a workload where one is infrequently writing small chunks of data multiple times. In other words, this is the performance of writing into the drive cache, but not necessarily writing into the drive itself. To simulate this, we will write 16 MiB (small enough to fit in cache) sequentially with variety of sizes.

- Sequential cold reading.
  This operation represents a workload where one is contiguously reading a large chunk of data. To simulate this, we will read 2 GiB (4 zones) worth of contiguous sequential zones.

- Random cold read.
  This operation represents a workload where one is reading multiple small chunks of data that is scattered in memory. To simulate this, we will read randomly from 2 GiB (8 zones) memory region with a range of read sizes.

- Sequential re-read.
  This operation represents a workload where one is reading contiguously in a small memory region multiple times. To simulate this, we will read from the a contiguous 32 MiB data regions multiple times.

- Random re-read.
  This operation represents a workload where one is randomly reading small chunks from a small region of memory. To simulate this, we will read randomly from a 32 MiB data region with a range of read sizes.

For timing, we used internal timers that were placed only around the I/O operations (read, write, flush) in order to gauge the performance of just the I/O operations. The full source codes for benchmarking scripts are available for download at the Colfax Research website[1].

4.1. SYSTEM CONFIGURATION

The system used was single socket Intel Xeon E5-1660 CPU (6 cores at 3.30GHz) running CentOS 7, kernel build 3.19. Model of the Ultrastar Archive Ha10 drives was HGST HMH7210A0ALE600 (obtained with `hdparm`), and we used libzbc 2.0.0 to write our benchmark applications. To compile the benchmark applications, we used gcc version 4.8.3.

4.2. Performance Results

4.2.1. Write Benchmarks

Figure 2 shows the average bandwidth and average latency of individual write operations. The bandwidth was computed by dividing the read chunk size the average latency of the reads. The IOPS (I/O per second = 1/latency) is the average value for a *single* write. For "Large Writes" we wrote 2 GiB of data contiguously with write chunk sizes ranging between 4 KiB and 4 MiB in powers of 2. With "Small Writes" we instead wrote 16 MiB of data contiguously with the same range of chunk sizes. The size chunk size of 16 MiB was determined empirically to be the largest write chunk size before we began to see the effects of flushing the cache. Furthermore, for small writes, between switching write sizes, we flushed the device and added a `sleep` for 3 seconds in our benchmarking bash script in order to ensure that the cache has enough time to flush its cache.

The maximum bandwidth for small infrequent writes was 136.4 MB/s (with 4 MiB write chunk size), which is twice as better than the maximum bandwidth of large writes, which was 62.5 MB/s (with 4 MiB write chunk size). This is the expected result because the total data size of 16 MiB is small enough to fit in the drive cache. With the Ultrastar Archive Ha10 drives, writes to the drive are re-read and verified. However this verification does not happen when the data is written to the cache, but when this data is flushed to the media. The overhead from verification also contributes to the asymmetry in bandwidth for read and write, as we will see in the next section. Some applications that use distributed storage may be able to take advantage of the infrequent write bandwidth, however, for applications that require writing a large volume of data continually, 62.5 MB/s will be the expected maximum bandwidth.

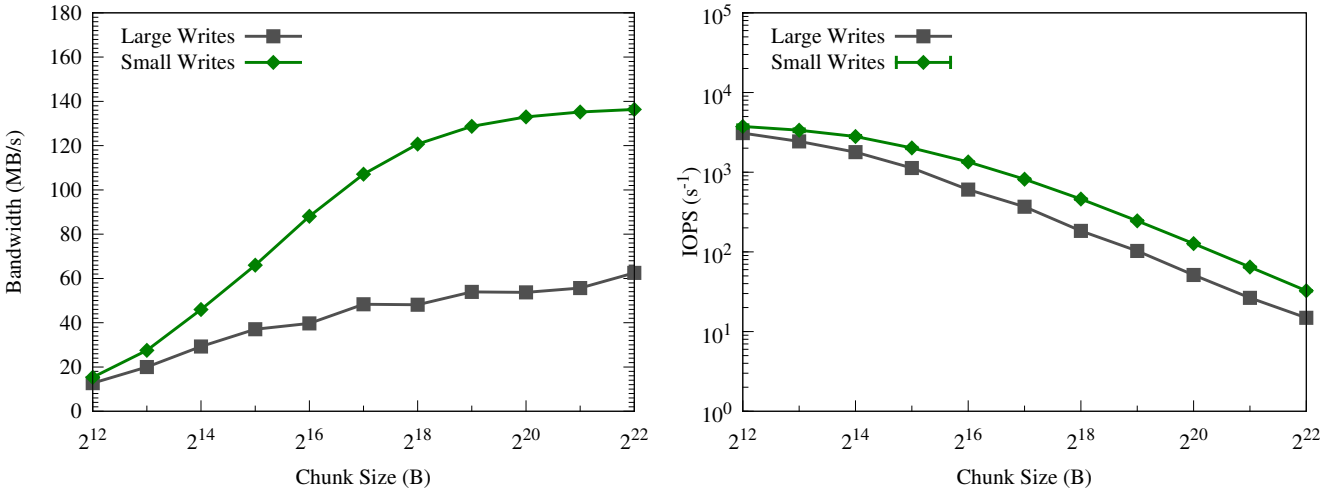

Figure 2: The average bandwidth (left) and IOPS (right) of individual write operations.

Our benchmarks show that, as expected, writing small amounts of data has better bandwidth and latency than writing large amounts of data. According to our result, the bandwidth for "large writes" tends to be less than half of the bandwidth with "small writes" at every chunk size. Therefore applications that

allow for infrequent writes, or distributed storage applications may be able to enjoy a better write latency an bandwidth than an application that dumps a large amount of data into a single drive.

Another interesting property to note is that the bandwidth improves as write chunk size increases, for both the "small writes" and "large writes". This is most likely due to the fact that there is some latency in starting to write into the drive. Therefore, if the application allows it, buffering the write data until 4 MiB is gathered will likely improve the I/O performance of the application.

4.2.2. Read Benchmarks

Figure 3 shows the bandwidth and latency of "cold random read" as a function of individual read chunk size. The dotted line in th bandwidth benchmark corresponds to the bandwidth of "cold sequential read" with 4 MiB chunks. The bandwidth was computed by dividing the read chunk size by the average latency of the reads. The IOPS is the average value for a *single* read. For "cold random read" the benchmark application read randomly from a 2 GiB region in memory that was pre-populated. The drive was flushed and closed/reopened after the write benchmark, before carrying out the read benchmark. For "cold sequential read" we only tested 4 MiB chunk sizes because this is simulating a workload where a large amount (much larger than 4 MiB) of contiguous data is needed by the application. In such case, it would not make sense to read the data in smaller chunk sizes. For cold random reads we tested the range of 4 KiB to 4 MiB in powers of 2.

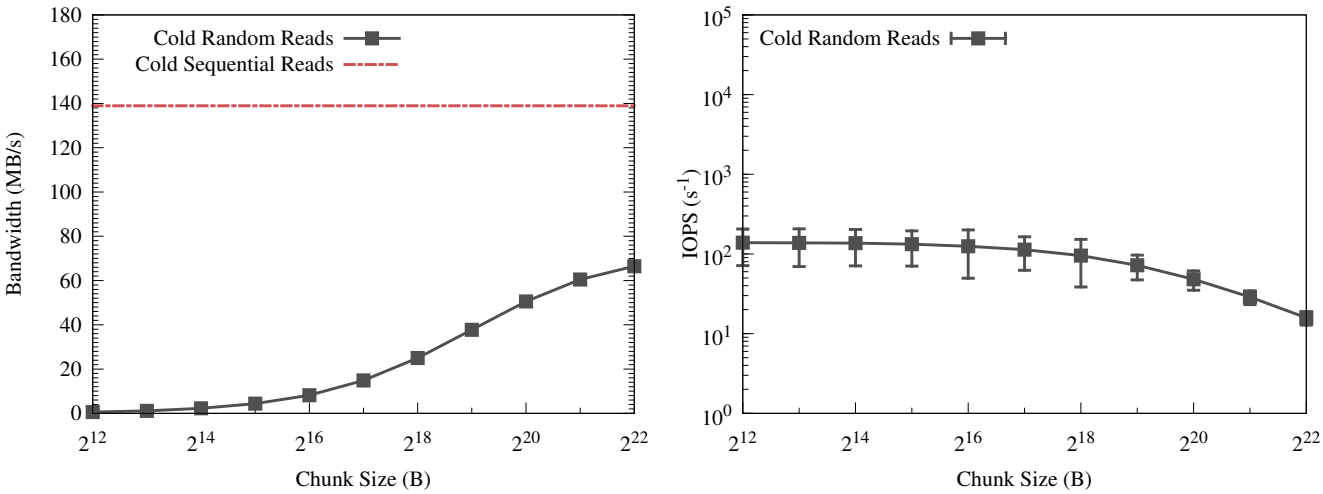

Figure 3: The average bandwidth (left) and IOPS (right) of individual read operations over a large data region (2 GiB).

According to our measurements, if the data is contiguous in the drive, the cold read bandwidth is 138.9 MB/s which is twice the bandwidth for writing the same size of data. However, if the data is not contiguous but randomly distributed, then the bandwidth drops drastically to 66.4 MB/s maximum bandwidth. This is to be expected because reading from distant locations on the drive requires the read-and-write head to shift a large physical distance, which introduces overhead. The overhead will be more significant for smaller the read chunk sizes, because this shift in the read-and-write head is more frequent.

This is reflected in our measurements, in which we see that smaller chunk sizes have lower bandwidth. In general, the more random the access pattern is, the lower the bandwidth performance. Therefore, when working with large reads, it is extremely important to be mindful of the structure of the data stored and avoid scattered drive access as much as possible.

Figure 4 shows the bandwidth and latency of "random re-read" as a function of individual read chunk size. Again, the dotted line in th bandwidth benchmark corresponds to the bandwidth of "sequential re-read" with 4 MiB chunks. Similar the cold read benchmarks, the bandwidth was computed by dividing the read chunk size the average latency of the reads and the IOPS is the average value for a *single* read. For these benchmarks, the drive accesses were limited to within a 32 MiB region that was pre-populated. Again the drive was flushed and closed/reopened after the writes before proceeding to the read benchmark. For "sequential re-read" we only tested 4 MiB chunk sizes, and for random re-read we tested the range between 4 KiB to 4 MiB.

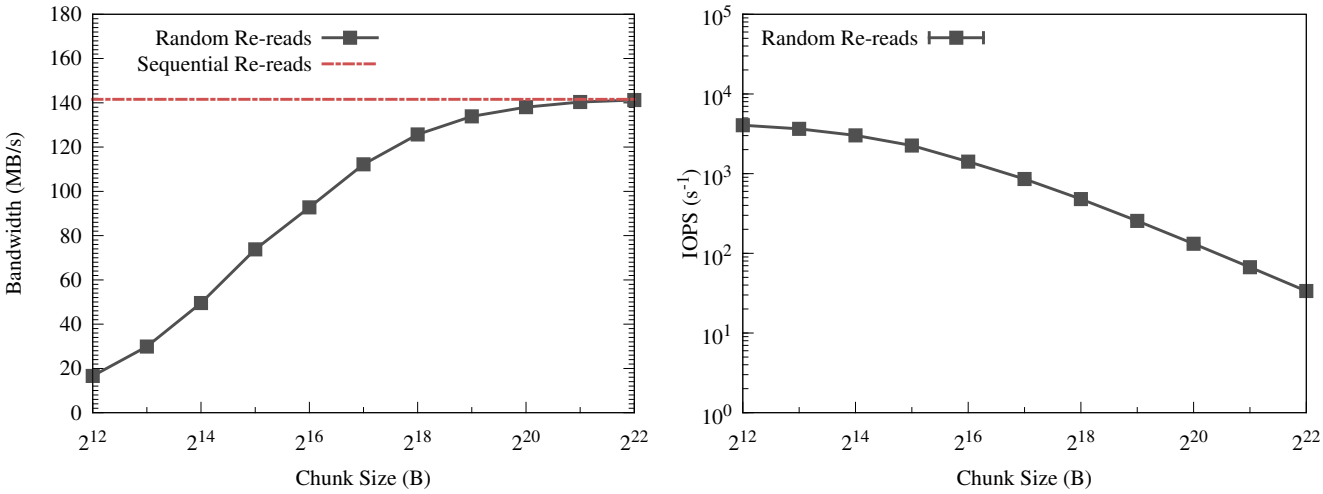

Figure 4: The average bandwidth (left) and IOPS (right) of individual read operations over a small data region (32 MiB).

According to our measurements, for small re-reads both the sequential and random access patterns was able to achieve 141.6 MB/s bandwidth when using 4 MiB chunk sizes. This is because 32 KiB is small enough of a region that it fits within the drive cache, so the data is re-read from there. Therefore, the latency of access is much smaller compared to "cold random read". However, this does not mean that data locality is not important. Random access only achieves 141.6 MB/s bandwidth when the data is in 4 MiB chunks. Reading data chunks with smaller sizes has a much lower bandwidth. Thus, with small re-reads, if contiguous data access can not be achieved, it is important to get data chunk to be as large as possible.

5. SUMMARY

In this publication we discussed the new HGST Ultrastar Archive Ha10 drives from an application developer's point of view. The SMR technology, which allows these drives to offer a massive 10 TB storage capacity, require that these drives are host managed. As a consequence, the application developer

must manage the data on the drive. To assist the developers, the HGST team created an open source library, libzbc [4] (version 2.0.0 at the time of this writing), which has been the main focus of this paper.

Using a simple code example (see Listing 1) we discussed the general work flow of using the SMR drives using libzbc. The code example (example.c) is available for download at REF. The steps demonstrated were:

- Open the device, then query information.
- Compile a list of zones.
- Write to and read from the drive.
- Close the device.

We discussed each step in detail, including the functions used, and discussed some restrictions (e.g. allowed write sizes) that a developer must be aware of. The topics ranged from very basic operations like simply opening the drive, to more advanced topics like padding. The main goal of the section was to have it serve as a general introductory reference for developers using libzbc.

Following the code examples, we presented several benchmarks of the HGST Ultrastar Archive Ha10 drives (model HMH7210A0ALE600) using a benchmarking application developed with libzbc. We first benchmarked contiguous write of 2 GiB and 16 MiB data, and saw 62.5 MB/s and 136.4 MB/s maximum bandwidth respectively. Then we benchmarked reading 2 GiB data sequentially and randomly. The maximum bandwidth for the sequential read was 138.9 MB/s, but with random access the bandwidth dropped to 66.4 MB/s Finally we benchmarked reading from a 32 MiB data region both sequentially and randomly, and found that both access patterns achieves 141.6 MB/s.

The 10 TB HGST Ultrastar Archive Ha10 drives are the first generation of high capacity SMR drives, and the SMR technology is likely to keep pushing the limits of storage density further. As the demand for storage space continue to grow, it is likely that SMR devices like HGST SMR drives will become increasingly common, especially in the domain of active archive applications.

6. Acknowledgments

The authors thank the members of the HGST team, C. Guyot, D. Jarvis, D. Le Moal, A. Manzanares, E. Wu and others, for their assistance and input in the writing of this paper.

References

[1] Landing page for the paper, and sample code download.
http://colfaxresearch.com/hgst-smr-drive-intro

[2] Shingled Recording for 2-3 Tbit/in2 by S. Greaves et al, IEEE Trans. Magnetics, Vol. 45, No.10, October 2009, 3823

[3] Ultrastar Archive Ha10 product page.
http://www.hgst.com/hard-drives/enterprise-hard-drives/enterprise-sata-drives/ultrastar-archive-ha10

[4] libzbc download.
https://github.com/hgst/libzbc