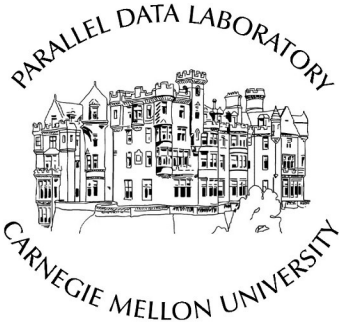

Shingled Magnetic Recording for Big Data Applications

Anand Suresh
Garth Gibson
Greg Ganger

CMU-PDL-12-105
May 2012

Parallel Data Laboratory
Carnegie Mellon University
Pittsburgh, PA 15213-3890

Acknowledgements: We would like to thank Seagate for funding this project through the Data Storage Systems Center at CMU. We also thank the members and companies of the PDL Consortium (including Actifio, American Power Conversion, EMC Corporation, Emulex, Facebook, Fusion-io, Google, Hewlett-Packard Labs, Hitachi, Huawei Technologies Co., Intel Corporation, Microsoft Research, NEC Laboratories, NetApp, Inc. Oracle Corporation, Panasas, Riverbed, Samsung Information Systems America, Seagate Technology, STEC, Inc. Symantec Corporation, VMWare, Inc. and Western Digital) for their interest, insights, feedback, and support.

Abstract

Modern Hard Disk Drives (HDDs) are fast approaching the superparamagnetic limit forcing the storage industry to look for innovative ways to transition from traditional magnetic recording to Heat-Assisted Magnetic Recording or Bit-Patterned Magnetic Recording. Shingled Magnetic Recording (SMR) is a step in this direction as it delivers high storage capacity with minimal changes to current production infrastructure. However, since it sacrifices random-write capabilities of the device, SMR cannot be used as a drop-in replacement for traditional HDDs.

We identify two techniques to implement SMR. The first involves the insertion of a shim layer between the SMR device and the host, similar to the Flash Translation Layer found in Solid-State Drives (SSDs). The second technique, which we feel is the right direction for SMR, is to push enough intelligence up into the file system to effectively mask the sequential-write nature of the underlying SMR device. We present a custom-built SMR Device Emulator and ShingledFS, a FUSE-based SMR-aware file system that operates in tandem with the SMR Device Emulator. Our evaluation studies SMR for Big Data applications and we also examine the overheads introduced by the emulation. We show that Big Data workloads can be run effectively on SMR devices with an overhead as low as 2.2% after eliminating the overheads of emulation. Finally we present insights on garbage collection mechanisms and policies that will aid future SMR research.

Keywords: Shingled Magnetic Recording, SMR, Big Data, Hadoop

1 Introduction

Moore's Law states that the speed of processors doubles every 18 months, while according to Mark Kryder, even this high rate of growth is a snail's pace compared to the rate of growth of HDD capacities [1]. Starting out in 1956 with the IBM 305 RAMAC delivering approximately 4.4 megabytes of storage using 50 platters measuring 24-inches in diameter [2], HDDs have shrunk in size by a factor of 8-10, but have grown in capacity by a factor of around 600,000 with Seagate's Constellation ES.2 ST33000650NS 3 TB Internal Hard Drive [3].

This phenomenal growth cannot be attributed to big changes in one aspect of recording technology. Rather, it is the summation of many small changes in the different components that make up a hard drive, each contributing a small percentage of the overall capacity growth; e.g. the shift from longitudinal recording to perpendicular recording, better recording materials, and steadily evolving head designs. Through these small and steady increments, the storage industry has been delivering HDDs with an annual capacity growth of 30%-50% [4] [5]. However, these historical growth rates are no longer sustainable.

1.1 The end of the line

With data-densities fast approaching the superparamagnetic limit [6], it will be impossible to reduce the size of a bit further without facing the problem of thermal self-erasure [7]. Moreover, the trend in computing is shifting towards Big Data and the demand for low-cost high-capacity storage is higher than ever.

New technologies like Heat-Assisted Magnetic Recording (HAMR) [8] and Bit-Patterned Magnetic Recording (BPMR) [9] are on the horizon, but there are a few engineering challenges which must be overcome before these technologies are commercially viable. BPMR requires advanced lithography techniques to produce bit-patterned media with the desired feature sizes while ensuring uniform internal magnetic structure, followed by techniques to ensure write synchronization. HAMR involves new recording physics and media, with the need to mount laser diodes on the write heads [10].

Once the engineering challenges have been hurdled, the cost barrier needs to be broken. New manufacturing techniques and production lines require capital investments of the order of a few billion dollars. It has been estimated that by 2012, the conversion of just 10% of the industry's storage capacity to Discrete Track Recording (DTR) and BPMR technologies will require investments of over $650,000,000 [11] for just the equipment! Thus, the storage industry is searching for alternative technologies that can not only provide high capacity, but also be manufactured on current production infrastructure.

1.2 The light at the end of the tunnel

Shingled Magnetic Recording (SMR) seems to be the best answer to the problem at hand. It is a strong contender to replace traditional magnetic media [12] [13] [14] [15] [16] [28] because of its ability to be manufactured by current manufacturing processes with minimal change [14], while still providing considerably higher storage densities. This is achieved by partially overlapping contiguous tracks, thereby squeezing more data tracks per inch. However, the overlapping of data tracks sacrifices the random writing capabilities of the device as any attempt to update data in a track will result in overwriting of data in overlapping data tracks down stream. Research suggests that the loss of random writing can be masked by modifying the system software stack [5] [12], thereby creating a storage system that augments traditional PMR to deliver higher storage capacities. Such innovation is needed, over and above HAMR and BPMR to allow continued advances in storage. However, SMR comes with its own set of challenges.

1.3 All hands on deck!

The path to the acceptance of SMR as the next major evolution of the HDD poses a cyclic dependency amongst the stakeholders. Users and service providers will not purchase any SMR devices until they are supported by systems software vendors, who have little incentive to support new devices until there is sufficient demand for them; after all, (re)developing OS modules, device drivers and file systems is a non-trivial endeavor with poor ROI.

Furthermore, device manufacturers are skeptical regarding the market acceptance of SMR devices; there is no evidence that the market would accept a radical approach that results in the loss of random-write capabilities in the underlying HDD, thus breaking one of the fundamental assumptions of HDDs and consequently, compatibility with all existing file systems. Given that device manufacture involves a substantial up-front investment on the part of the device manufacturers to deliver a product that retails for around $100, the ROI on SMR devices would not be attractive for manufacturers if the sales volumes are not sufficiently large.

In order to move along this path, both manufacturers and systems software vendors must work together to deliver and support SMR devices, since user demand for low-cost storage is not going to diminish in the near future. If SMR were to deliver good performance, then the demand for SMR devices is bound to be higher than traditional HDDs, assuming that the difference in cost is not very high. To this end, there is a need to identify popular workloads that can work well with SMR to prove to device manufacturers and systems software vendors that it is a step in the right direction, and should therefore be taken seriously.

This research aims to study the use of SMR for Big Data workloads using the Hadoop/HDFS framework. We develop an SMR device emulator and a SMR-aware file system. The SMR device emulator implements a subset of the Banded Device Command Set proposed to the T13 Technical Committee [17] to expose an API to the SMR-aware file system that intelligently places data on the emulated device to mask its sequential write nature. We begin with a discussion on Shingled Magnetic Recording, its architecture and implications in section 2, followed by the approach and assumptions for our research in section 3. Following that, sections 4 and 5 describe the design goals and implementation of the SMR Emulator and ShingledFS, the SMR-aware file system. Section 6 explores Garbage Collection techniques from a theoretical perspective, in the hope that it will provide some insight to researchers that undertake future studies on the subject. We then present the evaluation of the SMR infrastructure and its results in section 7. Future avenues of research are discussed in section 8, and finally we present the conclusions of our research in section 9.

2 Shingled Magnetic Recording (SMR)

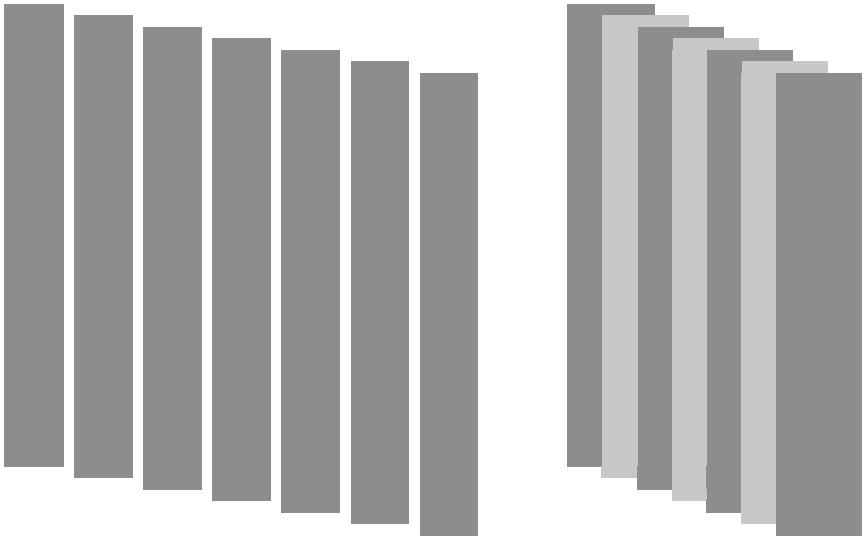

Figure 1: Cartoon rendering of tracks found in traditional HDDs and in SMR. (Colors used to make the overlapping of tracks more visible)

Traditional HDDs store data in concentric tracks that are normally separated by a small gap to prevent inter-track cross-talk as shown in **Figure 1**. The data is encoded using a technique called Perpendicular Magnetic Recording (PMR) that writes data bits by generating the magnetic field perpendicular to the medium so as to pass through it to the substrate and back, as depicted in **Figure 2**. This causes the bit footprint to be very small, resulting in high data density. However, despite the best attempts at miniaturizing components, the minimum width of a track seems to hit the floor at around 50nm [18], although some industry sources have reported track widths as low as 30nm [5]. The HDD uses a single head with two elements attached to the same movable arm; one for reading and one for writing. This results in different characteristics and behavior for each of the elements.

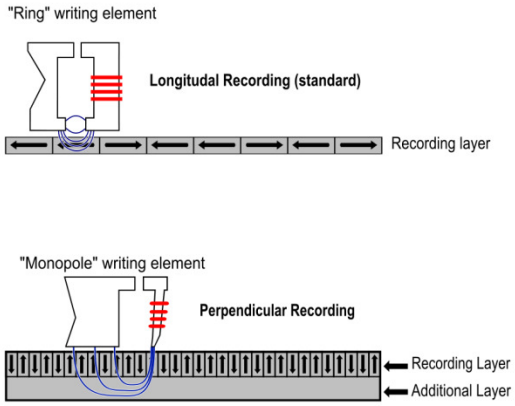

Figure 2: Longitudinal Recording vs. Perpendicular Recording [19]

Modern read elements require only a fraction of the track width laid down by the write element to read back the data. SMR exploits this property to partially overlap data tracks such that the width of the exposed track is sufficient for the read element to read the data stored in the track. Thus by reducing the effective track width, SMR is able to deliver capacity gains as high as a factor of 2.5 [5] [13].

2.1 Breaking tradition, compatibility… and then the disk

While SMR can deliver the much needed capacity gains, it does so by sacrificing the random write capabilities of the device. **Figure 1** shows that it is only possible to update data in the last track of data for SMR devices as updates to any other tracks will result in overwriting of data in tracks downstream. Thus we lose the ability to perform random writes, and consequently, lose compatibility with most existing file systems, which must be rewritten to perform expensive read-modify-write to handle updates. Consider an SMR device with a band size of 64MB. An update to the first 4KB of data in the band will require reading of all of the data in the band into memory, updating the 4KB of data, and then writing back the whole band. Thus, a 4KB write transforms into writing a whole band because of the sequential-write nature of SMR devices, resulting in "write-amplification".

The problem of write-amplification is also found in SSDs, where updates to data in a block may require a (slow) erase cycle to wipe out the entire block before the data can be written. The block is therefore read into a buffer, erased, and the updated data is written back to the block. This approach has worked well for SSDs because the read and write latencies of SSDs are far lower than traditional HDDs. Consequently, even in the worst case, SSDs outperform traditional HDDs.

To deal with the expense of write amplification, SMR breaks the disk surface into smaller pieces called bands [20], consisting of a set of consecutive tracks. Bands are then separated by a gap called the Band Gap. The width of the Band Gap is just enough to ensure that a write to the last track of a band does not interfere with a write to the first track in the next band. Thus, breaking the disk into bands effectively reduces the write-amplification to the size of the band in the worst case. Despite the improvement, this solution is far from ideal, even when band sizes are a modest 64MB, let alone multi-GB sized bands. Consider the same example as before; the update to the first 4KB of data in a 64MB band. The incoming write is for a mere 4KB, but the resultant disk write is 64MB; a write amplification of 1,638,400%! And this only gets worse for bands with larger sizes.

Similarly, deleting data results in the formation of voids or "holes" on the disk. The band has live or useful data before and after the data that was deleted, and the space freed up by the deletion cannot be reused because writing to it might clobber data tracks downstream. These holes then need to be garbage collected. In other words, all the live data in the band must be compacted to be in contiguous data blocks, thereby resulting in free data blocks towards the tail end of the band; blocks that held data that was deleted and can now be reused safely without concerns of overwriting data in tracks downstream. This garbage collection mechanism is yet another expensive operation and SMR needs to be more intelligent when handling updates and deletes.

2.2 Push intelligence up… not down!

As postulated by Gibson et al. [5], SMR devices can be made compatible with existing systems by introducing a shim layer between the SMR device and the host that can manage the translation of random writes into sequential writes and expose a standard interface to the host, much like a Flash Translation Layer (FTL) in SSDs. This allows

SMR devices to be used as drop-in replacements for traditional HDDs. We call this approach Type-I SMR, described in **Figure 3**.

At the start of our research, we had considered Type-I SMR as a possible avenue of exploration. However, over time it became clear that this may not be the best approach for SMR. The Shingled Translation Layer adds additional complexity to the SMR device and makes it harder to deliver consistent performance across workloads. Moreover, experience with SSDs shows that with garbage collection mechanisms on the disk firmware, the task of guaranteeing response latencies is non-trivial.

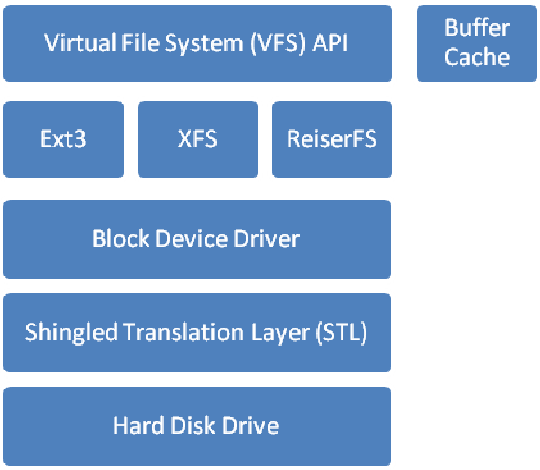

Figure 3: Type-I SMR

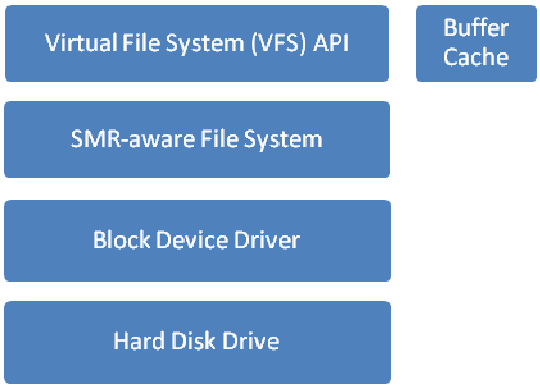

Figure 4: Type-II SMR

Garbage collection is highly dependent on the workload as some workloads tend to generate more dead data than others. E.g. TeraGen generates far lesser temporary files than TeraSort and consequently generates far lesser dead data blocks. This causes the garbage collection process execution to be unpredictable resulting in inadvertent blocking of application requests for reads/writes behind the garbage collection process. Prevention of this phenomenon calls for preemptive garbage collection mechanisms which are expensive to design and implement on the limited processing capabilities available on the disk. For these reasons, we opted not to go down this path and instead, chose to move intelligence up into the file system.

The banded nature of the SMR device makes it similar to log-structured file systems [21], which are well-understood and backed by years of research. Bands on SMR devices can be considered as the on-disk equivalent of a "log" that is written sequentially and garbage-collected over time to eliminate "holes" created by updates and deletes. We call this approach Type-II SMR, described in **Figure 4**, in which the majority of the intelligence is pushed up into the file system to make it "SMR-aware".

2.3 Shingled vs. unshingled

As we have seen, the surface of the disk is broken into bands. However, not all bands are the same. While the majority of bands are shingled, SMR allows for one or more bands to be unshingled; i.e. these bands will support random writes without any overwriting of other data or read-modify-write penalty. The idea is to allow the file system to use this random-write band to handle frequently updated data that have random write characteristics, such as metadata.

Thus, the SMR-aware file system is made up of 2 components:

- A normal random write unshingled partition
- A set of random-read/sequential-write bands making up the Shingled partition

The unshingled partition is used to store file-system metadata, while the actual file data is stored sequentially in the shingled partition.

3 Research Approach and Assumptions

Since we were starting from scratch, we were concerned about the correctness of the code that was being produced by the team. Not only did we have to develop an emulator and a file system from scratch, but we also had to develop the means to identify and isolate errors in our system, and understand the intricacies of some of the libraries that were being used, lest their behavior lead results astray.

Therefore, we took an iterative approach to development and testing. Starting out with the simplest case, we built the code base and test cases to be progressively intricate and complex. This incremental approach resulted in very short code-test-debug cycles that allowed us to identify and debug potential problems before moving to the next stage, thereby enabling us to develop and test small pieces rather than taking on the whole system in a single attempt.

In terms of results, we were not looking for precise measurements, because the test infrastructure was a software emulation of a non-existent piece of hardware. The idea was to conduct experiments on the emulated platform, and get a ballpark estimate of the performance of workloads after subtracting known overheads. Our assumption is that if we observe 15-20% overheads with the emulated system, then further effort on creating device drivers and writing kernel-space code is justified for further SMR research and evaluation.

3.1 Evaluating SMR without an SMR disk

In the absence of real hardware, we decided to develop an SMR Disk Emulator for this research. This decision was also influenced by the draft T13 Banded Device Command Set [17] for use with SMR/Banded devices. While it is hard to study the performance characteristics of a device on an emulator, we feel that the performance of a real SMR device will not be very different from a traditional HDD, since the underlying technology is still the same and the new command set is similar to the existing one, except for minor differences in the addressing model (covered in section 4.2). In other words, we assume that if we were to issue the same sequence of commands on a set of comparable block addresses (similar in terms of on-disk layout such that seek latencies to those blocks are similar) on traditional HDDs and SMR devices, the difference in their performance would be negligible. Since we are relying on the file system to do much of the decision making for data placement and cleaning, any overheads introduced would manifest at the file system level, instead of the device.

3.1.1 Special commands in the new command set

One area where the performance between a real SMR device and the emulator might differ is in determining disk metadata (such as write pointers for bands, discussed later). In the emulated setup, these values might get cached in the host buffer cache, and may be retrieved much faster than a real SMR device where the data will be returned after issuing the command to the disk.

3.1.2 Differences in disk geometry

Another area of difference might be the location of the unshingled band on the device with respect to the shingled bands. Since metadata updates will be written to the unshingled partition, and all file data is written to the shingled bands, the arm may have to move between the unshingled band and the shingled bands a lot more than with traditional HDDs.

However, with the correct placement of the unshingled band on the device, this can be avoided. E.g. if the unshingled band is placed in the middle of the disk surface with shingled bands on either side, the overall displacement of the disk arm will be equal for bands on either side. Furthermore, coalescing of multiple metadata updates into a single write in the host can further mitigate this problem.

3.2 File system in USEr space

We chose to develop the ShingledFS file system to operate in user-space using the FUSE library [22] for several reasons. Firstly, it allowed us to code and debug in user-space, which is a lot easier than having to deal with the intricacies of the kernel. Secondly, FUSE is available as part of the Linux kernel and is relatively easy to work with. Having developed the MelangeFS hybrid file system [23] using FUSE for the 15-746/18-746: Advanced Storage Systems course [24] at Carnegie Mellon University, the team had the necessary experience and skills to develop FUSE-based file systems. We were even able to reuse code from MelangeFS for the first version of ShingledFS (although ShingledFS v2, used in this evaluation, was developed from scratch). Lastly, it is very easy to generate traces of actual VFS-related system calls as seen by the FUSE module, allowing us to take a peek into the disk operations performed by the workload being evaluated.

Using FUSE, we build ShingledFS as a layered file-system that operates on top of the local file system. Each VFS call is intercepted by the FUSE kernel module and bounced back up into the ShingledFS process in user-space, where we perform multiple system calls on the local file system to emulate the operation requested. Thus, in essence, the FUSE-based ShingledFS file system is also an emulation of how an SMR-aware file system would behave, and being an emulation, it introduces a different set of problems, apart from the overhead of multiple transitions between user-space and kernel-space.

3.2.1 Code path extension

When an application issues a VFS-based system call, the system transitions to kernel-space, where the system call is executed in the context of the calling process and then returns to user-space. With FUSE being added to the code path, the call transitions to kernel-space as before, where it is intercepted by the FUSE module and bounced back to the ShingledFS process in user-space. ShingledFS then issues one or more system calls to the local file system to emulate the operation to be performed, and finally returns to the FUSE module in kernel-space, which in turn returns to the application that initiated the system call. Therefore, in the best case, where ShingledFS simply re-executes the same system call, the code path encounters 3 transitions between user-space and kernel-space.

3.2.2 Small writes… even with big_writes

FUSE tends to break up large writes into smaller writes of 4KB each. This is a major source of overhead because a single write() system call issued by the workload for a 1MB chunk of data will be broken down into 256 write() system calls of 4KB each! Thus we encounter 1026 mode-switches when executing this call through FUSE.

Luckily, FUSE version 2.9 introduces the big_writes option that raises the size from 4KB to 128KB. However, even with this option in use, we still see 34 mode-switches! The only way to get past these limits is to have a custom-compiled kernel with the necessary modifications in the FUSE module to support bigger writes.

3.2.3 getattr() injection

FUSE tends to inject quite a few calls to getattr() in order to determine the attributes of files that are being operated on. So even when the application issues one system call, like a chmod() or open(), FUSE injects one or more calls to getattr(), resulting in multiple system calls.

3.3 ShingledFS overheads

Being a file-system emulation, ShingledFS emulates each VFS system call it receives by performing one or more system calls on the local file system. So depending on the workload, we may observe additional mode-switches for each system call when running workloads with ShingledFS.

Furthermore, we are running ShingledFS in single-threaded mode, while the applications above and the file system below it may be multi-threaded. Thus ShingledFS acts as a point of serialization of concurrent operations.

Despite the overheads introduced by the added system calls, we are confident that the performance figures will not be affected by a large margin, mainly because of the intermediate buffer cache. Most system calls end up writing data to the buffer cache, which then coalesces multiple writes and asynchronously writes the data back to disk. Thus, even though we may have a fair amount of system call amplification, most of the data operated on is in memory, thereby reducing their impact on the overall performance figures. Furthermore, for Big Data workloads, the time spent on I/O will have a higher impact on the performance. However, we do expect to these overheads to manifest in the form of higher CPU utilization.

Our tests with single-threaded vs. multi-threaded FUSE did not reveal significant differences between the two modes. So we do not consider this to be a major factor that affects the outcome of the evaluation.

3.4 Evaluation workloads

To evaluate the SMR infrastructure developed, we needed a workload generator that is popular with Big Data applications, which made Hadoop/HDFS the obvious choice. Not only is it used by a lot of Big Data systems, but HDFS stores the data generated by Hadoop as “chunks” in the local file system. Each chunk is stored as a file in the local file system and has a configurable upper bound on its size, which by default, is 64MB. Furthermore, chunks in HDFS are immutable. In other words, once the data has been written, it will not be updated. This has some interesting implications. Firstly, a closed file will never be reopened for writes. Secondly, no offset in the file will ever be written to twice. This allows us to incorporate a few assumptions that made it easier to develop ShingledFS.

The maximum file size property allows us to work with the assumption that a file in ShingledFS will never exceed the size of a band. The immutability property allows us to hold open files in memory until they were closed, at which point they were written out to the SMR emulator, safely bypassing the need for any mechanisms to keep

partial files in memory, or page out cached data to the SMR emulator prematurely, thereby greatly simplifying the buffer cache mechanism modeled in ShingledFS (described in section 5.2.2).

Our primary Hadoop workload is the TeraSort suite [25], which consists of a set of 4 programs. We use 3 of the 4 programs, namely,

- TeraGen – generates a set of random records
- TeraSort – sorts the random records
- TeraValidate – ensures the validity of the TeraSort output

This forms a very good benchmark for storage systems mainly because of the way the 3 programs operate. TeraGen is a write-intensive workload that only writes data to HDFS. TeraValidate, on the other hand, only reads data (and writes a few hundred bytes of results), while TeraSort is a good mix of reads and writes. Also, these 3 programs are very easily expressed in the Map-Reduce programming model and are used extensively for benchmarking Hadoop.

4 The SMR Emulator

4.1 Design goals

The iterative design and research process mandates the need for an emulator that will offer a high degree of flexibility in terms of supporting different approaches to studying SMR, and allowing for easy extension or modification of the parameters of the SMR disk being emulated. This is essential to explore options for fixed and variable band sizes for studying the impact of data placement and cleaning policies, and different internal metadata structures.

Secondly, the emulator needs to be portable across platforms and independent of underlying file systems and hardware. This enables the use of emulated SMR drives on local machines for development and deployment on large clusters for evaluation with minimal administration and change. We also feel the need to emulate a wide range of disk sizes, with small sizes for testing code correctness and larger sizes for evaluation.

Thirdly, the emulator needs to allow for easy retrieval of the data on the emulated drive along with internal metadata structures to study correctness, not just of the data, but also of the emulator codebase itself, which is susceptible to bugs given its development from scratch.

Finally, the ability to store disk images is considered to be desired feature to allow us to return to old disk images, if required, as the research progresses. The disk image is called an Emulated Disk Image (EDI).

- **Implementation**

Keeping the above principles in mind, the emulator has been designed to store all data and internal metadata to one or more files in the local file system. By using the standard POSIX interface, the emulator can emulate SMR drives on any file system that conforms to the POSIX standard, whilst being separated from the underlying hardware.

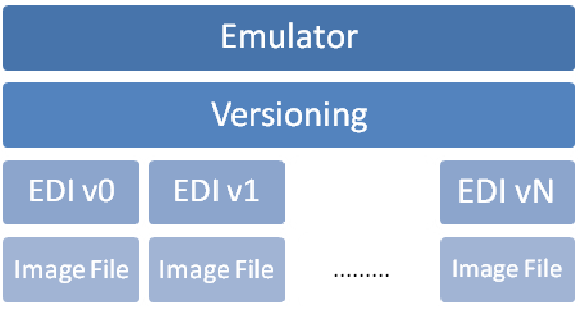

Figure 5: SMR Emulator Design

Figure 5 shows a birds-eye view of the emulator design. This structure addresses all of the design principles set out initially. Starting out at the top of the structure, the Emulator interfaces with the FUSE-based file system and exposes a disk command interface. It then passes the commands to the underlying Versioning module.

The Versioning module allows for creation of multiple formats for the emulated SMR drive. By coding against the interface defined by the Versioning module, individual modules can be created that implement different types of SMR disk formats, utilizing any underlying file system.

Finally, the actual data can be stored in one or more files in the local file system. Depending on the implementation, the data and the disk's internal metadata can be stored in a single disk file or in multiple files.

The Emulator is compiled as a static library in C that can be linked to the file system. We chose this approach for its simplicity and ease of use. We also considered running the Emulator as a separate user-space process that communicated with the file system using standard IPC mechanisms, but felt that it would have been much harder to debug the system. While we feel that this was a good decision, based on our experiences with debugging ShingledFS, this approach may warrant a second look as part of future work.

4.2 The SMR address model

As specified by the T13 Banded Devices Feature Set draft [17], SMR devices will use a Banded Addressing method, where each band will be given a unique number starting from 0, with the block numbering in each band starting over at 0. Banded Addressing is only intended for read and write commands that support an 8-byte LBA field. The 8-byte field is then broken into 2 fields of 4-bytes, with the upper 4-bytes storing the Band Number and lower 4 bytes storing the Block Number, also called the Relative Block Address (RBA) because each block is addressed relative to the other blocks in the band.

As part of internal metadata, the SMR device stores the RBA for each band where the next sequential write should take place. This RBA is called the Write Pointer or Cursor for the band and is managed by the device to ensure that all writes within a band are sequential. In the next section, we will see how this metadata can be retrieved and managed by the file system.

4.3 The emulator API

The Emulator API is modeled on the T13 Banded Device Command Set and consists of 4 commands:

- edi_modesense()
- edi_managebands()
- edi_read()
- edi_write()

This is just a subset of the actual command set. We will be discussing this again in section 8.2.

4.3.1 edi_modesense()

The Mode Sense command is used to determine the characteristics of the disk such as the sector size and the number of bands on the SMR device. This command is expected to be called only once when mounting a device to determine the device's parameters.

4.3.2 edi_managebands()

This is a new command proposed as part of the T13 specification, specifically for SMR/Banded devices. As the name suggests, its purpose is to allow the file system to retrieve or set the band metadata, as required. The subset of T13 operations supported includes:

1. Get Write Pointer for a band
2. Set Write Pointer for a band
3. Reset the Write Pointers for all bands

Future versions can provide more operations as part of this command.

4.3.3 edi_read()

This command is similar to the existing SCSI READ command, except for modifications to the parameters, which are specific to SMR devices - the band number and the Relative Block Address (RBA).

Data can be read from the start of the band to the last written block in the band. Any attempts to read past the last written block, identified by the Write Pointer for the band, will result in an error.

There is the possibility of linking multiple physical bands to form a larger logical band. If this approach were used, then it would be possible to read past the end of the band and into the start of the next linked band. Band linking is not supported in the current version of the emulator.

4.3.4 edi_write()

Like the edi_read() command, the edi_write() command is simply an extension to the existing WRITE commands with parameters specific to SMR devices.

Writes can only take place from the Relative Block Address of the block following the last written block in a band. Any attempts to write at any location other than the block being pointed to by the current Write Pointer will result in an error. Furthermore, it is not possible to write past the end of the band, except for cases where multiple bands are linked.

Once data has been written to a band, the Write Pointer for the band is updated in the internal metadata, to ensure that future writes are written to the end of the band.

4.4 The Emulated Disk Image (EDI) format

The Emulator stores all disk data and metadata in a file in the local file system. This file is based on the Emulated Disk Image Format, which defines an extensible binary file format for storing SMR Emulated Disk images. The format contains a basic file header that describes a magic number (0xE5D1000, that stands for Emulated 5hingled Disk 1mage) and a version number. This feature permits each version to describe its own format for the EDI file, allowing different versions of images to store data and metadata together in a single file or in different files.

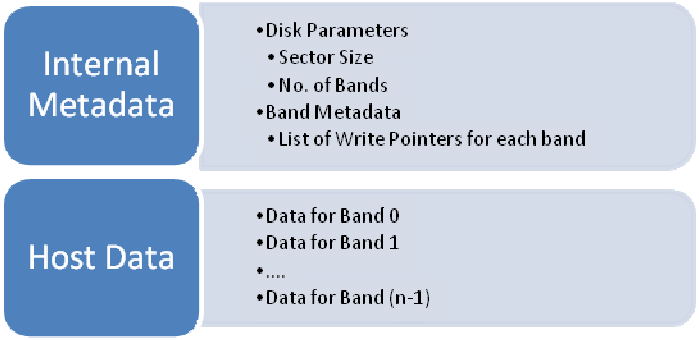

Figure 6: EDI v0 Format

Figure 6 describes the EDI format for version 0 of the SMR Emulator that is used in the evaluations in this dissertation. This EDI version models an SMR disk that has fixed-sized bands. This is the simplest form of the SMR device, requiring the bare minimum internal metadata storage, consisting of the disk parameters such as Sector Size and Band Count and the RBA of the next block to be written, also known as the Write Pointer or Write Cursor, for every band in the emulated SMR device.

4.5 EDI utilities

Developing the emulator is only half the task. Ensuring its correctness of operation requires the study of the data and internal metadata structures. To this end, we also developed utility programs that would allow for creation of Emulated Disk Images (EDIs) and dumping of data and disk metadata from an EDI.

4.5.1 edicreate

The edicreate utility is used to create an Emulated Disk Image. The user only needs to specify the version of the EDI format, the sector size of the emulated disk, the number and size of bands and the path to the output file for storing the disk image. Future versions may add more parameters.

The utility can be used as follows:

```
edicreate -s<block size> -b<band size> -c<band count> -f<filename>
```

The block size, specified in bytes, is the size of a block or sector on the emulated disk. This is the smallest unit of operation on the disk. The band size is an integer value that specifies the number of blocks that a band contains. The band count is again an integer value that specifies the number of bands on the emulated SMR device. Lastly the name of the EDI file is specified.

4.5.2 edidump

The edidump utility displays the internal metadata structures for the emulated disk image, specifically focusing on the Write Pointer for each band. It can be invoked as follows:

```
edidump <path to EDI file>
```

5 ShingledFS: An SMR-Aware File System

5.1 Design goals

In designing ShingledFS, our intent is to model all aspects of a file system in a simple manner to ensure that the implementation is easy to code and test. This includes the underlying storage device, the VFS-compliant file system and the intermediate buffer cache. At the same time, we also want the ShingledFS architecture to be as open as possible to allow for future modifications without much difficulty.

5.2 Evolution

The iterative design process has shaped ShingledFS over the past year and this is the 3rd generation of the file system. Before we get into the nitty-gritty of its implementation, an overview of its evolution is in order.

5.2.1 MelangeFS: The academic fore-father

The origins of ShingledFS can found in the MelangeFS Hybrid File System [23] – an academic project developed as part of the 15-746/18-746: Advanced Storage Systems course [24] at Carnegie Mellon University. The idea was to create a hybrid file system that delivered the speed of Solid State Disks, yet had the low cost/byte storage capacity of traditional HDDs. This was based on the fact that SSDs, especially NAND flash devices, outperformed traditional HDDs for small random accesses magnitude, albeit at a much higher cost for sequential access pattern workloads [26].

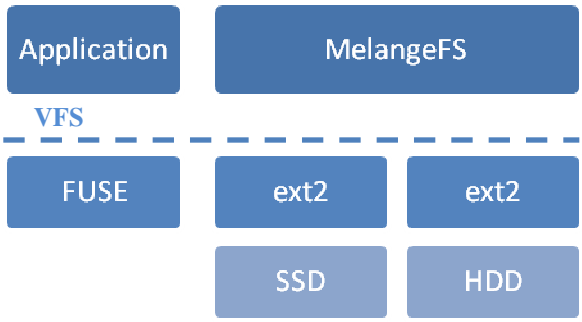

Figure 7: MelangeFS Architecture

When an application issues a VFS API call on a file in the MelangeFS mount point, the operation is redirected to the FUSE kernel module, which bounces the call up to the MelangeFS user-space process. MelangeFS then proceeds to process the request, and returns the response to the FUSE kernel module, which forwards it back to the application via the VFS interface.

MelangeFS stores the file system name space, small files (small here is defined by the user, which in the case of the project was 64KB), and all file-system metadata on the SSD (mounted by an ext2 file system). When a file grows to exceed the user specified size threshold, MelangeFS migrates the file from the SSD to the HDD (also mounted by an ext2 file system) and replaces the original file on the SSD with a symbolic link to the newly migrated file on the HDD. This ensures that all VFS operations are transparently routed to the new copy of the file.

While migration of large files ensures that the HDD only sees large sequential reads and writes in terms of file data, all metadata queries, which are essentially small random reads/writes are also redirected to the HDD, thus beating the purpose of the whole system. Therefore, MelangeFS replicates the attributes of each migrated file in the

extended attributes of a hidden file on the SSD, thus ensuring that the SSD seems small random reads/writes and the HDD sees large sequential reads/writes.

5.2.2 ShingledFS v1: One file per band

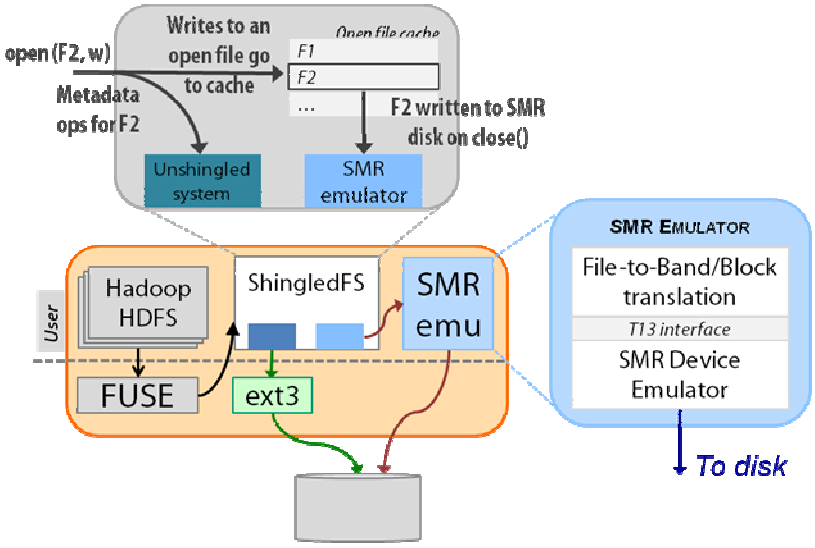

Figure 8: Architecture of ShingledFS v1 (Image by Swapnil Patil)

Figure 8 describes the architecture of ShingledFS v1 where file data and metadata is split between the shingled and unshingled sections of the file system. The unshingled partition is modeled using a directory on the local file system and it stores the namespace of ShingledFS, along with all the file metadata that is replicated in the extended attributes of a set of hidden ShingledFS internal files. All the file data is stored in the shingled partition, modeled by the SMR Emulator. Thus, it uses the same idea as MelangeFS, and this allowed us to reuse a fair bit of code.

ShingledFS models the OS buffer cache using an in-memory file system, which in this case is tmpFS. When a file is opened in ShingledFS, the copy of the file on the unshingled partition (that holds the namespace) is replaced with a symbolic link to a new file of the same name in the tmpFS partition. The contents of the file are then read from the SMR Emulator and written to the newly created file on tmpFS. All further operations on this open file now take place on the tmpFS copy of the file. When a file is ready to be closed, ShingledFS writes its data back to the SMR Emulator, closes the tmpFS copy of the file and updates its replicated attributes. Files are only deleted from the tmpFS partition when there isn't enough space to create/store files being created and operated on. Thus, tmpFS model allows us to easily and transparently model the behavior of the OS buffer cache.

Our first attempt at modeling an SMR-aware file system was not perfect. As it turned out, the model was far too simplistic to model any real-world file systems. Firstly, it did not account for FUSE breaking up large writes into 4KB writes. This was a very big source of the overhead. But more importantly, the buffer cache model had a major flaw. In traditional file systems, most operations are asynchronous, including `close()`. This allows the `close()` system call to return immediately, and the buffer cache later writes the data out, performing more optimizations (like coalescing multiple writes into a single write). ShingledFS v1 did not support this.

When a file was closed, ShingledFS wrote out all of the contents of the file from tmpFS to the SMR Emulator before returning. Consequently, for large files (64MB chunks), `close()` introduced a latency of over half a second because of the synchronous writing, thus lowering the overall throughput considerably.

5.3 ShingledFS v2 design

Based on our findings and the lessons we learned from ShingledFS v1, we developed ShingledFS v2 from scratch as we wanted to raise the complexity of the model a little further, namely packing of multiple files into a single band, and the beginnings of garbage collection.

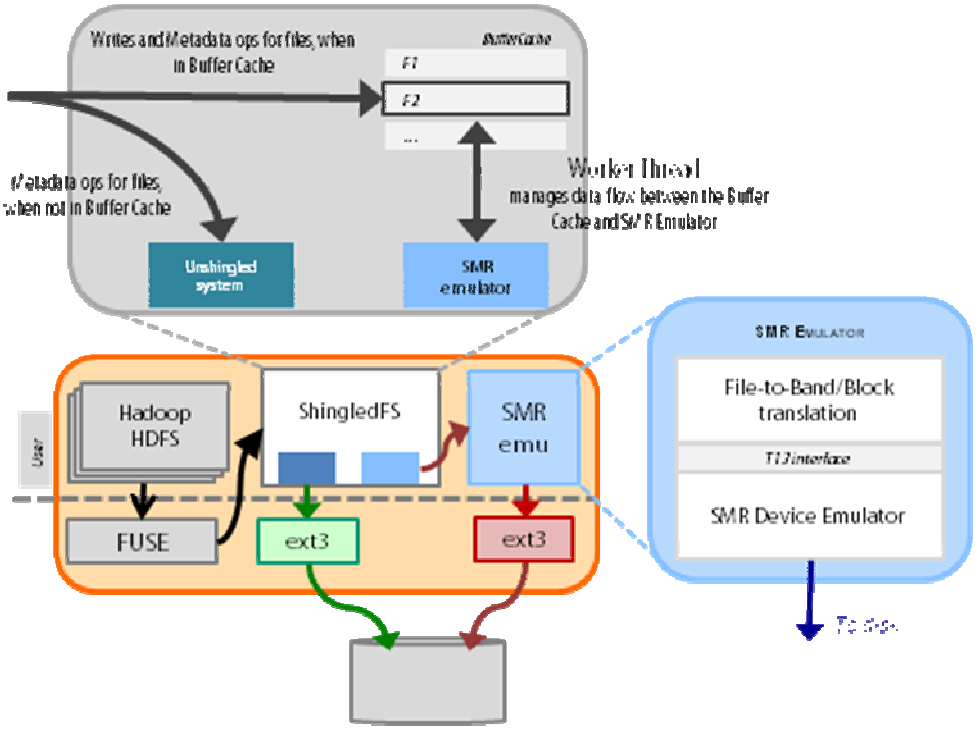

Figure 9: Architecture of ShingledFS v2 (Original Image by Swapnil Patil)

5.3.1 Inodes… we meet again!

In ShingledFS v2, we introduce the concept of an inode that stored some file metadata, namely, the Band ID of the band where the file is stored on the emulator, the RBA of the block where the file begins within the band and an extent specifying the no. of contiguous blocks that the file spans.

Thus each file has a unique inode that stores the details of where the file is stored on the SMR Emulator. Newly created files are assigned a BandID of 0 because the unshingled band on the SMR device will always be assigned band ID 0. Thus it indicates that the file has just been created and has never been written to the shingled section.

5.3.2 `close()` and move on…

ShingledFS v2 also introduces asynchronous `close()` which was one of the primary reasons for the poor performance of ShingledFS v1. To this end, a new Worker thread is spawned that is responsible for handling data movement between the SMR Emulator and the ShingledFS Buffer Cache model. This also doubles as a request queue modeling the Disk Controller that receives request from the host, executes them and returns results to the host.

When a file is opened, the FUSE thread performs the necessary operations to create and open the file on tmpFS. It then passes a request to the Worker to load the file's data from the Emulator and waits for the file data to be loaded, similar to a thread blocking on disk I/O. When the data is loaded, the Worker awakens the FUSE thread and the operation proceeds.

When a file is closed, the FUSE thread passes a request to the Worker to close the file, and returns. The Worker thread then writes out the file's data to the SMR Emulator, closes the file and updates the its replicated attributes.

5.3.3 Clean up the mess

Thus far, the system only uses the space available on the SMR Emulator without any notion of live and dead data. As operations update/delete data on the SMR Emulator, "holes" crop up representing space that cannot be reused without resetting the Write Pointer/Cursor for the band to a position prior to the hole. In ShingledFS v2, we address some of these issues using logs.

For each band, two logs are maintained that keep track of additions and deletions to the band respectively. When a file is written to the band, an entry is made in the Append Log for the band noting the inode no. of the file, its starting RBA and extent. Likewise, when a file is deleted or updated, its disk space needs to be garbage collected. This time, an entry is made to the Delete Log that records the same information as the previous log. Additionally, it also makes note of the record number of the Append Log entry for the file. This is used as an optimization to easily locate the Append Log entry for the file without having to read and parse the log again.

With these two logs, a Garbage Collector can identify live and dead data within the band and migrate the live data out, and reset the Write Pointer/Cursor to the start of the band to allow it to be reused.

5.4 ShingledFS v2 implementation

In implementing ShingledFS v2, we knew that the simplistic approach taken in v1 would not make the cut. So we decided to rebuild ShingledFS from scratch, incorporating all that we had learnt from the first version.

Asynchronous close was one of the top priorities along with a more robust and well performing buffer cache mechanism. A file can either reside on the unshingled partition or in tmpFS, with a symbolic link on the unshingled partition pointing to it. Consequently, when implementing functions such as `chmod()` and `utime()`, we decided to perform the operation on the extended attributes or the file itself, depending on its location. If the file was closed, then the operation was manually performed on the copy of the file's stats stored as replicated attributes on the unshingled partition. If the file was open, then the operation proceeded normally. The only exceptions were the extended attribute functions which were performed on the hidden file on the unshingled partition that was used to replicate the attributes of the actual file. This was to ensure that the extended attributes weren't lost when the file was deleted and recreated on tmpFS on `open()`. Finally, this has an impact on the implementation of the `getattr()` function, which returns results based on whether the file is open or closed.

This approach eliminates the need to update the file size in `write()` and the access timestamps in `read()` and `write()` as they are now performed implicitly by the underlying file system. For Big Data workloads, these functions will be in all major code paths, and therefore need to be streamlined to eliminate as much overhead as possible. In 7, we will see the benefits of this implementation.

5.5 ShingledFS utilities

5.5.1 edistats

This utility uses a combination of the EDI file and the unshingled partition that stores SMR's internal files to display live/dead data blocks stats for the emulated drive. It can be used as follows:

```
edistats <EDI path> <unshingled partition path> <output file path>
```

This utility reads the metadata in the EDI file along with the append and delete logs created by ShingledFS and generates the following statistics for each band:

- The total number of blocks in the band
- The position of the Write Pointer for the band
- The number of blocks of live data in the band
- The number of blocks of dead data in the band

The output from this utility can be seen in section 8.1.

5.5.2 profiler

While not directly linked to ShingledFS or the SMR Emulator in any way, this utility warrants a discussion in this section along with others in the ShingledFS suite. The profiler is used to identify the FUSE functions that are called when a particular application or workload is executed and the number of times each function is called. This is useful with analysis of results to identify which functions the workload relies on most, or to tweak ShingledFS to work better with certain workloads.

In essence the profiler utility is a FUSE file system that simply executes the VFS system call executed by the workload. The only difference is that because of the inclusion of the FUSE kernel module into the active code path, all of FUSE's internal behaviors such as the breaking up of `write()`s are visible to the profiler. The profiler simply counts the number of times each FUSE function is executed, giving an insight into the workload being executed.

The profile can also be used with batch scripts to profile each workload executed by the script. This is accomplished using signals. Each time the profiler process receives a USR1 signal, it outputs the values of all counters to the output file and resets the counters to 0. Thus the batch scripts need only to be modified to send a signal to the profiler process.

The profiler is executed as follows:

```
profiler <FUSE mount point> <FUSE backing directory>
```

Following the loading of the profiler FUSE file system, the workload can be executed as it usually is, ensuring that its output is being stored within the FUSE mount point of the profiler. Once the workload has executed, a USR1 signal needs to be sent to the profiler to output the counters to the output file. We profile the workloads that we use in 7.

6 Garbage Collection

Like traditional log-structured file systems, ShingledFS also needs to garbage collect bands regularly to consolidate file data and ensure that the bands have as much usable free space as possible. However, ShingledFS needs to deal with bands in the multi-megabyte to the multi-gigabyte range and consequently, cleaning is extremely expensive and needs to be handled with care. Not only does a lot of data need to be migrated, but it needs to be done in a reliable manner and, in some cases, when the disk is still in use! This also has a major impact on response time variance as application requests can get queued behind the Garbage Collector, thereby forcing them to wait until the Garbage Collector has finished its task; it may take a few milliseconds to move just a few blocks of live data, or a few seconds to move a half a gigabyte of live data, depending on the amount of live data in the band, and whether the Garbage Collector is preemptive.

Since cleaning has a direct impact on performance, it should not get in the way of work unless absolutely essential. The idea is to work for the shortest amount of time and get the most return in terms of space freed. To this end, there is a need to study data placement and cleaning policies. While these are only partially implemented in ShingledFS, and therefore, not evaluated, we hope to present some food for thought for future work in this section.

6.1 Data Placement

The importance of data placement on the performance of ShingledFS cannot be overstated. Simple data placement policies make it efficient for the Worker to quickly identify a band to write data to, while more elaborate policies might lower the throughput of the file system. However, care needs to be taken in placing data correctly, even more so in the absence of mechanisms that allow files to span band boundaries. An interesting observation was made during our experiments with regards to the importance of data placement.

The experiment involved generating data with TeraGen using an EDI with band size 64MB. We noticed a very high number of bands being used for a very modestly sized dataset. The data placement policy mandated that data would be placed in the last used band. If that band didn't have enough space to accommodate the file, then a free band would be allocated. If no bands were free, then any band with enough space to accommodate the file would be selected. When running the experiment, HDFS wrote chunks of size 64MB along with an associated .meta file that

was a few hundred bytes. The .meta file was written to disk first, which occupied the first few blocks in the band. When the chunk was to be written, the space remaining in the band was not sufficient for the full chunk. Consequently, it was written to a free band. When the next chunk was ready to be written, again the .meta file was written first. This time, the last written band was full, and a new band was allotted. When the chunk was to be written, again it was seen that the remaining space in the band was insufficient, and a new band was chosen. Thus, the *edidump* report for the EDI showed every even numbered band to be full, and every odd numbered band with less than 10% space usage! In such a scenario, cleaning would be triggered fairly often, especially if the size of the data being written to the band is comparable to the size of the band.

6.1.1 First-fit

This is one of the simplest algorithms to implement, and will usually mean that the Worker doesn't have to wait for long to identify a band that can be used to store the file. However, as we can see from the previous anecdote, this may result in very poor performance when the sizes of files are comparable to the size of the band.

6.1.2 Best-fit

This is a slightly more complicated algorithm as it involves searching for the best band that can fit the file. Ideally, this would be a band that is just large enough to fit the file. However, searches can be expensive, especially, when incoming data has a high arrival rate. On the other hand, it can also be argued that the cost of searching is much lower than having to clean.

6.1.3 Age-fit

One option is to group data by age. Studies have shown that the current age of a file is a good predictor of future lifetime of the file, given no other information [21]. In other words, the longer a file persists, the higher is the probability that it will continue to persist. With SMR, we would like to exploit this property by ensuring that the oldest data is placed at the head of a band and colder data gets placed towards the tail of the band, where the cost of cleaning least.

6.2 Cleaning

Waiting until the last minute to clean a band might not be the best strategy as it forces the application to wait while the band is being cleaned. Also, while it is tempting to do so, cleaning a whole band in one attempt may also be a bad idea. Depending the workload, the application might get queued behind the Garbage Collector cleaning out a 1GB band, forcing it to wait for the band to be cleaned. This depends heavily on the amount of live data in the band, rendering response time variations to be high.

6.2.1 Whole band cleaning

This strategy involves cleaning a whole band in one step and might work for SMR devices with small bands and low data arrival rates. The returns of this policy are also high; each round of cleaning will result in at least one full clean band. However, this policy might exact a high penalty on applications that write data while a cleaning operation is in progress.

Picking the band to clean is also of importance. We want to maximize our return for the amount of work done. Consequently, the strategy should be to pick the band that has the largest amount of dead data in it. This ensures that we get the maximum return for the minimum amount of live data moved.

6.2.2 Tail cleaning

As the name implies, this strategy attempts to clean data at the tail end of each band. The idea is to maximize the usable free space at the tail end of bands. This might be a good option for SMR devices with large bands in the multi-gigabyte range.

The Garbage Collector can start migrating files from the tail end of bands to incrementally free up space. Since the update happens at the tail end, the Write Pointer for the band can be reset as soon as the file data has been moved, thereby resulting in an immediate increase in the amount of usable free space. Moreover, the Garbage Collector can migrate one file at a time or choose a smaller unit of data migration to raise preemptibility (although this might add a lot of complexity to the garbage collection algorithm which must keep track of partially migrated files and their possible use by applications, the band that the data was being migrated from and the one the data was being migrated to ensure they aren't used to write new data while the migration is in process, thus leading to fragmented storage of file data, and any locking mechanisms to handle concurrency).

The amount of space gained after the migration of live data blocks from the tail of the band depends on whether or not there is a "hole" of dead blocks just prior to the live data blocks migrated. If so, then the return (the number of blocks freed) is higher than the work done (the number of live data blocks migrated). So one aspect of the strategy is to identify bands where the distance between the Write Pointer and the last block of the last "hole" in the band is the least. This, however, isn't sufficient because the size of the "hole" also comes into play.

	Band A	Band B
Size of last hole in the band (D) (blocks)	10	30
Size of live data at the tail end of the band (L) (blocks)	10	15
Space gained on tail cleaning (S = D + L)	20	45
Profit = (S / L)	**2**	**3**

Figure 10: Tail Cleaning Band Selection

Consider 2 bands where band A has 10 blocks of dead data, followed by 10 blocks of live data at its tail and band B has 30 blocks of dead data, followed by 15 blocks of live data at its tail. If we consider only the distance of the hole from the Write Pointer, then we would choose band A for cleaning. However, as can be seen in Figure 10, the returns on selecting band B are higher.

Thus, the ideal band to clean is the one where the ratio of space gained from tail cleaning (S) and the size of live data at the tail (L) is the highest.

6.3 Cleaning triggers

Another important question with regards to cleaning is when to trigger cleaning. The answer to this question may depend entirely on the workload. In this section we define a couple of ideas for cleaning triggers.

6.3.1 Idle trigger

This is a simple policy where ShingledFS tries to identify when the disk is idle and schedules cleaning. This could be as simple as waiting for a pre-defined amount of time or slightly more complex such as a when the moving average of the arrival rate of data drops below a defined threshold.

Each time that ShingledFS is idle, it starts a timer that fires after a pre-defined interval. If the timer fires, then it is known that the file system has been idle for the defined interval of time and therefore cleaning can be triggered. If however, the file system receives one or more incoming requests, then the timer is cancelled and reset when the requests have been processed.

The selection of the interval may be an interesting avenue of exploration as it will vary depending on the workload and the arrival rate of requests.

6.3.2 Free space trigger

This policy triggers cleaning when the amount of usable free space drops below a pre-defined threshold. Usable free space can be defined as the sum of the number of free blocks available at the tail end of bands or as the number of free bands. Furthermore, multiple thresholds can be defined, such that the priority of the Cleaner is varied as

lower thresholds are met. For example, when the usable free space remaining is drops below 30%, then the Cleaner can be run in low priority mode, where it can be preempted by incoming application requests. However, if the amount of free space drops below 10%, then the Cleaner can be run in high priority mode where application requests are forced to wait, to ensure that the disk does not completely fill up.

This policy may be more suited to SMR devices with large bands where the arrival rates are low enough that background cleaning is sufficient to keep up with the rate of space consumption.

7 Evaluation

7.1 The SMR test-bed

We conduct testing on the Marmot cluster [27] at CMU, where we use 6 nodes (1 master and 5 compute and data nodes), each of which has an Opteron 252, 64-bit, 1 MB L2, 1Ghz HT CPU, 8GB RAM, and one Western Digital Black SATA 7200 rpm 2TB Hard Disk Drive, running Ubuntu 11.04 with ext3 as the local file system. The nodes are connected via Gigibit Ethernet over 2 independent switches: a 152-port Extreme Networks BlackDiamond 6808 Ethernet switch and 6x 24-port Dell PowerConnect 5224 Ethernet switches.

The Linux Kernel Compilation test was conducted on a single node in the same cluster. For the Big Data test cases, we use Hadoop 0.20.2 using most of the defaults including the HDFS chunk size of 64MB. We change the replication factor to 1 to reduce network traffic.

Each node runs an instance of ShingledFS and all data generated by HDFS is stored in ShingledFS on all of the nodes. Furthermore, all temporary files generated by Hadoop (except the log files and PIDs) are also stored in ShingledFS.

7.2 Test cases

For the SMR evaluation, we have chosen 2 workloads: the Linux kernel compilation test and the Hadoop TeraSort suite [25].

While not a real case for Big Data, the Linux Kernel compilation test is a good stress testing workload. Being a rather complex sequence of operations, it also serves to expose any bugs in the file system code, which was the initial intent of the exercise. However, since it operates on very small files, it can be used as a model of real-world worst-case of ShingledFS, which works best with large writes.

Hadoop is a very popular distributed computing framework that provides an easy-to-use parallelization abstraction for tasks that can be broken down as one or more map and reduce operations. A popular benchmark is the TeraSort suite, which we use to evaluate the SMR infrastructure for Big Data. Before we proceed to the evaluation and the results, we shall take a look at the profiles of these workloads to gain more insight into their operation on FUSE.

7.3 Workload profiling

We use the *profiler* utility to construct a profile for each of the workloads that we use in the evaluation and present graphs depicting the number of times each FUSE function is called by the FUSE kernel module as a result of the execution of the workload on FUSE. Keep in mind that the graph is scaled logarithmically to ensure visibility of values and the operations appear in decreasing order of the number of times called by the FUSE kernel module.

We start out by profiling the Linux Kernel compilation workload, which consists of 2 operations: untaring the linux kernel tarball, and compiling the kernel. Figure 11 shows the profile of the untar operation, while the compilation profile is shown in Figure 12.

The graph shows that the untar operation is mainly a write-based operation that consists of `getattr()`, `write()`, `utime()`, `mknod()`, `open()`, `release()` and `mkdir()`. The calls to `getxattr()` are made immediately after each call to `write()` to retrieve security information that are stored as extended attributes. This was verified from reviewing logs files that captured function call information.

Moving on to Figure 12, it is clear from the figure that the Linux kernel compilation lives up to its reputation. It generates a total of 5.8 million requests (keep in mind that the graph has been scaled logarithmically to ensure maximum data visibility), the most prominent being `getattr()`, `read()`, `open()`, and `release()`.

Following the Linux kernel compilation, we profile the TeraSort suite. As we know, this suite consists of 3 programs, TeraGen, which is a write-intensive workload, TeraSort, which is a read/write workload and TeraValidate, which is a read-intensive workload. The profiles for these programs are displayed in Figure 13, Figure 14 and Figure 15 respectivelyss.

The TeraGen workload generates most calls of `write()` and `getxattr()` (used to read security permissions from extended attributes). TeraSort on the other hand seems to be fairly write oriented. This is contrary to our belief that TeraSort is a read/write workload. The reason for the difference between the number of calls to read() and write() could be because the read() calls read a much higher volume of data as compared to write(). We hope to verify this as part of future research. Lastly, TeraValidate issues most calls for `getattr()` and `read()`, suggesting that this is a read-oriented workload.

Now that we understand the workloads and their primary functions, we proceed to the evaluation.

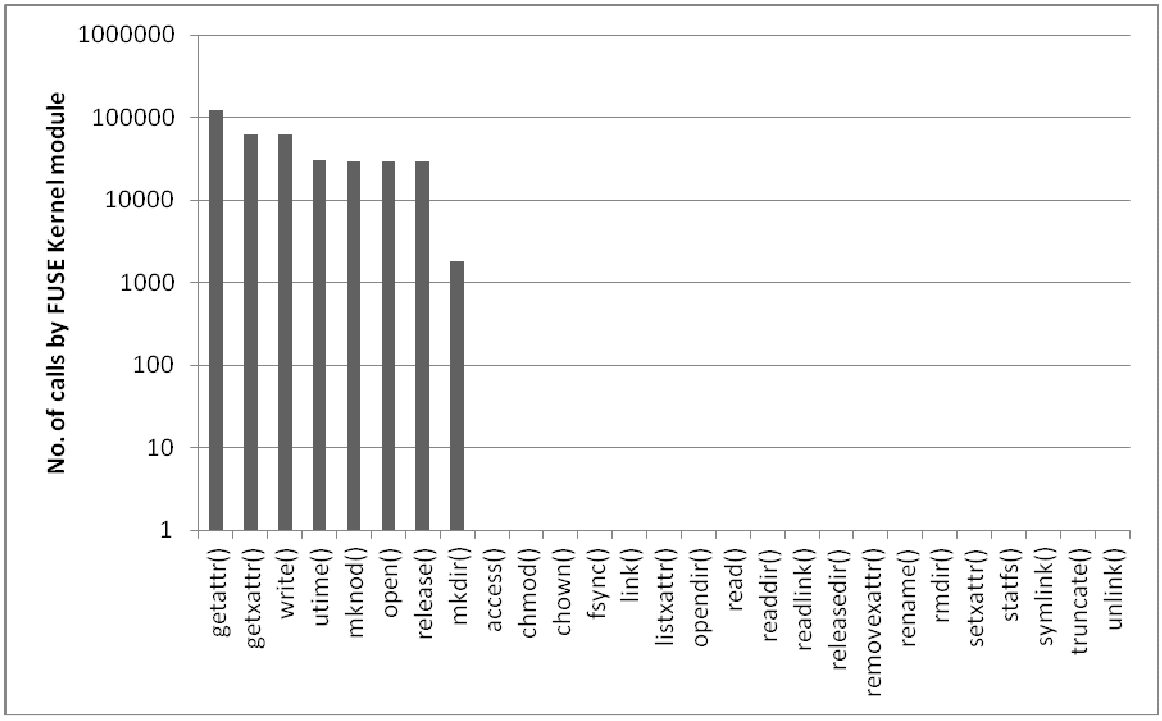

Figure 11: Workload profile: Linux Kernel Untar. Total number of calls: 367,315

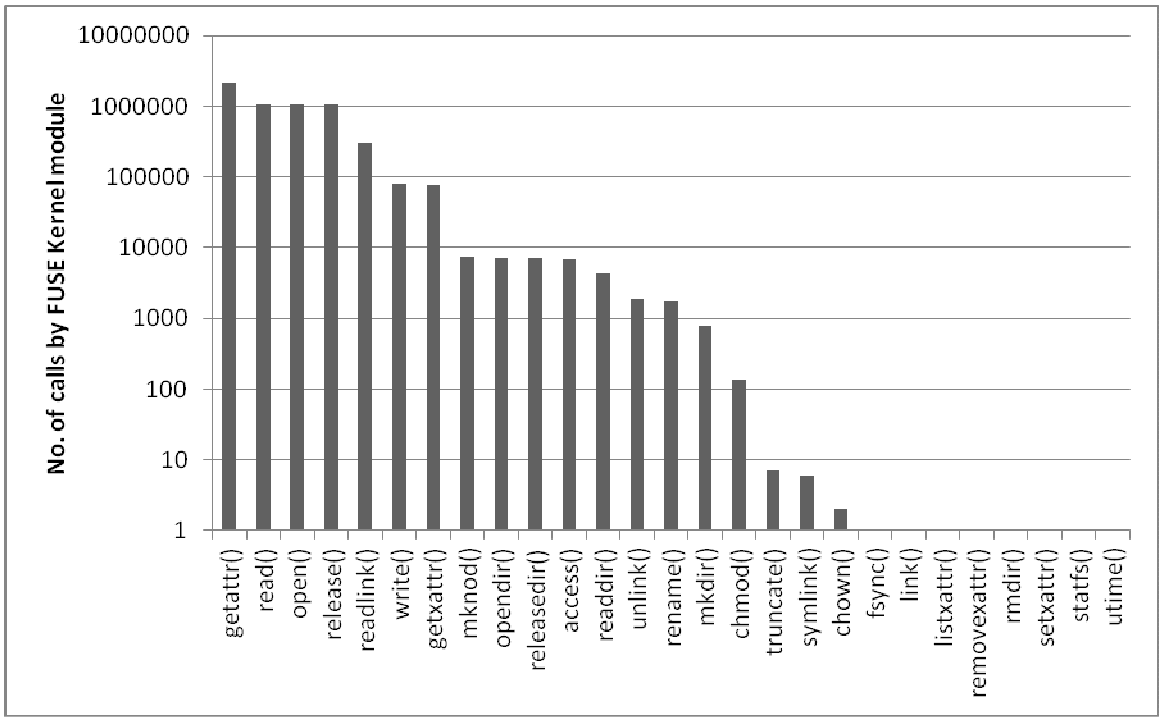

Figure 12: Workload profile: Linux Kernel Compilation. Total number of calls: 5,813, 167

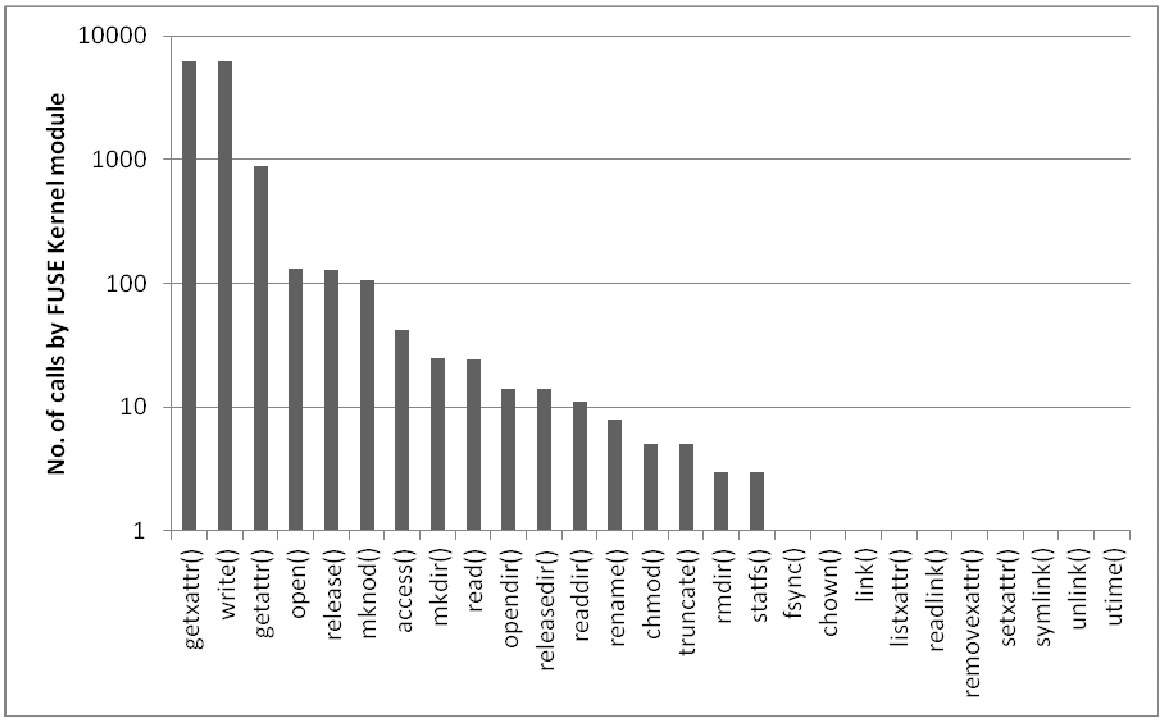

Figure 13: Workload profile: TeraGen. Total number of calls: 13,976

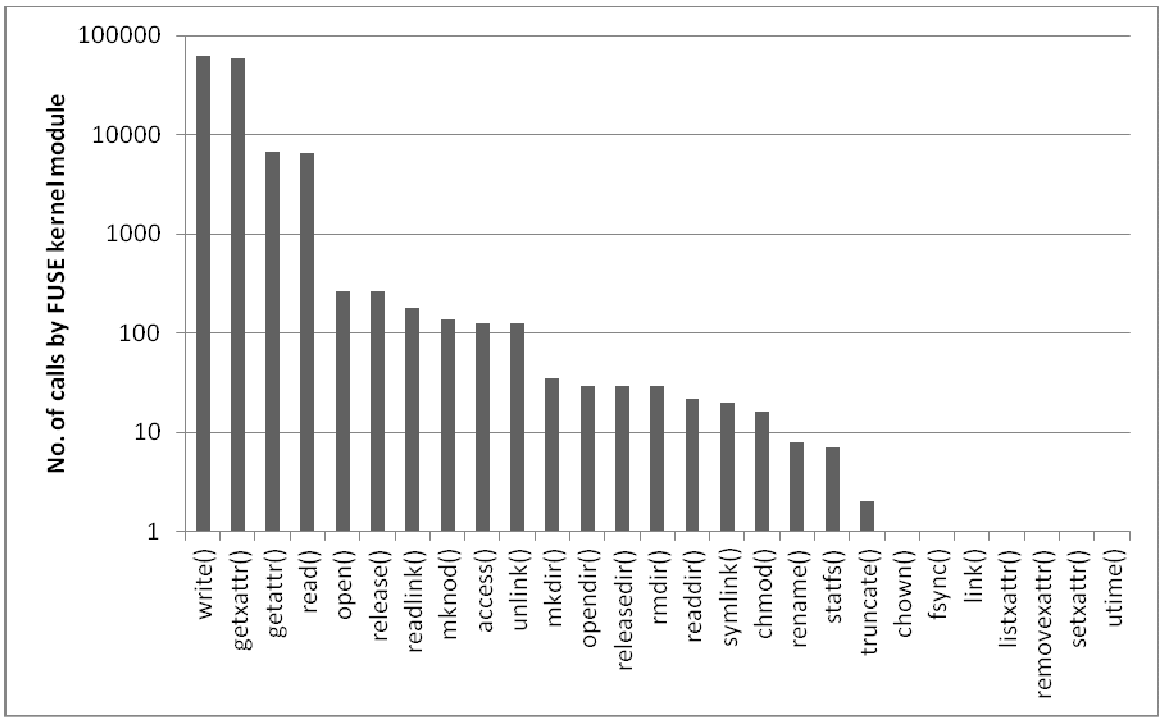

Figure 14: Workload profile: TeraSort. Total number of calls: 136,334

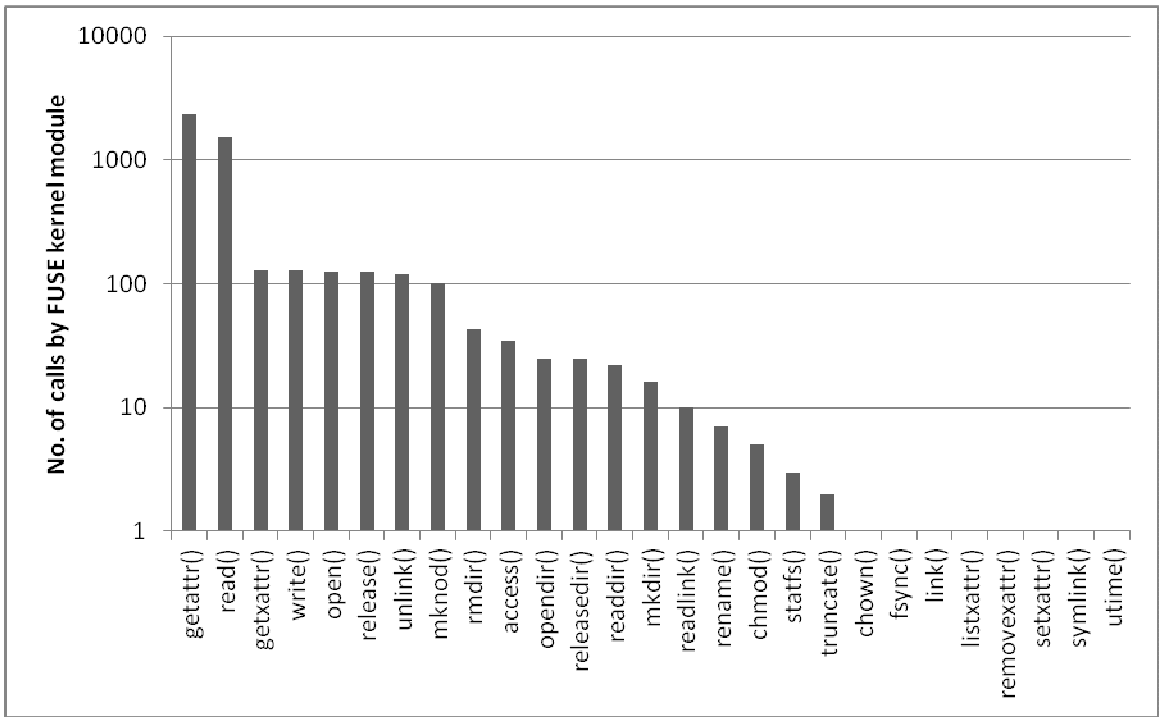

Figure 15: Workload profile: TeraValidate. Total number of calls: 4,809

7.3.1 Compiling the Linux kernel

Using the Linux 2.6.35.10 kernel, the test untars it into a directory (a regular directory for ext3, and the respective mount point for the FUSE and SMR cases), and then compiles it. The Linux time utility is used to record the resource usage statistics of the untar operation and the compilation process.

It should be kept in mind ShingledFS does not have any cleaning mechanisms in place. Thus while the workload is running, there is no cleaning being performed on the bands.

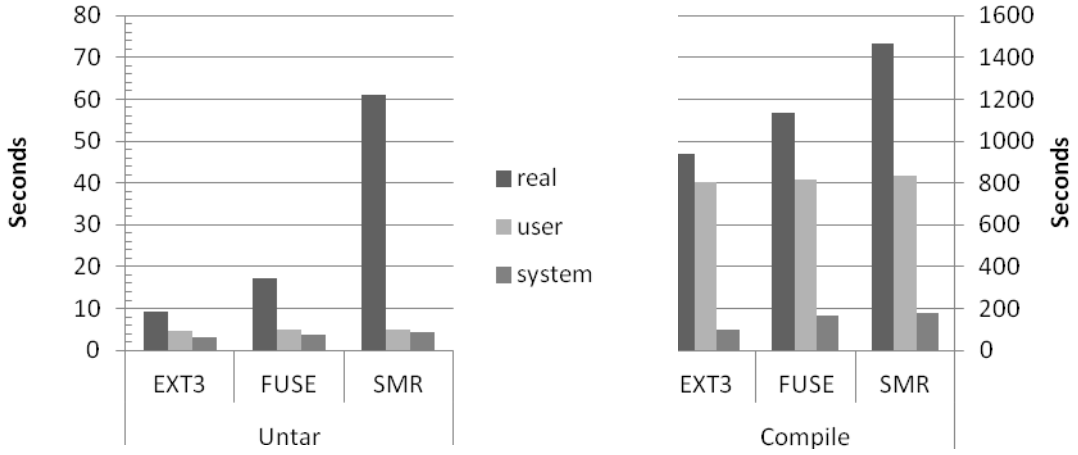

Figure 16: Linux Kernel Compile Test

Figure 16 shows the results of the Linux Kernel Untar and Compilation test. The difference in run-time between ext3 and SMR is much more pronounced during the untar operation, with the SMR setup lagging in performance by 555%! We feel that this is probably because the untar operation uses the `getattr()`, `utime()`, `open()`, `release()` and `mknod()` functions, all of which are expensive in terms of their overheads. The overhead introduced by FUSE is a shade under 86%.

Moving on to the compilation process, the differences between FUSE and SMR become much less pronounced; the SMR performance lags only by 56% compared to ext3; a huge improvement over the untar operation! It fares even better when compared with FUSE, at just 29% overhead. This can be considered a good worst case scenario for SMR as Big Data workloads operate on much larger sizes and usually involve a mixed reads and writes.

As part of future work, we intend to determine the specific causes of the numbers we are seeing in our evaluation. This will involve extracting more information using the profiler, including the sizes of reads and writes, and the total amount of data read and written by the workload, along with runtime statistics for the FUSE and ShingledFS processes.

7.3.2 The TeraSort suite

For the TeraSort suite, we configure Hadoop to have 1 map slot and 1 reduce slot per node. TeraGen spawns 5 mappers to generate the data. TeraSort uses 5 reducers for sorting, but we do not explicitly specify the number of mappers. Lastly, TeraValidate operates on the 5 output files produced by TeraSort (1 output file per reducer).

For the evaluation, we create an SMR Emulated Disk Image that has 500 1GB bands, to give an effective SMR disk size of 500GB. ShingledFS is configured to use 4GB of memory for tmpFS. This is almost half the memory available in the system. However, it is the minimum required to get through the tests.

With 5 mappers, Hadoop splits the dataset 5 ways. For the 16GB dataset, this implies that each node operates on 3.2GB of data and TeraSort will form 5 buckets (since TeraSort is configured to use 5 reducers) of intermediate data. Since SMR maintains all open files in memory, the size of tmpFS has to be at least 3.2GB plus additional

space for other data chunks. Thus we configured the size of tmpFS to be 4GB, which is half the memory in the machine

As part of future work, we intend to improve ShingledFS to allow for spilling over of open files in tmpFS to SMR so as to have a more accurate model of the buffer cache.

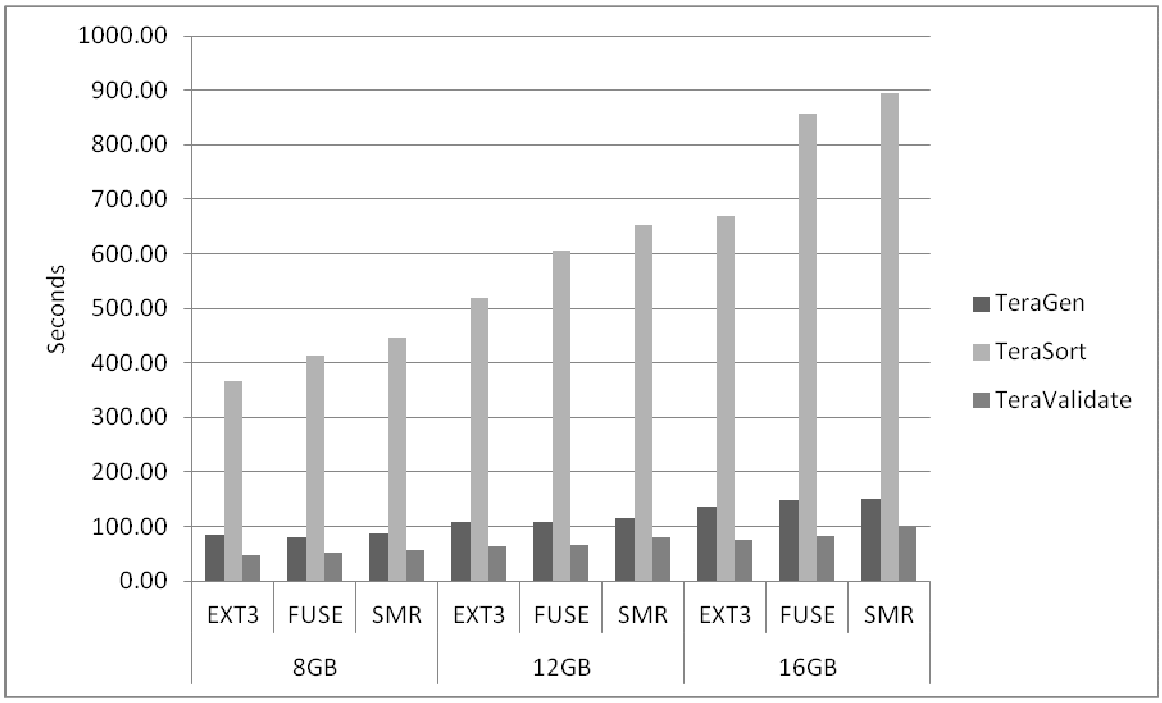

Figure 17: TeraSort suite across ext3, FUSE and SMR for 8GB, 12GB and 16GB datasets

As can be seen in Figure 17, our assumption regarding performance as a function of the size of the dataset holds true. As the size of the dataset increases, the performance becomes increasingly dependent on I/O. Thus the effects of the overheads introduced by FUSE and other elements tend to decrease as the dataset size increases.

There are a few interesting observations in Figure 17. The differences between the ext3 and FUSE bars shows the overhead introduced by FUSE and the same between the FUSE and SMR bars is representative of the additional work being done in SMR to emulate the file system operations.

The performance of TeraGen doesn't vary much between FUSE and SMR. In fact, it varies from 6% for the 8GB dataset to 2% for the 16B dataset; it is inversely proportional to dataset size. This is an artifact of the implementation of the `read()` and `write()` functions in ShingledFS, which work great for Big Data workloads. As the workloads gets larger, the performance is dominated more by disk than CPU. Thus, most of the overheads that we discussed in section 3.2 do not have a significant impact on results.

For the TeraSort workload, the difference FUSE and SMR diminishes as the size of the workload increases, dropping to just 4.34% at 16GB. Again, this is expected because the code paths for FUSE and SMR are similar for `read()` and `write()`, and only differ in other FUSE functions, which aren't called very frequently in the TeraSort workload.

8 Future Work

8.1 A second look at cleaning

Taking one of the EDI images from the experiments, we use the edistats utility to identify live vs. dead data in the bands. For a simple case of TeraSort on a 1GB dataset using a 100GB EDI with 1GB bands, the edistats report is shown below.

```
EDI Image: /mnt/smr/edi.img

Unshingled Partition: /mnt/smr/unshingled

Report Time: Tue May 29 06:06:49 2012


Sector Size: 4096 bytes

Bands: 100 bands

Band Size: 262144 blocks (1073741824 bytes)
```

Band No.	State	Band Size	Cursor Position	Live Blocks	Dead Blocks
Band 1	Used	262144	250216	160260	89938
Band 2	Free	262144	0	0	0
Band 3	Free	262144	0	0	0
Band 4	Free	262144	0	0	0
Band 5	Free	262144	0	0	0
Band 6	Free	262144	0	0	0
Band 7	Free	262144	0	0	0
Band 8	Free	262144	0	0	0
Band 9	Free	262144	0	0	0
Band 10	Free	262144	0	0	0

```
(90 more...)
```

This is the EDI at one of the 5 data nodes. Using the above, we can calculate that the space overhead of TeraSort on 1GB of data at each node will be:

= (Live Data Blocks + Dead Data Blocks) / Live Data Blocks

= (160260 + 89938) / 160260) = 1.56

It should be noted that this is the overhead in terms of SMR disk blocks; not actual data. Application data may be quite fragmented within this space. In other words, the application data may not line up with the block boundaries, leading to wastage of some space in the last block for each file.

A 1GB TeraSort run would imply 2GB of data being stored in the SMR device: 1GB from TeraGen and 1GB from TeraSort. Assuming that the data is evenly divided between the datanodes, this implies roughly 400MB of data per node. From the edistats report, it is clear that the amount of live data blocks accounts for nearly 626MB, including all the additional data that HDFS would be storing for internal use, and space wasted due to internal fragmentation.

Again, from the edistats report, the amount of dead data blocks accounts for 351.33MB. During the execution of the workload, TeraSort reported around 300MB of data being written to the file system (this refers to non-HDFS data). Again, we have some room for internal fragmentation as the intermediate file sizes may not line up with the block boundaries.

Based on these figures, we can estimate the amount of cleaning that needs to be done. For each 1GB of TeraSort, we would use 1.56GB of disk space that would need to be cleaned at some point. The total run-time of this experiment was 150.27 seconds, in which 0.95GB of data was written to the disk. Thus the total time taken to fill 70% of this disk would be:

= ((70 / 100) * 100) * (150.27 / .95)

= 70 * 158.18

= 11072.6 seconds or 3.08 hours!

Amount of dead data in 100GB = 100 * 0.56 / 1.56 = 35.90GB

Thus, if the Garbage Collector can clean up around 3.32MB of dead data per second (35.90 / 11072.6), then the SMR device will not have any space wasted in dead data blocks! While this is merely theoretical conjecture at this point, the numbers seem quite manageable for a real Garbage Collection implementation, even if there are overheads and other issues that we might have overlooked.

We hope that the data presented here will serve future research on SMR Garbage Collection mechanisms and the future authors of these mechanisms can use these numbers as the basic requirements for garbage collection performance.

8.2 The emulator API

The SMR Emulator currently only implements a subset of the T13 Specification for SMR/Banded devices. This is the bare minimum needed to run applications on an SMR device. However there are a couple of avenues to explore with the SMR Emulator and its API.

8.2.1 Disk zoning

One of the options we have not considered yet is the issue of zoning on disks. Like traditional HDDs, SMR devices will also be zoned, and might warrant emulation. In future work, we may consider adding Zoning capabilities to the SMR Emulator and tie it in with the `edi_modesense()` function, allowing the host to query the SMR Emulator for zoning information as well.

8.2.2 Variable band sizes

For our experiments, we have emulated an SMR device that has bands of a fixed size. It is conceivable to have SMR devices where the sizes of bands may vary based on the zone they are in. This implies that ShingledFS should also consider band sizes in its data placement policy.

8.2.3 Band linking

There is a set of options in the T13 specification that allow for linking of physical bands on the SMR device to form a larger logical band. This allows the host to read and write past the end of the physical band, into the physical band linked to it. Future research can implement this feature in the SMR Emulator and study its utility and implications.

8.3 ShingledFS improvements

In its present form, ShingledFS v2 is robust enough to be used for evaluating Big Data workloads. It even works well with demanding workloads like the Linux Kernel compilation. However, there is still a lot of room for improvement.

8.3.1 Block on `write()`

One of the off-shoots of an attempt at keeping the implementation simple is block-on-`open()`. Consider a case where a file has been written to and then closed. This file is handed off to the Worker thread for storing to the SMR Emulator. But then, the FUSE thread re-opens the file. In its present form, the FUSE thread blocks on `open()`, waiting for the Worker thread to finish writing the file to the SMR Emulator, even if the file is being opened only for reading. This behavior needs to be modified to block-on-`write()`, allowing the FUSE thread to proceed until it attempts to write to the file. At that time, if the file has been written to the SMR Emulator by the Worker thread, then the `write()` can proceed normally. Otherwise, the FUSE thread will block waiting for the Worker thread to finish.

8.3.2 Attribute caching

To counter the additional calls to `getattr()` injected by FUSE, it is imperative to cache attributes of files being operated on in memory so that the overheads of multiple system calls to retrieve these values can be avoided. This also means adding code to ensure cache coherency, making the task fairly tricky.

8.3.3 Better data structures and algorithms

Yet another artifact of the simple implementation is the use of inefficient data structures such as linked-lists and string-based comparisons. These need to be replaced with hash tables and faster comparisons based on inode numbers instead of file paths.

9 Conclusion

Shingled Magnetic Recording is a novel approach to improving the data density of HDDs and is a useful add-on to Perpendicular Magnetic Recording. However, its adoption depends on coordinated efforts from device manufacturers and systems software vendors.

We present two pieces of software; a SMR Emulator and ShingledFS, the second-generation SMR-aware FUSE-based file system that provide a test bed to evaluate the conduciveness of SMR devices to Big Data workloads. We have shown the design and evolution of the SMR Emulator and ShingledFS over the past year, highlighting some of our key findings, their implementation and their impact on evaluation workloads.

Our evaluations show that SMR is definitely conducive to Big Data workloads and can be used to deliver high-capacity low-cost storage that does not exact a high performance penalty.

Furthermore, we discuss some of the challenges involved with garbage collection and present a few ideas that we hope will serve future research in this field.

10 References

[1] C. Walter, "Kryder's Law," 25 Jul 2005. [Online]. Available: http://www.scientificamerican.com/article.cfm?id=kryders-law. [Accessed 22 May 2012].

[2] Wikipedia, "IBM RAMAC 305," 12 Feb 2012. [Online]. Available: http://en.wikipedia.org/wiki/IBM_305_RAMAC. [Accessed 18 May 2012].

[3] Amazon.com, "Seagate Constellation ES.2 ST33000650NS 3 TB Internal Hard Drive," [Online]. Available: http://www.amazon.com/Constellation-ES-2-ST33000650NS-Internal-

Drive/dp/B004YQXZ1M/ref=sr_1_1?s=electronics&ie=UTF8&qid=1312137256&sr=1-1. [Accessed 25 May 2012].

[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 G. Ganger, "Principles of Operation for Shingled Disk Devices," Technical Report CMU-PDL-11-107, Parallel Data Lab, Carnegie Mellon University, Pittsburgh, PA, 2011.

[6] R. Wood, "The Feasibility of Magnetic Recording at 1 Terabit per Square Inch," *IEEE Transactions on Magnetics,* vol. 36, no. 1, Jan 2000.

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

[8] M. H. Kryder, E. C. Gage, T. C. McDaniel, W. A. Challener, R. E. Rottmayer, G. Ju, Y.-T. Hsia and M. F. Erden, "Heat Assisted Magnetic Recording," *Proceedings of the IEEE,* vol. 96, no. 11, pp. 1810-1835, Nov 2008.

[9] E. Dobisz, Z. Bandic, T. Wu and T. Albrecht, "Patterned media: nanofabrication challenges of future disk drives," *Proceedings of the IEEE,* vol. 96, no. 11, pp. 1836-1846, Nov 2008.

[10] Y. Shiroishi, K. Fukuda, I. Tagawa, H. Iwasaki, S. Takenoiri, H. Tanaka, H. Mutoh and N. Yoshikawa, "Future Options for HDD Storage," *IEEE Transactions on Magnetics,* vol. 45, no. 10, pp. 3816-3822, Oct 2009.

[11] M. Geenen and I. TRENDFOCUS, "FOCUS ON: Patterned Media's Economics, Challenges, and Overwhelming Potential," *FOCUS ON:,* 20 Jan 2009.

[12] G. Gibson and M. Polte, "Directions for Shingled-Write and Two-Dimensional Magnetic Recording System Architectures: Synergies with Solid-State Disks," Technical Report CMU-PDL-09-104, Parallel Data Lab, Carnegie Mellon University, Pittsburgh PA, 2009.

[13] S. Greaves, Y. Kanai and H. Muraoka, "Shingled Recording for 2-3 Tbit/in sq.," *IEEE Transactions on Magnetics,* vol. 45, no. 10, Oct 2009.

[14] R. Wood, M. Williams and A. Kavicic, "The Feasibility of Magnetic Recording at 10 Terabits Per Square Inch on Conventional Media," *IEEE Transactions on Magnetics,* vol. 45, no. 2, Feb 2009.

[15] A. Amer, D. D. Long, E. L. Miller, J.-F. Paris and T. S. Schwarz, "Design Issues for a Shingled Write Disk System," in *26th IEEE (MSSR2010) Symposium on Massive Storage Systems and Technologies*, Lake Tahoe, NV, 2010.

[16] Y. Cassuto, M. A. Sanvido, C. Guyot, D. R. Hall and Z. Z. Bandic, "Indirection Systems for Shingled-Recording Disk Drives," in *26th IEEE (MSST2010) Symposium on Massive Storage Systems and Technologies*, Lake Tahoe, NV, 2010.

[17] T13 Technical Cmte., 02 Aug 2009. [Online]. Available: www.t13.org/documents/uploadeddocuments/docs2009/d2015r2-ATAATAPI_Command_set_-_2_ACS-2.pdf.

[18] S. J. Greaves, H. Muraoka and Y. Kanai, "Simulations of recording media for 1 Tb/in2," *Journal of Magnetism and Magnetic Materials,* vol. 320, pp. 2889-2893, 2008.

[19] L. Cassioli, "Perpendicular Magnetic Recording (Image)," 21 Jun 2005. [Online]. Available: http://en.wikipedia.org/wiki/File:Perpendicular_Recording_Diagram.svg. [Accessed 22 May 2012].

[20] P. Kasiraj, R. M. New, J. C. de Souza and M. L. Williams, "System and Method for Writing Data to Dedicated Bands of a Hard Disk Drive". USA Patent US7490212, 10 Feb 2009.

[21] M. Rosenblum and J. K. Ousterhout, "The Design and Implementation of a Log-Structured File System," *ACM Transactions on Computer Systems,* vol. 10, no. 1, pp. 26-52, Feb 1992.

[22] "File system in USEr space," [Online]. Available: http://fuse.sourceforge.net. [Accessed 25 May 2012].

[23] "Programming Project 2: Storage Management in a Hybrid SSD/HDD File System," 14 Mar 2011. [Online]. Available: http://www.ece.cmu.edu/~ganger/746.spring11/projects/proj2_fuse/18746-s11-proj2.pdf. [Accessed 25 May 2012].

[24] Carnegie Mellon University, "15-746/18-746: Advanced Storage Systems," 2011. [Online]. Available: http://www.ece.cmu.edu/~ganger/746.spring11/. [Accessed 25 May 2012].

[25] O. O'Malley and A. C. Murthy, "Winning a 60 Second Dash with a Yellow Elephant," Technical Report, Yahoo! Inc., 2009.

[26] M. Polte, J. Simsa and G. Gibson, "Comparing Performance of Solid State Devices & Mechanical Disks," in *3rd Petascale Data Storage Workshop (PDSW)*, 2008.

[27] "PRObE: Parallel Reconfigurable Observarional Environment," 09 Sep 2011. [Online]. Available: http://marmot.pdl.cmu.edu. [Accessed 25 May 2012].

[28] R.W. Cross and M. Montemorra, "Drive-based recording analyses at >800 Gfc/in^2 using shingled recording," *Selected papers of the 9th Perpendicular Magnetic Recording Conference (PMRC 2010), Journal of Magnetism and Magnetic Materials*, vol. 324, no. 3, pp. 330-335 Feb 2012.