The Applications of Workload Characterization in The World of Massive Data Storage

A THESIS
SUBMITTED TO THE FACULTY OF THE GRADUATE SCHOOL
OF THE UNIVERSITY OF MINNESOTA
BY

Weiping He

IN PARTIAL FULFILLMENT OF THE REQUIREMENTS
FOR THE DEGREE OF
Doctor of Philosophy

David H.C. Du

August, 2015

Acknowledgements

There are many people that have earned my gratitude for their contribution in my journey to earning my Ph.D.. First and foremost, I would like to express my deepest gratitude to my advisor Prof. David Hung-Chang Du for his training, guideline and encouragement. I am extremely grateful to him for not only training me to be a better researcher but also leading me to be a better person. I could not have focused on my research and pushed the limit of my ability without his thought-provoking words.

I would like to thank my committee members, David Lilja, Abhishek Chandra, Anand Tripathi and Gopalan Nadathur for their constructive advice and previous time. I learned that persistence and patience are two important merits of being a good graduate student.

I was lucky enough to have the opportunities to work with many smart and lovely people in CRIS. I would like to thank Cory Devor, Chung-I Lin, Alireza Haghdoost, Weijun Xiao, Guanlin Lu, Nohhyun Park, Dongchul Park, Youngjin Nam, Shanshan Li, Peng Li, Anna Kryzhnyaya, Muthukumar Murugan, Joe Naps, Xiang Cao, Ziqi Fan, Xiongzi Ge, Zhichao Cao, Meng Zou, Fenggang Wu, Hao Wen, Keerthi Palanivel, Rohan Pasalkar, Brandon Hoffmann, Manas Minglani, Bingzhe Li, Jeremy Kieser, Jingwei Ma, Shravya Rukmannagari and Kewal Panchputre for their engaging and inspiring discussions. The numerous meetings we had together are the basis of my graduate life. Special thanks go to Chung-I Lin and Alireza Haghdoost for motivating me to improve myself and develop more friendship.

I also would like to thank Sai Narasimhamurthy and Giuseppe Congiu of Seagate Technology for being a constant source of great thoughts. I could not have explored so much in the field of high performance computing and parallel I/O without the stimulating weekly meetings. I'm grateful for their invaluable time and efforts.

I would like to thank Jerry Fredin of NetApp for being my mentor for the block I/O workload replayer project and stan skelton of NetApp for offering me all kinds of help during my internship. I also would like to thank Jim Rohde, Dan Oelke, Mike Klemm, Jim Kmiec and Mark Bakke of Dell Compellent for their encouragement, advice and help during my internship. The thinking and development skills contribute a lot in obtaining my Ph.D..

Finally, I would like to thank NSF and CRIS sponsor companies for funding my projects as well as Minnesota Supercomputing Institute for providing access to their research facilities and offering timely support.

Dedication

I dedicate this dissertation to those who held me up over the years, especially

To my parents, Qijin He and Yumei Ma, who made every possible effort that I can never image to support me and offload any distraction. Their encouragement and faith in me stimulated me to strive for my success. They are proud of me for every tiny success I made and I am so grateful for that.

To my wife, Xin Wan, who has accompanies me through all the ups and downs, laughs and tears during my journey to my Ph.D.. She has done an incredible job motivating and cheering up me whenever I got frustrated. I feel so grateful for having her as my wife and friend.

To my sister, Aiping He, who has encouraged me all along and who has undertaken my part of responsibilities to take care of our parents. I am as much proud of her as she is proud of me. I also would like to thank my brother-in-law, Zhigang Fang, for being a great husband and father.

To my parents-in-law, Dacheng Wan and Shulan Zhao, for their constant encouragement, cares and trust.

Abstract

The digital world is expanding exponentially because of the growth of various applications in domains including scientific fields, enterprise environment and internet services. Importantly, these applications have drastically different storage requirements including parallel I/O performance and storage capacity.

Various technologies have been developed in order to better satisfy different storage requirements. I/O middleware software, parallel file systems and storage arrays are developed to improve I/O performance by increasing I/O parallelism at different levels. New storage media and data recording technologies such as shingled magnetic recording (SMR) are also developed to increase the storage capacity. This work focuses on improving existing technologies and designing new schemes based on I/O workload characterizations in corresponding storage environments.

The contributions of this work can be summarized into four pieces, two on improving parallel I/O performance and two on increasing storage capacity. First, we design a comprehensive parallel I/O workload characterization and generation framework (called PIONEER) which can be used to synthesize a particular parallel I/O workload with desired I/O characteristics or precisely emulate a High Performance Computing (HPC) application of interest. Second, we propose a non-intrusive I/O middleware (called IO-Engine) to automatically improve a given parallel I/O workload in Lustre which is a widely used HPC or parallel I/O system. IO-Engine can explore the correlations between different software layers in the deep I/O path, as well as workload patterns at runtime to transparently transform the workload patterns and tune related I/O parameters in the system. Third, we design several novel static address mapping schemes for shingled write disks (SWDs) to minimize the write amplification overhead in hard drives adopting SMR technology. Fourth, we propose a track-level shingled translation layer (T-STL) for SWDs with hybrid update strategy (in-place update plus out-of-place update). T-STL uses dynamic address mapping scheme and performs garbage collection operations by migrating selected disk tracks. This scheme can provider larger storage capacity and better overall performance with the same effective storage percentages when compared to the static address mapping schemes.

Contents

List of Tables

List of Figures

Chapter 1

Introduction

Traditionally, High Performance Computing (HPC) applications and big data applications ran in different types of systems. HPC applications, due to their computational intensive and/or I/O intensive nature, usually run in HPC systems with powerful computing capability and high I/O bandwidth. Message Passing Interface (MPI), the defacto standard of distributed computing, is used for inter-process communication in these large HPC systems, which also defines a subset of programming interfaces to parallelize the concurrent I/O accesses to shared files. Some HPC applications are so I/O intensive that data access has to be highly parallelized to achieve satisfactory I/O throughput. Therefore technologies including object storage based I/O servers and parallel file systems are developed to satisfy these applications. Big data applications, on the other hand, usually run on cluster systems consisting of commodity hardware. MapReduce [3] and Hadoop [4] platforms are developed to efficiently utilize these commodity hardware. These applications generally process a large volume of data sets.

However, HPC applications and big data applications share common general goals - to maximize the computing power utilization and minimize I/O overhead via optimal task and data distribution. This common objective, in addition to the fast advances in server technologies and the lower cost of HPC systems, is leading to the convergence of the two types of systems. One major form of convergence is to run big data applications or MapReduce applications on top of HPC systems by interfacing Hadoop with a parallel file system instead of the traditional Hadoop Distributed File System (HDFS)

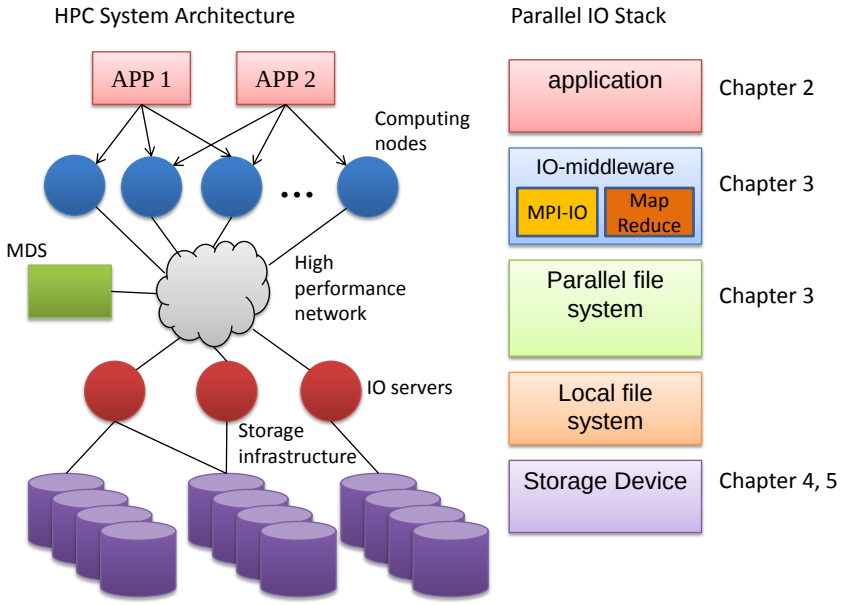


Figure 1.1: Massive Storage System Echo-system

[5]. As a result, there will be a mix of HPC application workloads and big data application workloads running on the same set of computing nodes. Figure 1.1 shows the overall hardware architecture and I/O software stack of a typical converged HPC system. The hardware components include computing nodes, high performance network, Metadata Server and I/O servers for parallel file systems or distributed file systems, as well as backend storage infrastructure. The corresponding I/O software layers include the application layer, I/O-middleware layer, parallel file system, local file system and storage device drivers. This type of converged HPC system are becoming more and more popular, especially in science fields [5, 6, 7].

There are two major objectives in this work as our efforts on improving the converged HPC ecosystem. First, we aim to improve parallel I/O performance by characterizing applications' I/O workloads and making optimizations in the application layer, I/O-middleware and parallel file system layer. Second, we target increasing the backend storage capacity by introducing a new type of hard drives using Shingled Magnetic Recording (SMR) technology [8]. The data management schemes in these new hard drives also take into account of I/O workload characteristics.

1.1 I/O in High Performance Computing

Scientific applications from various fields including climate studies, molecular dynamics, earthquake predictions and genomic engineering usually run on HPC systems. Current HPC systems have several tens to hundreds of thousands of processor cores which is expected to increase into the future with the technological advances. Researchers are already preparing for exascale systems with millions of cores [9]. As a result, I/O performance and data parallelism are becoming more challenging. Understanding application I/O workloads and system I/O capabilities is important for optimizing I/O performance and designing new I/O systems. To achieve these goals, tools that can efficiently characterize workloads and generate realistic synthetic parallel I/O workloads are often needed.

There are generally three ways of generating parallel I/O workloads. The first approach is to use existing parallel I/O benchmarks such as IOR2, MPI-IO Test, FLASH-IO and NPB benchmark suite. Although most of these benchmarks provide a set of tunable parameters, they can only produce simple I/O patterns which are often unable to truly represent the fast evolving HPC I/O workloads. The second approach is replaying I/O traces captured from existing HPC applications. This approach can produce precise HPC I/O workloads of the target applications but is inflexible. Moreover, capturing I/O traces can be difficult in certain production environments. The third approach is to generate synthetic parallel I/O workloads which is essentially a balance between the prior two approaches. Synthetic workloads are generated based on a specified workload model and many I/O characteristics can be tuned using corresponding parameters. The synthetic workload generator introduced in this work is such a tool.

On the other hand, parallel I/O or HPC I/O is very complex because of the deep I/O stack and various I/O related system parameters along the I/O path. In order to fully exploit the parallel I/O performance, many factors including the I/O access pattern, I/O access mode, file allocation scheme and I/O parameters, as well as their correlations must be investigated. For example, files in parallel file systems are usually striped over multiple I/O servers with a specified striping width and stripe size. Significant inter-process overhead or locking overhead on the stripes can be incurred if the requests issued by application processes are not coordinated well. Different striping policies can

also have diverse read and write performance impacts. In this work, we thoroughly investigate the existing parallel I/O stack via instrumenting the MPI-IO library and conducting designed experiments. We also propose a comprehensive parallel I/O model to discover the relationship between logical I/O access pattern and physical data layout. Subsequently, an automatic parallel I/O optimization tool called "IO-Engine" is designed to transparently improve the I/O performance by dynamically transforming incoming workloads and tuning I/O parameters.

1.2 Storage Systems with Large Capacity

Data generated by all kinds of internet service, industrial applications and various institute is increasing exponentially. 5 exabytes (10^{18} bytes) of data were created by human by 2003, which can be easily generated in two days today. In 2012, the data volume of the digital world reached 2.72 zettabytes (10^{21} bytes), which is predicted to double every two years reaching 8 zettabytes by 2015 [1]. Infographic, a social media, created a chart to show how much data was generated every minute in 2014 [2]. According to the chart, 72 hours of new video was uploaded to Youtube, 204 millions emails were sent, 2.5 millions pieces of messages were shared on Facebook and 216,000 new photos were posted on Instagram, etc. These data has eventually to be stored on physical storage devices.

In order to keep up with the pace of data generation, storage media capacity has been growing over the year. Recently storage devices especially traditional hard disk drives start to reach the maximum areal data density that the current perpendicular recording can achieve. Instead of disk capacity doubling every 18-24 months as the disk industry did in the 2000s, it is now only offering approximately 20% capacity growth per year recently. Therefore different magnetic recording designs have been proposed including Heat-assisted magnetic recording (HAMR) [10], Bit-Patterned Media Recording (BPMR) [11, 12] and Shingled Magnetic Recording (SMR) [8, 13, 14].

Among these new techniques, SMR is the most promising because it does not require significant changes to the current manufacturing techniques. It increases data capacity by overlapping the adjacent tracks and thus packing more data tracks into platters with the same physical dimensions. The asymmetric requirements for head width of

read and write requests make shingling technically feasible. Disk heads write a wide track but only need a narrow track for reading. Thus SMR works by writing a wide track then overwriting most of it when performing another write. The downside of this technique is that random write to a particular track may overwrite the valid data on the following tracks because data tracks are shingled. Hard drives using SMR techniques are called Shingled Write Disks (SWDs) or SMR drives. The SWDs can only be applied for cold write workloads or write-once-read-multiple-times workloads without addressing the overhead incurred by random writes.

Two main physical layouts are being explored to address this problem which are in-place update SWD (I-SWD) and out-of-place SWD (O-SWD). An In-place update performs as follows. Assuming the updated data resides on a specific track, the data on the following tracks will first be read out and written back to their original positions later after the desired data has been written or updated. As a result, a single update operation is amplified to several read and write operations. The write amplification overhead generally increases with the number of following tracks that are affected.

An out-of-place update operation performs in a copy-on-write manner. The updated data or the new data will be written to a new position and the old data will be invalidated. An address mapping scheme must be used to keep track of the data movement and a garbage collection scheme must be designed to reclaim the invalidated space later, which are essentially other forms of write amplification overhead.

The main challenge of designing a shingled write disk (SWD) is the balance between write amplification overhead minimization, space gain and overall I/O performance. According to system level that handles the write amplification, address mapping and space management, SWDs can also be classified into drive-managed SWDs, host-aware SWDs and host-managed SWDs. As the names suggest, drive-managed SWDs encapsulate all these functions inside the drives themselves and provide transparent block interface to the upper levels including file systems and applications. While host-aware SWDs and host-managed SWDs offload these functions to the host machines. To accomplish this, the internal track layout must be reported to the host operating system via a set of new commands newly defined in the T10 industrial standard [15]. The T10 standard also specifies that the tracks are grouped into zones of size 256 MB and that there are three types of zones: conventional zone, sequential write preferred zone and sequential

write required zone. Conventional zone is optional for both host-aware SWDs and host-managed SWDs. Sequential write preferred zone is exclusive to host-aware SWDs and sequential write required zone is exclusive to host-managed SWDs.

This work focuses on drive-managed SWDs because they can be used in existing storage systems in a drop-in manner which requires no modification. For example, the two SWDs on the market today, Seagate Archive HDD (8 TB) [16] and Western Digital Ultrastar Archive Ha10 (10 TB) [17], are all autonomous drives. However, both of them are targeted for only cold workloads and archive workloads due to the unresolved write amplification problem. Therefore, in this work, we propose two drive-managed SWD designs, one based on the in-place update method and the other based on a combination of in-place update and out-of-place update methods, in order to make SWDs that can handle primary workloads instead of only cold workloads.

1.3 Contributions

The contributions of this work can be summarized into four pieces, two on improving parallel I/O performance and two on increasing storage capacity. First, we design a comprehensive parallel I/O workload characterization and generation framework (called PIONEER) which can be used to synthesize a particular parallel I/O workload with desired I/O characteristics and also precisely emulate an HPC application of interest. Second, we propose a non-intrusive I/O middleware (called IO-Engine) to automatically improve a given parallel I/O workload in Lustre system. IO-Engine can explore the correlations between different software layers in the deep I/O path, as well as workload patterns at runtime to transparently transform the workload patterns and tune related I/O parameters in the system. IIO-Engine can also be extended to support other parallel file systems. Third, we design several novel static address mapping schemes for SWDs to minimize the write amplification overhead in hard drives adopting SMR technology. Fourth, we propose a track-level shingled translation layer (T-STL) for SWDs with hybrid update strategy (in-place update plus out-of-place update). T-STL uses a dynamic address mapping scheme and performs garbage collection operations by migrating selected disk tracks. This scheme can provide larger storage capacity and better overall I/O performance under the same effective capacity percentages when compared to the

static address mapping schemes.

1.4 Organization

The rest of this work is organized as follows. Chapter 2 describes a complete solution to parallel I/O workload characterization and synthesizing. Thoroughly understanding parallel I/O workloads is the basis for optimizing existing storage systems and designing new storage systems. Our solution not only introduces an efficient workload characterization framework but also contains a tool to synthetically generate parallel I/O workloads. Based on this knowledge, Chapter 3 further discusses parallel I/O optimizations that can be achieved with more in-depth I/O workload characterization. After discussing ways to characterize parallel I/O workloads and optimize parallel I/O performance, Chapter 4 and 5 shift the gear and focus on the other storage requirement in converged HPC systems - storage capacity. Conventional hard disk drives comprise a significant percentage (up to 80%) of the total storage capacity in today's massive storage system but are hitting their areal data density limit. SWDs using shingled magnetic recording will be the optimal near-term solution. Chapter 4 describes several novel static address mapping schemes for in-place update SWDs and Chapter 5 proposes a track-level translation layer for out-of-place update SWDs. Finally, Chapter 6 makes some conclusions and states future research directions.

Chapter 2

Parallel I/O Characterizations and Generation

The demand for parallel I/O performance continues to grow. However, modeling and generating parallel I/O workloads are challenging for several reasons including the large number of processes, I/O request dependencies and workload scalability. In this chapter, we propose the **PIONEER**, a complete solution to Parallel I/O workload characterizatioN and gEnERation. The core of PIONEER is a proposed *generic workload path*, which is essentially an abstract and dense representation of the parallel I/O patterns for all processes in a High Performance Computing (HPC) application. The generic workload path can be built via exploring the inter-processes correlations, I/O dependencies as well as file open session properties. We demonstrate the effectiveness of PIONEER by faithfully generating synthetic workloads for two popular HPC benchmarks and one real HPC application.

2.1 Introduction

The computing scale is currently expanding from Petascale to Exascale. This will make the data parallelism even more challenging. Thoroughly understanding parallel I/O workloads is therefore critical for designing storage systems and improving their performance. Synthetic parallel I/O workload generation tools are also greatly needed in storage system performance tuning, testing and measurement.

Table 2.1: Trace Snippet

```
10:48:52.404754 MPI_File_open(92, 0x807a7f8, 34, 0x8078e38, 0xbf83b9e0 <unfin-
ished ...>
10:48:52.405470 SYS_statfs64(0x807a7f8, 84, 0xbf83b788, 0xbf83b788, 0x81dff4) =
0 <0.008302>
10:48:52.414801 SYS_umask(022) = 077 <0.000025>
10:48:52.414866 SYS_umask(077) = 022 <0.000017>
10:48:52.414936 SYS_open("/panfs/caddypan.lanl.gov/scratch1/
nobody/tests/OUTPUT.1206553581.0", 32768, 00) = 36 <0.000301>
10:48:52.416760 <... MPI_File_open resumed> ) = 0 <0.011931>
10:48:52.416807 MPI_Wtime(0xf6000000, 0x41b419c7, 0x1dc50cea, 0x4067346f, 0
<unfinished ...>
10:48:52.417341 <... MPI_Wtime resumed> ) = 0x6490a356 <0.000416>
10:48:52.417419 MPI_Wtime(0x75a0a0, 0x8053cc3, 0x45fe1127, 0x4041e0d0,
0x81f8b8 <unfinished ...>
10:48:52.417922 <... MPI_Wtime resumed> ) = 0x2f661f1e <0.000401>
10:48:52.417986 MPI_File_seek(0x807a8a0, 0, 0, 600, 0x406734d6) = 0 <0.000195>
10:48:52.418364 MPI_File_iread(0x807a8a0, 0xa7d19008, 458752, 1, 0xbf83b898
<unfinished ...>
10:48:52.418681 SYS_read(36, "EDITED"..., 458752) = 458752 <0.029523>
10:48:52.449365 <... MPI_File_iread resumed> ) = 0 <0.030849>
```

Many HPC benchmarks such as IOR2 [18], NPB [19], and FLASH-IO [20] are developed to help test system performance. HPC benchmarks are usually easy to use and can be tweaked by the users. However, real HPC applications in many scientific domains keep emerging such that system designers often have a hard time to find a benchmark which can represent a particular I/O workload for their need. Furthermore, many HPC benchmarks have little control on certain IO dimensions such as the IO arrival pattern [21, 22].

On the other hand, synthetic parallel I/O workload generation based on existing traces is more promising and practical as long as I/O tracing tools are enabled in a production environment. There is no need to access the source code of a real HPC application. Besides, the characteristics of the existing traces can be modified and tuned to generate a desired workload pattern.

However, parallel I/O workload modeling and synthesizing are very challenging. Raw

traces captured by tools like *LANL-Trace* framework tool [23] have to be sanitized before workload characterizing and modeling. For example, Table 2.1 shows a parallel I/O trace snippet that is generated by a HPC benchmark application called MPI-IO Test [24] which contains several <unfinished> and <resumed> tag pairs indicating the start and completion of a particular I/O request. Each sanitized I/O record contains several important fields including timestamp, request type, request argument list, execution time, etc. A comprehensive workload modeling and generation framework should consider all of these important factors.

A parallel I/O trace usually contains both POSIX-IO operations and MPI-IO operations, which may have quite different syntaxes and arguments. For example, `SYS_open` requires 3 arguments but `MPI_File_iread` needs 5 arguments. However, disk I/O workloads usually only deal with two operation types (READ and WRITE) with unified syntax and argument dimensions. Furthermore, there are all kinds of request dependencies in both the POSIX-IO library and the MPI-IO library. For example, `MPI_File_iread` depends on `MPI_File_open` since no process will be able to access the file data without opening it first. The challenges of modeling and generating synthetic parallel I/O workloads can be easily recognized when one realizes that actual parallel I/O workloads are generated by hundreds or thousands of these processes and these processes are correlated in specific ways.

Many prior studies such as [25, 9, 26, 27] have been done on parallel I/O characterization to acquire meaningful characteristics to unveil application behaviors and provide valuable insights for parallel I/O workload synthesizing. However, as far as we know, there are rarely any complete parallel I/O workload synthesizing solutions that generate realistic parallel I/O workloads. Most of the existing parallel I/O modeling and synthesizing studies focus on a single dimension such as inter-arrival time [21, 22] or request offset [28]. The existing two-dimensional [29] or multi-dimensional characterization schemes [30] for block I/O workloads cannot be applied to parallel I/O workload modeling directly because they do not consider the uniqueness of parallel I/O workloads.

Our solution in this chapter handles these uniqueness and challenges with effective approaches. We propose the concept of a "*generic workload path*" based on the inter-process correlations to abstract and present the I/O patterns of all processes; we also set

library enforcement rules to deal with the I/O library complexities and request dependencies; we develop the *file open session framework* to describe the file access patterns in the generic workload path; we characterize all the I/O operations that appeared in the generic workload path, the outcome of which can be tuned to generate a corresponding *synthetic generic workload path*; and we also develop a *workload generation engine* to expand a synthetic generic workload path into a complete parallel I/O workload, which can be scaled with any desired number of processes. Details about these approaches will be presented in Section V. These approaches together enable us to take the initial step to propose a robust and scalable solution in this chapter. As demonstrated later, our solution can significantly reduce the overhead of workload tracing, characterization and generation.

The structure of this chapter is as follows. We introduce some background and related work in Section 2.2 and Section 2.3. In Section 2.4, we discuss the uniqueness and characteristics of parallel I/O workloads and consequent challenges, followed by corresponding approaches in Section 2.5. We then propose our comprehensive solution of parallel I/O characterization and synthesizing in Section 2.6. We demonstrate the effectiveness of our solution with experiments in Section 2.7. We finally make some conclusions in Section 2.8.

2.2 Background

In this section, we introduce the assumed HPC environment and the software applications that generate the parallel I/O workloads. We will also summarize the existing work and show that major gaps exist in characterizing parallel I/O workloads and in generating synthetic parallel I/O workloads.

2.2.1 Parallel I/O Workloads

The concept of "parallelism" exists in many layers of the HPC software stack, such as application, I/O library, and file system. Therefore it is important to define parallel I/O workload carefully. In the scope of this chapter, we define parallel /IO as I/O workload generated by HPC applications that use MPI-IO library or higher level I/O libraries built on top of MPI-IO library. These I/O workloads may also contain POSIX-I/O requests.

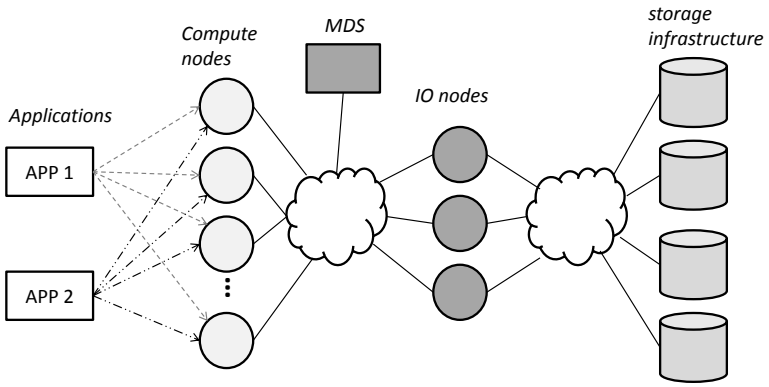


Figure 2.1: Parallel I/O Environment Abstraction

2.2.2 Assumed HPC Environment

A typical HPC system usually consists of several major components including computing nodes, I/O nodes (I/O servers), metadata servers and back-end storage infrastructures as shown in Figure 3.1. A real system may have different implementations for each of these components. For example, Lustre systems implement the I/O nodes as Object Storage Servers (OSSs), which manage one or more Object Storage Targets (OSTs). Back-end storage infrastructures could be fulfilled by a certain number of Storage Area Networks (SANs), each of which can be dedicated to a single OST or shared by multiple OSTs. Metadata servers in these systems maintain and manage a unified logical namespace for parallel file systems.

2.2.3 Parallel I/O Software Applications

HPC applications typically execute hundreds or thousands of processes. These applications can be either computational intensive, I/O intensive or both. The I/O behaviors and access patterns of these HPC applications depend, to a large degree, on the way they access files and the I/O libraries. Different I/O libraries can be used such as POSIX-IO, MPI-IO, HDF5, netCDF, etc. In the scope of this work, we define parallel applications as those utilizing MPI-IO libraries and higher level I/O libraries built on top of MPI-IO libraries such as HDF5. The processes of these parallel applications, however, are also technically allowed to access files with POSIX-IO library. When a parallel application

runs, it will assign an *MPI process rank* to each process. The MPI process rank is essentially an internal process ID used by the parallel application to identify each process. The root process is assigned with rank 0.

In the HPC realm, there are generally three types of files: root exclusive files, shared files, and private files. A *shared file* is defined as a file that is shared by all participating processes, while a *private file* means that every process has its own version of this file, usually with customized file name. The naming convention for private files is a common prefix string plus the MPI process rank. A *root exclusive file* is only accessed by the root process.

2.3 Related Work

I/O workload characterization is important for understanding the workload and improving system performance. Different mathematical models were proposed to describe the workload patterns, either temporally or spatially. For example, Poisson process have been widely used to describe I/O arrival patterns until several prior studies indicated that statistical properties including autocorrelations and self-similarity [31] exist in these workloads due to their nature of burstiness [32, 33, 34]. Since then, various self-similarity oriented models have been proposed to emulate the burstiness feature. For example, FARIMA [33] and FBM [35] models were proposed to describe network traffic workloads and later applied to disk I/O workloads. Models that utilize multiple ON/OFF models [36] or a combination of ON/OFF model and Cox's model [37] also have been used to characterize the burstiness in storage systems. An I/O workload model based on the Alpha-stable process was also proposed to generate synthetic disk I/O workloads and parallel I/O workloads [21].

Although each of the above models has its own effectiveness in modeling the workload burstiness and temporal properties such as self-similarity, most of them are limited to only one dimension such as inter-arrival time or request offset. However, parallel I/O workloads contain important information in other dimensions as well. Therefore, models that can incorporate multiple dimensions become more preferred for workload characterization and generation. Wang *et al.* proposed a model that can model the spatio-temporal

correlation via an entropy plot of two-dimensional disk I/O request sequence [29]. Another work by Sriram and Kushagra utilizes the *probabilistic state transition diagram* [38] to describe disk I/O workloads in a comprehensive way where multiple dimensions including inter-arrival time, operation type, LBA offset, etc. are all considered to some degree. This work was then extended by Delimitrou in [30] to generate synthetic disk I/O workloads in data centers. However, both the two-dimensional and multi-dimensional characterization mechanisms are not suitable for parallel I/O workloads because they did not consider parallel I/O dependencies, let alone inter-process correlations.

On the other hand, there exist several studies on characterizing parallel I/O workloads. For example, Wang *et al.* used a series of empirical distributions to characterize parallel scientific applications in [25]. Distributions are presented independently though. Carns *et al.* developed the Darshan I/O characterization tool that can unveil some I/O behaviors of applications at extreme scale [9]. Carns *et al.* then outlined a methodology for continuous, and scalable I/O characterization that instruments Darshan and utilizes coarse-grained information from storage devices and file systems to help further interpret application level behaviors [26]. Cope *et al.* worked on the IOVIS project and proposed a portable I/O tracing system and visualization method to help analyze captured parallel I/O traces in an end-to-end manner [27]. However, most of these existing studies did not address the parallel I/O generation problem.

As a result, in this chapter we take the initial step of proposing a complete solution to parallel I/O characterization and synthesizing. We will demonstrate later in the chapter that this solution is robust and scalable.

2.4 Characteristics of Parallel I/O Workloads

In this section, we describe the uniqueness and associated challenges of modeling parallel I/O workload characterization and synthesizing.

2.4.1 Inter-Process Correlations

HPC applications usually execute hundreds or thousands of processes, with one root process and a bunch of child processes. When characterizing the MPI-IO Test traces that are published by LANL and IOR2 traces that we captured on a cluster system

at *Minnesota Supercomputing Institute* [39], we made some interesting observations on inter-process correlations. In general, the child MPI processes always mimic the root process. In other words, child processes share most of the I/O requests with the root process except those root exclusive I/O requests which are unique to the root process only. Table 2.2 shows the operational statistics of one root process and four child processes in a captured parallel I/O trace. This trace (identified as 32PE_N-N_448K) is a running instance of MPI-IO Test published by LANL. The table shows the statistics for selected major POSIX-IO and MPI-IO operations in the trace. The operational statistics of the child processes are nearly identical with each other. There is also a great similarity between the root process and child processes, except that the root process issues more POSIX-IO operations. This is mainly because the root process has to manage the parallel I/O environment. Besides, HPC application developers tend to choose the root process to perform temporary data management. This similarity can also be validated by profiling the *operation reuse distance*, which is defined as the number of operations between two successive appearances of same operation, for root process and all child processes. There may exist some HPC applications that their child processes are to be divided into several subgroups. The processes in each subgroup perform similar operations.

Besides, HPC applications usually make performance gains by utilizing advanced I/O features in MPI-IO and higher level I/O libraries such as collective I/O [40], when processes are accessing shared files. Shared files therefore tend to show stronger inter-process correlations than private files. Behind these advanced I/O operations, inter-process communications, data manipulation and synchronization are transparently managed by the I/O libraries.

2.4.2 Complexities of I/O Libraries

Due to the I/O software stack depth in the parallel environment, parallel I/O workloads generally involve more than one I/O library, which include, in most cases, both POSIX-IO and MPI-IO [26]. Computational science applications also typically use higher level I/O libraries such as HDF5 and parallel netCDF. The operations in these higher level I/O libraries will be eventually translated into MPI-IO and POSIX-IO operations. The

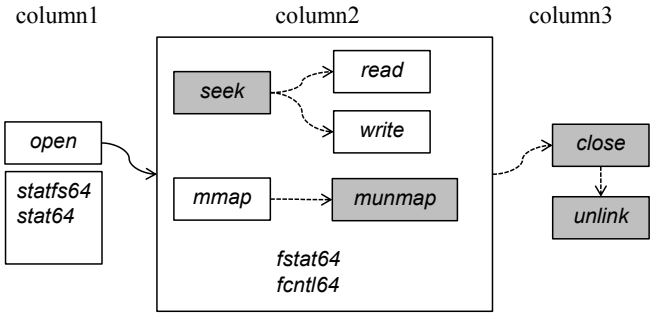
Table 2.2: Operation Statistics for 32PE_N-N_448K

Operation	child x1	child x2	child x3	child x4	root
SYS_open	195	195	195	195	257
SYS_close	82	79	76	77	126
SYS_read	10737	10309	10522	10306	11907
SYS_write	9905	9889	9891	9887	21995
SYS_fstat64	56	56	56	56	101
SYS_fcntl64	102	102	96	96	334
SYS_statfs64	4	4	4	4	5
MPI_File_seek	18724	18724	18724	18724	18724
MPI_File_iwrite	9362	9362	9362	9362	9362
MPI_File_iread	9362	9362	9362	9362	9362
MPIO_Wait	18724	18724	18724	18724	18724
MPI_File_close	2	2	2	2	2
MPI_Barrier	73	73	73	73	73
MPI_Wtime	56188	56188	56188	56188	56188

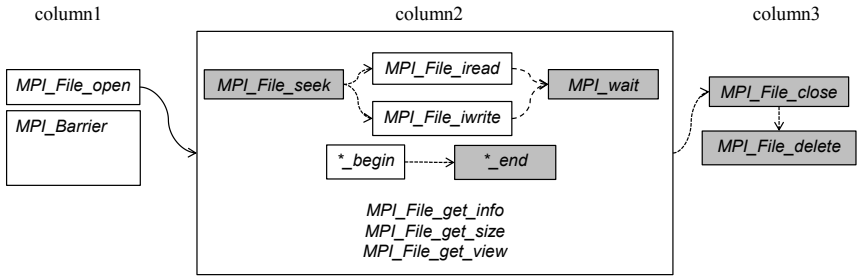
LANL-Trace framework tool [23, 41] can capture both POSIX-IO and MPI-IO commands.

Besides, POSIX-IO library and MPI-IO library have a rich set of I/O operations. They both contain metadata operations and file data operations. *Metadata operations* deal with file manipulations such as create, delete, open, close, file handler manipulation and file pointer manipulation. *File data operation*, on the other hand, do I/O transfer jobs for either read or write. Metadata operations are important in a parallel I/O environment. As explained in [42], the MDS servers play a critical role in large scale I/O since they may potentially become the performance bottleneck. As a result, it is important to characterize metadata operation patterns and the way they are related to file data operations as part of parallel I/O workload modeling.

Furthermore, unlike disk I/O operations, file system level I/O operations have to follow certain rules to be functional and meaningful. In this chapter, we classify these rules into two categories: *hard dependency* and *soft dependency*. Hard dependency has to be respected when generating synthetic file system I/O workloads and thus is mandatory. A simplest example for hard dependency will be that a process has to open a file first before it can read/write that file. Soft dependency, on the other hand, is optional, yet



(a) POSIX-IO



(b) MPI-IO

Figure 2.2: Enforcements for I/O Libraries

important for creating meaningful and high quality synthetic workloads. For instance, a process may need to seek to a specific offset inside an opened file in order to access a particular data section. Both POSIX-IO and MPI-IO libraries have to consider these dependencies.

2.4.3 File Access Pattern

File open session is one of the most important aspects because there are many important properties associated with it, including *sequentiality*, *ownership* and *access mode*. Sequentiality describes the file offset pattern and a file session is said to show high sequentiality when offsets generally present monotonically increasing or decreasing patterns. In regards to file ownership, a file can be shared by multiple processes or dedicated to a single process during a given open session. Access mode controls categories of file data operations that can be made to an opened file, such as read only, write only or both. Besides, there are some general characteristics of file open sessions, such as duration of file open sessions and frequencies of different I/O requests. Furthermore, metadata operation ratio and file data operation ratio are also important characterization criteria of file access patterns, considering the potential bottleneck at metadata servers. In conclusion, the access pattern of parallel I/O workloads can be summarized as the question: "which portion of which file is accessed by which process at what time in which way?" Our solution uses the framework of open session to answer this question and describe the parallel I/O patterns.

2.5 Approaches to Uniqueness

In this subsection, we describe the proposed approaches to addressing the challenges of the identified uniqueness.

2.5.1 Generic Workload Path

The large number of processes in an HPC application makes it impractical to characterize every single process separately although this is possible. Besides, even assuming that significant time and effort can be afforded for characterizing all the processes, challenges

for scaling the workloads will be faced. Our proposed generic workload path provides a more efficient and better way to characterize parallel I/O workloads.

Simple HPC applications may require only two processes, one root process and one child process, to create a generic workload path. In this chapter, we will consider this simple case. However, the same technique can be applied if child processes are partitioned into multiple subgroups. In other words, the generic workload path constructed with a very small number of processes is a dense representation of all processes. We then assign a *global sequence ID* to each of the operations in the generic workload path. The global sequence ID is the index number of a request to indicate the request order and position in the generic workload path or trace file. For example, the first request has global sequence ID 0.

There are generally three advantages creating a generic workload path. First of all, by merging the root process and selected child processes into a single generic workload path, we can characterize the correlation and mix patterns between their I/O requests with the help of global sequence IDs.

Secondly, generic workload path makes our solution highly scalable and robust because it works like a workload template which will be customized for each process based on their MPI process ranks when executed by our workload generation engine. Any desired number of processes can be used during the execution so we can scale the workload by varying the number of processes. For example, we can synthesize a parallel I/O workload of 1000 processes based on an original workload of 100 processes for scaling up, or synthesize a workload of 50 processes for scaling down. The approach of extracting, characterizing, synthesizing and executing generic workload path will be presented later in Section V.

Another great benefit of using generic workload path is that we may now only need to capture I/O traces for a small number of processes instead of all the processes, which significantly reduces the capturing overhead and resulting trace sizes. Besides, unnecessary characterization on most child processes can also be avoided.

We use *slack time* as timing control mechanism for the generic workload path. Slack time is defined as the time between the completion of a previous request and the start of the next request. We compute the slack time based on <timestamp> and <execution time>. The use of slack time enables a synthetic workload to be executed in storage

systems with different performance without overloading them.

2.5.2 I/O Library Enforcement

Unlike the block level I/O workloads, parallel I/O workload generation has to respect the inherent request dependencies of POSIX-IO library and MPI-IO library. As indicated previously, we consider two types of dependencies: hard dependency and soft dependency.

Figure 2.2a shows an enforcement diagram that represents these two types of request dependencies for selected POSIX-IO operations. Column 1 contains independent operations that do not require opening a file and are metadata operations. Column 2, on the other hand, contains dependent operations that operate on opened files, including both metadata operations and file data operations. The solid arrow means hard dependencies; the dashed arrow means soft dependencies. Operations in grey boxes can be unnecessary to be used together with their counterparts. For example, read/write may or may not be preceded by a seek operation in practice. Each open session begins with an open operation and ends with a close operation. For files accessed by MPI-IO operations, we have constructed a similar enforcement diagram in Figure 2.2b to represent the dependencies among selected MPI-IO operations.

In implementation, hard dependency is generally guaranteed by the framework of file open session described in Section IV. Dependent operations in Column 2 can only exist in corresponding file open sessions while independent operations are not limited by this constraint. Soft dependency is preserved by replacing certain operations with newly defined operations. For example, we replace `seek` and `read` operations with two new operations: `read0` and `read1`. `read0` indicates a `read` operation not preceded by a `seek` operation while `read1` indicates a preceded `read`. This is especially useful for MPI-IO library where more operation combinations are possible. For instance, the non-blocking `MPI_File_iread` can be either preceded by `MPI_File_seek`, followed by `MPIO_Wait` or both. New operations such as `MPI_File_iread11` therefore can be defined to indicate these soft dependencies. This technique can also be used to preserve some hard dependencies such that no `munmap` should occur before a corresponding `mmap` operation. We characterize these new operations in workload characterization to capture the operation patterns.

2.5.3 Framework of File Open Sessions

The file access patterns of parallel I/O workloads are complicated due to the fact that parallel I/O workloads usually deal with multiple processes, multiple files and multiple I/O libraries. In order to model such a workload, we apply file open session oriented characterization. A file open session describes the I/O request pattern during a specific period which starts with opening this file and ends with closing it. A file can have multiple open sessions throughout the workload.

We assign a global sequence ID to each request in the workload and represent them into a framework of file open sessions. Figure 2.3 shows such an example framework. There are 4 files in this example: A and B are private files; C and D are shared files. Each rectangle represents an open session of its corresponding file, where each numbered circle inside rectangles is an I/O request accessing this file. These operations are from column 2 which require opening the file first. Column 1 operations, on the other hand, are independent and thus are not restricted to be inside file open sessions. A number of workload characteristics will be extracted as explained in Section V. Creating such a framework of open sessions helps explore how the I/O requests to different files are correlated and how the file data operations and metadata operations are mingled.

2.6 Procedure of A Complete Solution

In this section, we describe a complete solution for parallel I/O workload characterization and generation. All the proposed approaches described previously are included in this implementation.

The whole procedure of our implemented solution has been summarized as five phases in Figure 2.4, including *sanitization phase*, *generic workload path extraction phase*, *characterization phase*, *synthetic generic workload path generation phase* and *parallel I/O generation phase*. As the flowchart shows, we first sanitize and reformat the raw parallel I/O traces, produced by LANL-Trace framework or captured by other tools, for the root process and the selected child process or processes. We then extract generic workload path using these sanitized traces. A framework of open session oriented I/O characterization on this generic workload path produces a set of characteristics which are later used to create a synthetic generic workload path. We input this synthetic generic workload

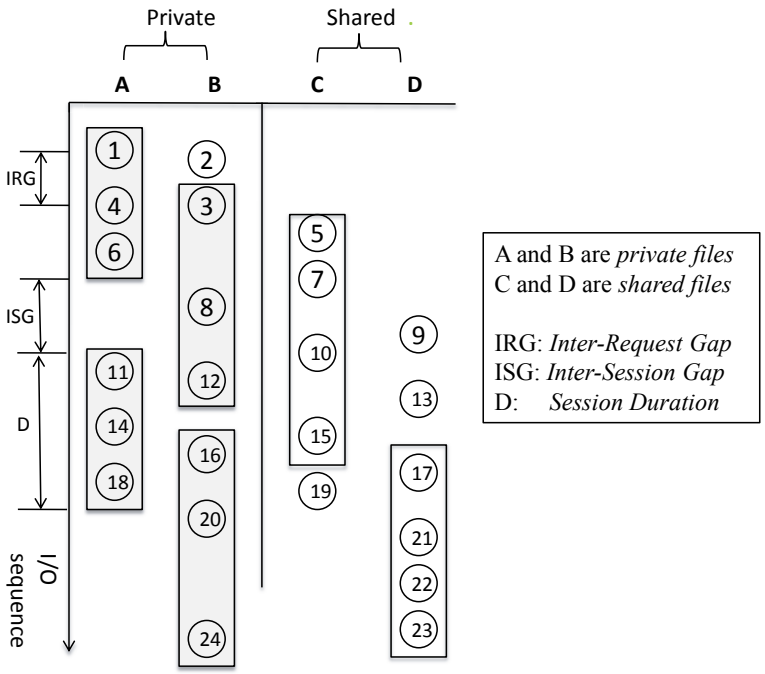


Figure 2.3: Framework of File Open Sessions

path to our workload generation engine to generate the actual parallel I/O workload for a desired number of processes. In the case that no original workload is provided, this procedure starts with the *synthetic generic workload path generation phase* but requires the user to specify the desired workload characteristics. Each of these steps is detailed in the following subsections. The output of a previous phase is the input of the next phase.

2.6.1 Sanitization Phase

The raw traces produced by LANL-Trace framework need to be sanitized due to the following reasons. First, some POSIX-IO operations are not issued directly by the application but instead are internally used by MPI-IO operations. However, due to the fact that LANL-Trace framework tool is using *ltrace* to capture POSIX-IO operations, it will just record any POSIX-IO operation it sees, irrespective of whether it is issued by the application or internally called by MPI-IO operations. Therefore, these internally used POSIX-IO operations need to be masked by a script program. Fortunately,

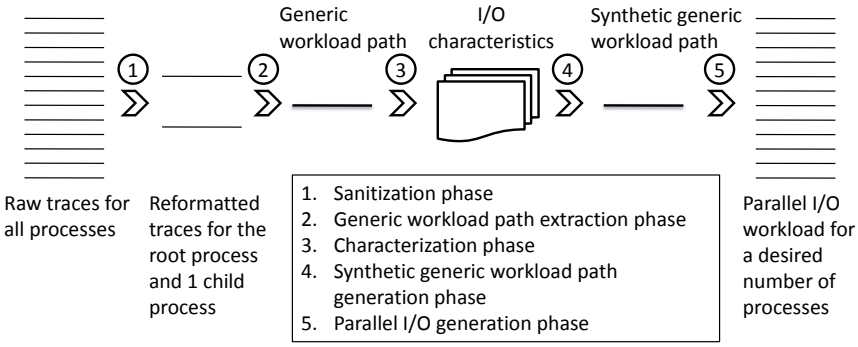


Figure 2.4: Workflow of Our Solution and Trace Transformations

LANL-Trace framework tags those MPI-IO operations that internally call POSIX-IO operations with <unfinished> and <resumed> tag pairs. Table 2.3 presents trace examples before and after trace sanitization. Before sanitization, there are four POSIX-IO operations between the tag pair of `MPI_File_open`, which are actually internally called by `MPI_File_open` itself. Thus we convert it into a single MPI-IO operation record line. Besides, a unified trace format is used to keep consistency which contains 7 dimensions: <timestamp>, <operation type>, <file name>, <offset>, <request size>, <execution time> and <extra>. We decode the file descriptors or handlers into the actual file names for ease of characterization. These 7 dimensions help us to characterize the workload later.

Only the traces of the root process and one child process need to be sanitized unless the processes of a sophisticated HPC application are divided into multiple subgroups, in which case we randomly choose one child process from each of the subgroups and sanitize their traces. The number of required subgroups can be either provided by the application owner or obtained by a bootstrap characterization. In bootstrap characterization, we take a small portion of the traces, such as first 10% of the traces, to compare the similarity between the child processes. Child processes are said to be in the same subgroup if their traces show similar operational statistics and operation reuse distance profile as we described in Section III. The number of subgroups can be decided as a result.

Table 2.3: Trace Sanitization Before and After

Before	10:48:52.404754 MPI_File_open(92, 0x807a7f8, 34, 0x8078e38, 0xbf83b9e0 <unfinished ...> 10:48:52.405470 SYS_statfs64(0x807a7f8, 84, 0xbf83b788, 0xbf83b788, 0x81dff4) = 0 <0.008302> 10:48:52.414801 SYS_umask(022) = 077 <0.000025> 10:48:52.414866 SYS_umask(077) = 022 <0.000017> 10:48:52.414936 SYS_open("/panfs/caddypan.lanl.gov /scratch1/nobody/tests/OUTPUT.1206553581.0", 32768, 0600) = 36 <0.000301> 10:48:52.416760 <... MPI_File_open resumed>) = 0 <0.011931>
After	10:48:52.404754 MPI_File_open /panfs/caddypan.lanl.gov/scratch1/nobody/tests/ OUTPUT.1206553581.0 NULL NULL 0.011931 MPI_COMM_SELF,34

2.6.2 Generic Workload Path Extraction Phase

The major task here is to separate the root exclusive I/O requests from mimicked I/O operations by comparing the root process trace with the traces of the selected child processes. To achieve this, we first create a shadow trace file for each of these sanitized traces. A shadow trace file only contains the <operation type> and <file name> dimensions of its original trace file. The rank suffix in the shared file names should be removed though.

Let us denote the root shadow trace file as R and denote the child shadow trace file as C. Then we compare R and C using native Linux utilities such as `diff` command which will output the differences as line indices. Other comparison tools can also be used here. These differences indicate root exclusive requests. The indices of these root exclusive requests in R map them back to the original trace records in the root process trace, which will be annotated with a root exclusive flag in the <extra> dimension. Other auxiliary annotations such as offset of reads and writes to shared files can also be added to facilitate later characterization. This annotated root process trace becomes the generic workload path.

The most important reason why we merge the I/O patterns of the root process and

child processes into a generic workload path, as discussed previously, is that it helps model how the root I/O requests are correlated with those of child processes, as shown in the next phase. If we characterize them separately, this connection information will be lost.

2.6.3 Characterization Phase

We characterize the extracted generic workload path to obtain the characteristics of the selected dimensions. The characteristics of the *root exclusive files*, the *shared files* and the *private files* will be stored separately to reduce the statistical interference among them.

Requests in the generic workload path are organized into a framework of file open sessions according to the global sequence IDs and the specific request information. Requests fall either inside or outside file open sessions according to the request dependencies.

For workload profiling, we made a list of open session oriented characteristics as well as their definitions in Table 2.4, which will be represented as empirical Probability Distribution Functions or Probability Functions. We use these empirical distributions for synthesizing although some dimensions such as slack time can be even fitted into existing models like Pareto distribution. Many characteristics are measured by the global sequence IDs in the generic workload path and therefore they can model the correlations between the root process and the child processes inherently. For example in Figure 2.3, there are 3 requests issued to file A during its first open session. The second request to file A (with global sequence ID 4) will not be issued until two requests (with global sequence IDs 2 and 3) are made to other files. This kind of I/O behaviors can be well represented by Inter Request Gap (IRG).

Similarly, Inter Session Gap (ISG) can control how many I/O requests should be issued to other files between two successive file open sessions of a particular file. Duration (D) describes the number of I/O requests that are issued to a particular file during one open session. File Open Times (OT) means the frequency of opening a target file. We also use file sequentiality (SEQ) and Access Mode (AM) to control the specific I/O pattern of a particular file open session. SEQ describes the I/O randomness and AM controls the types of file data operations. Some other standard characteristics such as request size (SIZE) and request offset (OFF) are also used. We use two characteristics

to count the I/O request type frequency with RT1 for the frequency of different I/O requests inside file open sessions and RT2 for those outside file open sessions. This allows us to more precisely synthesize the generic workload path.

Furthermore, the characteristics can describe the I/O patterns more precisely when used together. For example, characteristics IRG together with D can not only describe how the requests to different files are mixed with each other, but also control the arrival rate of requests to a particular file. For instance, in Figure 2.3, file A and file B have the same open duration (i.e., 3 requests) but the requests come to A in a more compact manner and go to B at a slower pace. OT and ISG together can not only control the opening frequency of a file, but also manage when to open this file.

2.6.4 Synthetic Generic Workload Path Generation Phase

The synthetic generic workload path can be constructed in a hierarchical manner with all the input characteristics. The input characteristics can also be modified or tuned to generate desired workload patterns.

In general, we first create the framework of file open sessions. I/O requests residing in these open sessions will be assigned a global sequence ID, and then we sample based on I/O characteristics previously discussed and determine the specific I/O request types and their associated arguments according to the input characteristics. Finally, we handle the I/O requests outside the file open sessions, which are generally metadata operations belonging to column 1 in Figure 2.2.

In practice, we use Algorithm 1 to create the basic framework of file open sessions. It is important to sample the dimensions in the right order so that the I/O enforcement rules and the framework of file open sessions are reflected in the synthesized generic workload path. We first set the number of files in each file type which can be the same as that of original workload (OS) or specified/modified by user. Then for each file, we need to determine how many times it will be opened in the synthetic generic workload path according to OT distribution. For each open session, we then decide their properties including duration (D), sequentiality (SEQ), and access mode (AM) using corresponding distributions. The global sequence ID of file open request can be calculated by adding an ISG to the global sequence ID of previous file close request to the same file. Specially, the open request of the first open session of each file assumes its previous file close request

Table 2.4: I/O Characteristics

Terms	Explanation
Inter file session gap (ISG)	The distance between the close of previous open session and the open of next open session of the same file, which is measured by the global sequence ID difference.
Inter request gap (IRG)	The distance between the previous request and the next request in the same file open session. It is also measured by the global sequence ID difference.
File open duration (D)	The duration of a file open session, which is measured by the number of requests belonging to this open session.
File open times (OT)	The number of open sessions of a file throughout the workload.
File sizes (FS)	The file size, which is set to the maximum sum of request offset and request size in the workload.
I/O request type (RT1)	The frequency of different I/O requests inside file open sessions
I/O request type (RT2)	The frequency of different I/O requests outside file open sessions
Offset (OFF)	The I/O access offset inside files
Request size (SIZE)	Simply the request size
File sequentiality (SEQ)	The sequentiality of a file open session, measured by the ratio of sequential file I/O requests.
File access mode (AM)	The access mode of a file open session. The same file can be opened multiple times, each of which can have a different mode. For example, a new file can be created with write only mode and reopened later with read only mode. Access mode is important to synthetic workload generation because it constrains the request types that are allowed on the target file.
Ownership (OS)	The ownership of a file, which can be either root exclusive file, private file or shared file. We count number of files in each of the three file type.
Slack time (ST)	Slack time, defined as the time between the completion of the previous request and the start of the next request. It can follow different distributions such as the *Pareto* distribution.

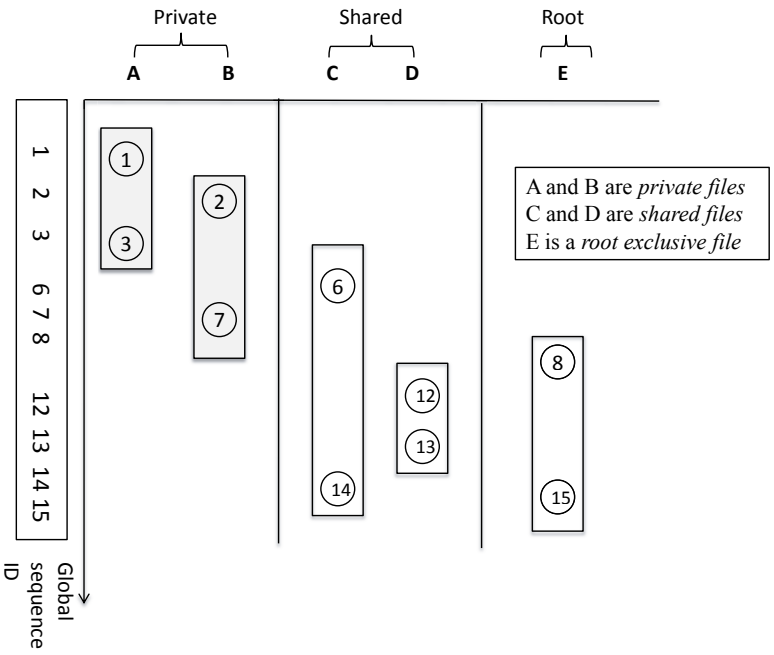


Figure 2.5: Example of Creating Framework of Open Sessions

has global sequence ID 0. Inside each file open session, we already know the number of requests (including file close request) residing there and their global sequence IDs can be easily calculated by adding an IRG to that of a previous request. Meanwhile, the operation type can be decided by access mode (AM) of the file open session and request type (RT1) distribution together. The associated offset (i.e., <offset>) can be decide by the OFF distribution and the sequentiality of the corresponding file open session together. The associated request size (i.e., <request size>) can be drawn from SIZE distribution. <extra> dimension will be fulfilled with corresponding values or default values when applicable. <slack time> can also be sampled based on a user specified distribution or the distribution in the original workload.

After we complete the framework of file open sessions, we merge the requests to different files into a single request stream according to their global sequence IDs. Throughout the whole process, we make sure every global sequence ID is unique. This single request stream might end up with "holes", which indicate the missing global sequence IDs. For example in Figure 2.5, the merged request stream is [1, 2, 3, 6, 7, 8, 12, 13, 14, 15, ...]

```
for each file class do
    determine number of files (OS);
    for each file do
        determine number of open sessions (OT);
        for each open session do
            determine open session properties (D, SEQ, AM);
            compute file open request sequence ID (ISG);
            for each operation do
                compute its request sequence ID (IRG);
                determine its operation type (RT1,AM);
                determine its argument list (OFF, SEQ, SIZE);
            end
            compute file close request sequence ID;
        end
    end
end
```

Algorithm 1: Creating the framework of file open sessions

with [4, 5], [9, 10, 11] missing. Requests in these holes are designated to be from column 1 in Figure 2.2 and they are metadata operations. This also shows how PIONEER mingles the column 1 operations with other operations.

Now we need to sample the metadata operations that are outside the file open sessions. The operation type can be sampled according to RT2. If this metadata operation requires a filename, we first sample a file type according to the sizes of RT2 of the three file types. We then randomly choose a file in the selected file type. Default values can be used for the rest of the arguments. We can make this decision because these are metadata operations that do not require file descriptors and simply retrieve file related information.

The synthetic generic workload path is created at this point and ready to be executed with the workload generation engine.

2.6.5 Parallel I/O Generation Phase

We develop and implement our own workload generation engine to actually schedule and issue the requests in a real parallel I/O environment. The workload generation engine is essentially an MPI program written in C. It takes a synthetic generic workload path

and the desired number of processes as input and generates corresponding parallel I/O workloads. The principle of our execution engine is to convert the generic workload path into an MPI program whose computing job is emulated by the slack time between successive I/O requests and whose I/O job is represented by I/O requests in the synthetic generic workload path.

The child processes spawned by the execution engine customize the generic workload path based on their MPI process ranks and then issue I/O requests according to the patterns in the synthetic generic workload path. Since file descriptors will be reused in execution, we use actual file names as argument for those I/O requests requiring a file descriptor when generating the synthetic generic workload path in the previous phase. As a result, the workload generation engine can keep track of the file descriptor usage at runtime and translate the file names into the right file descriptors or handlers for both MPI-IO and POSIX-IO operations.

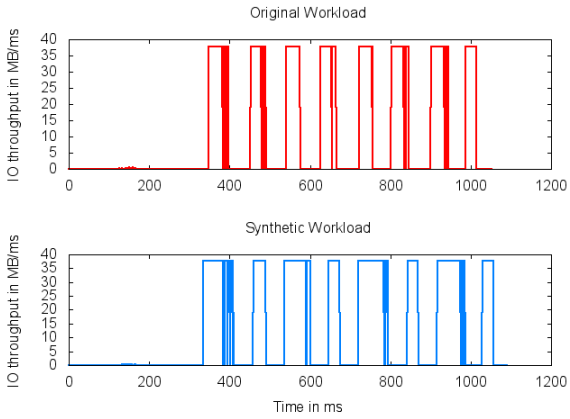
2.7 Evaluation

The goal of our evaluations is to demonstrate the effectiveness of our solution by comparing original workloads and synthetic workloads using popular HPC benchmarks and applications, in a representative HPC environment.

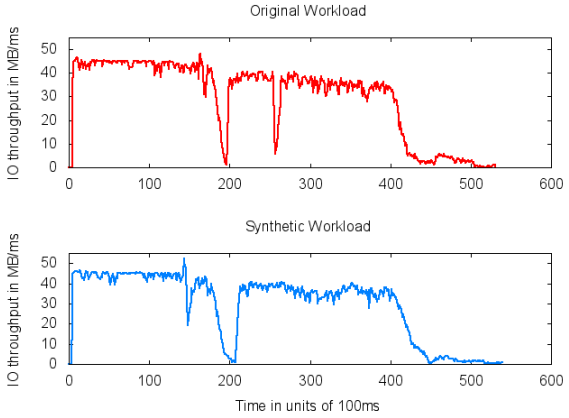
2.7.1 Target Applications and Traces

In our experiments, we use two popularly used HPC benchmarks: MPI-IO Test and IOR2, both of which are I/O-intensive. We also use a real HPC application called iPic3D [43]. MPI-IO Test is developed by LANL and is written with parallel I/O and scale in mind. As a result, it is popularly used to test parallel I/O performance at the scale of big clusters. By default, MPI-IO Test will write a specific pattern to a file, close the file, open the file for read, read the data, check for data integrity and close the file. Access patterns can be tuned with several parameters. On the other hand, IOR2 is part of the ASCI Purple Benchmarks developed at LLNL for evaluating parallel I/O performance [44]. iPic3D, a three-dimensional parallel code, is a high performance simulator of space weather. It uses HDF5 to store and access data.

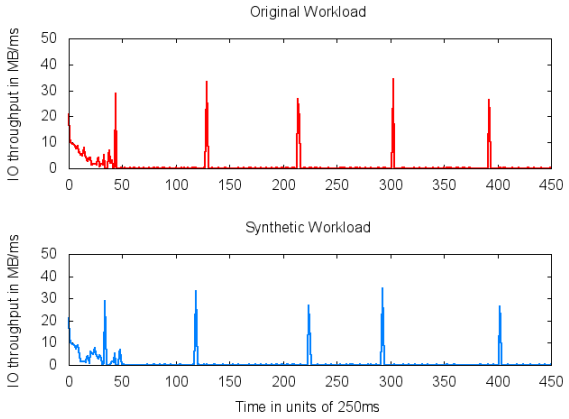
We run the three applications using 1024 processes on Itasca [45], a cluster system



(a) IOR2

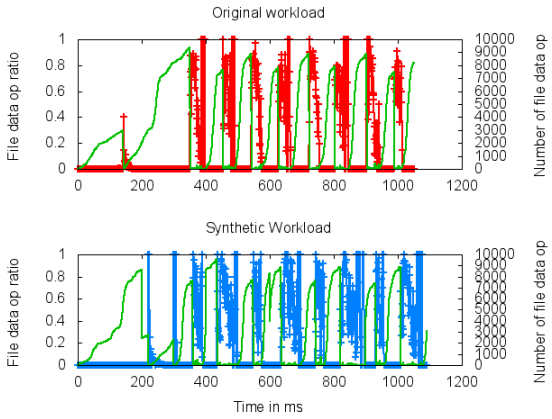


(b) MPI-IO Test

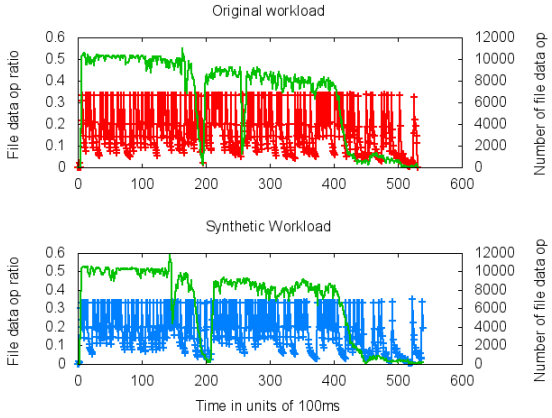


(c) iPic3D

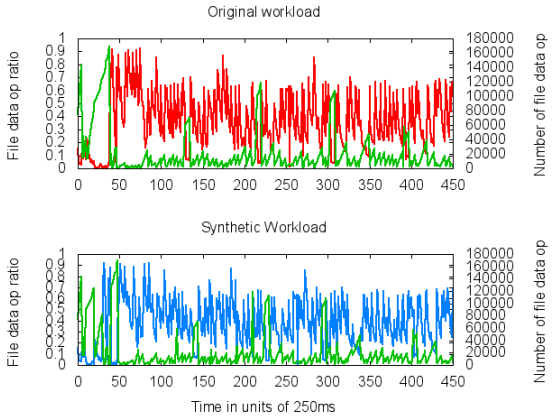
Figure 2.6: I/O Throughput Evaluation



(a) IOR2

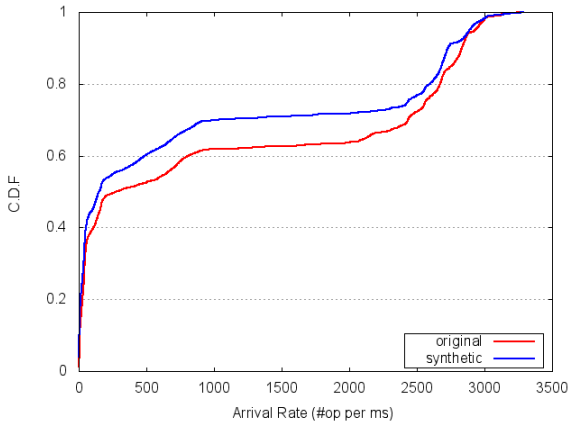


(b) MPI-IO Test

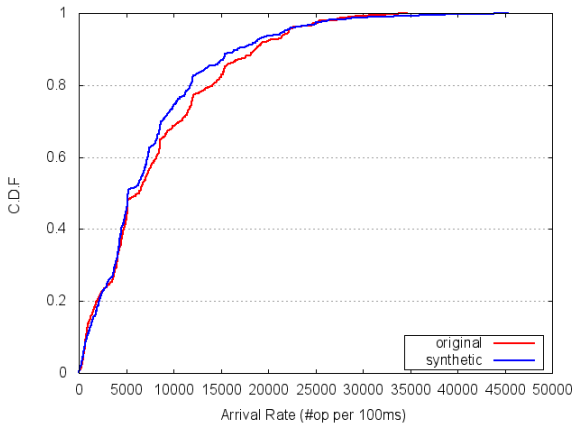


(c) iPic3D

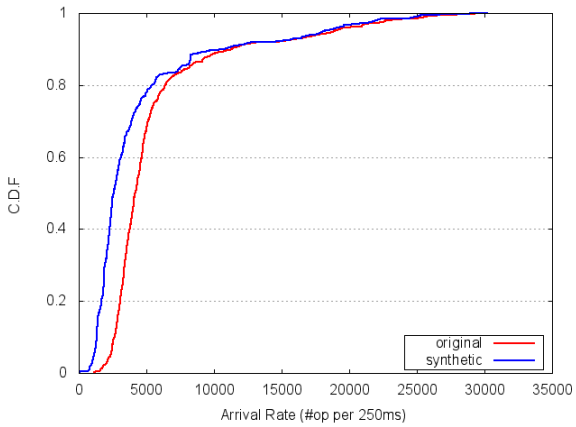
Figure 2.7: File Data Operation Ratio Evaluation



(a) IOR2



(b) MPI-IO Test



(c) iPic3D

Figure 2.8: Arrival Rate Evaluation

at Minnesota Supercomputing Institute. We capture their traces with LANL-Trace framework tool. LANL-Trace is also used when we synthesize corresponding parallel I/O workloads with our workload generation engine using the same numbers of processes. Itasca consists of more than 8000 compute cores and 24 TB of main memory. It also has a large Lustre file system storage (/lustre) of size more than 500 TB, which serves as a large shared scratch space. In all of our experiments, principal data files used by both benchmarks reside on the Lustre file system.

2.7.2 Comparison Metrics

For meaningful comparisons and evaluations, we use I/O throughput and file data operation ratio, which are orthogonal to input parameters but implicitly controlled by our solution. We also use the request arrival rate to show the burstiness in the workload are also emulated.

I/O Throughput

I/O throughput shows how fast data can be read or written which inherently indicates the impact of slack time and request response time. We present in Figure 2.6 the comparison between the original workloads and the synthetic workloads in terms of I/O throughput in MB/ms. The general patterns show that synthetic workloads match the original workloads well.

File Data Operation Ratio

Metadata operation overhead can have significant impact on I/O performance due to possible bottleneck at the metadata servers [42] so we choose file data operation ratio (1 - metadata operation ratio) as an evaluation metric. We present a comparison of file data operation ratio along time in Figure 2.7, after executing the generic workload paths. Each plot includes two parts, the file data operation ratio (red and blue lines, measured by left Y axis) and the absolute number of file data operations (green lines, measured by right Y axis). We can observe that the green lines generally follow the I/O throughput curves, meaning that I/O throughput is dominated by the number of read/write requests and the read/write request sizes have small variations, which is true

Table 2.5: K-S Test Results

Application	IO Throughput	File Operation Ratio	Arrival Rate
IOR2	0.08	0.097	0.084
MPI-IO Test	0.062	0.07	0.069
iPic3D	0.118	0.152	0.416

after inspecting the traces. As a result, we believe the synthetic workload emulate the originals very well.

Arrival Rate

Figure 2.8 shows the arrival rate comparisons. X axis is the number of operations per time unit. A larger X value means I/O is more bursty in that time period. Y axis is the corresponding cumulative distribution function. MPI-IO Test has a more stable I/O patterns than IOR2 and iPic3D, as seen in I/O throughput and file data operation ratio plottings. As a result, its arrival rate is relatively easier to preserve in the synthetic workload. For IOR2 and iPic3D, the original workload and synthetic workload are close to each other especially when the numbers of operations per time unit are large, meaning that the burstiness is well modeled.

Plotting Measurement

We apply KolmogorovâĂŞSmirnov test [46] to measure the similarity and closeness between the original and synthetic workload plottings. In statistics, the KolmogorovâĂŞSmirnov test (KâĂŞS test or KS test) is a nonparametric test. It can be used to compare a sample with a reference probability distribution (one-sample KâĂŞS test), or to compare two samples (two-sample KâĂŞS test). In particular, the two-sample KâĂŞS test is one of the most useful and general nonparametric methods for comparing the similarity and closeness of empirical cumulative distribution functions of the two samples. It can also be modified to serve as a goodness of fit test. The smaller the K-S value, the more similar the two distributions. The K-S test results for the three measure metrics and three applications are shown in Table 2.5. The table demonstrates a good confidence of the synthetic workloads with all K-S values being smaller than 0.5.

2.8 Conclusions

In this chapter, we propose a complete solution to parallel I/O workload characterization and synthesizing. Unlike existing work, we deal with several uniqueness and challenges with parallel I/O workloads, including inter-process correlations, I/O library complexities and dependencies, as well as specific file access patterns.

In our solution, we first shrink original parallel I/O workloads into a generic workload path by exploiting and utilizing the inter-process correlations. Then we characterize and model the resulting generic workload path, where we set enforcement rules to preserve I/O request dependencies and we model file access patterns by profiling file open sessions. We use the extracted characteristics to construct a synthetic generic workload path. Our complete solution also includes a workload generation engine that can expand the synthetic generic workload path into a complete parallel I/O workload for a desired number of processes. Our solution is demonstrated to be effective via experimenting with two popular HPC benchmarks and a real HPC application.

Chapter 3

Parallel I/O Optimizations

As mentioned in the previous chapter, I/O performance becomes increasingly challenging in HPC systems. I/O requests issued by the applications have to traverse through several software layers to access the data in the storage system. Failing to respect the correlations between many factors along this parallel I/O stack, including file access pattern, parallel I/O modes and file allocations, will lead to undermined I/O performance. Existing solutions consider only a subset of these factors or lack of efficiency in some aspect such as determining optimal I/O parameters.

Based on the knowledge we acquire from characterizing parallel I/O workloads and the investigation of many I/O factors in this chapter, we introduce IO-Engine, an intelligent I/O middleware module built into the MPI-IO library, to optimize the parallel I/O performance. IO-Engine can automatically and transparently optimize parallel I/O performance in Lustre environment based on the workload characteristics. Key IO-Engine features include I/O mode transformation, optimizing collective-I/O as well as determining proper striping policies for new files. Our experiments with three popular HPC benchmarks demonstrate that IO-Engine can constantly outperform an unenhanced MPI-IO library.

3.1 Introduction

The recent advances in High Performance Computing (HPC) realm have created more challenges on scalable and sustainable parallel I/O performance. A typical HPC system

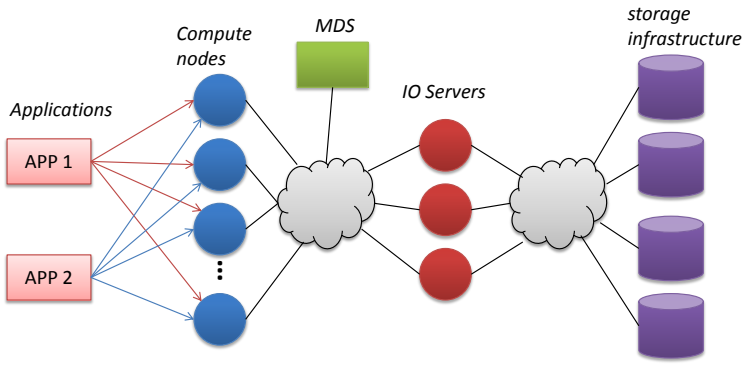


Figure 3.1: HPC Environment Abstraction

consists of several hardware components including computing nodes, Metadata Server (MDS), I/O servers, and background storage infrastructure, as shown in Figure 3.1. The Lustre environment in this work also follows this architecture. Applications are running on the computing nodes with hundreds or thousands of processes, which are using Message Passing Interface (MPI), the de-facto standard of distributed computing, for inter-process communications. The storage of a typical HPC system is managed by a scalable parallel file system such as Lustre [47], PVFS [48] and GPFS [49]. One or more metadata servers are used to maintain a unified file system namespace and file metadata. Actual file data is striped over multiple I/O servers to support concurrent accesses to the same file and improve aggregate I/O throughput.

HPC applications typically use the MPI-IO library or higher level I/O libraries such as HDF5 [50] and Parallel netCDF [51] to access their data in the parallel file system, although POSIX-I/O is still supported. These parallel I/O requests have to traverse through a deep I/O stack to access the physical storage devices as shown in Figure 3.2. There are many factors along the I/O path that can affect parallel I/O performance. These include I/O access patterns, parallel I/O access modes, parameter settings in the MPI-IO layer and parallel file systems, file allocations as well as I/O server performance impacts. If the correlations among these factors are not fully exploited, parallel I/O performance will be constrained.

Some existing studies made attempts to improve parallel I/O performance by performing a designated I/O behavior [40, 52, 53, 54, 55] or tuning I/O related parameters

[56, 57, 58, 59]. However, they either focus on a particular aspect of parallel I/O and a small set of parameters, limit to specific platforms or represent only preliminary results. We will distinguish our work more in Section 3.3. In this chapter, we investigate the impacts of those aforementioned factors in an in-depth manner and propose IO-Engine, an intelligent module built into the MPI-IO library that coordinates multiple I/O layers to optimize parallel I/O performance in Lustre environment. Given a workload, IO-Engine can automatically transform it I/O requests, tune specific system parameters and make proper file allocations based on the workload characteristics. Our experiments with several popular benchmarks demonstrate that IO-Engine can constantly outperform an unenhanced MPI-IO library.

Our contributions of this chapter can be summarized below. 1) we design a model for exploring and understanding the relationships between request size, stripe size, region size and access pattern; 2) correlations between application layers, MPI-IO layer and parallel file system, as well as the parameter impacts are explored and integrated as part of the heuristic logics; 3) we instrument MPI-IO library to help explore the root causes for low performance and bottlenecks;4) implementation of the non-intrusive IO-Engine.

The rest of this chapter is organized as follows. We introduce the parallel I/O background in Section 3.2 and related work in Section 3.3. Motivations and problem definition are presented in Section 3.4, followed by our proposed solution in Section 3.5. We then discuss the experiment results in Section 3.6. We also describe some extended work based on this study in Section 3.7. Finally we draw some conclusions in Section 3.8

3.2 Background

In this section, we introduce several important parallel I/O related concepts that will be used in our problem definition and IO-Engine's optimization heuristics.

3.2.1 File Types and MPI-IO

There are generally two types of files used by parallel applications: private files and shared files. Private files are accessed on a one-file-per-process basis without inter-process contentions. Although it is simple and effective for a small number of processes,

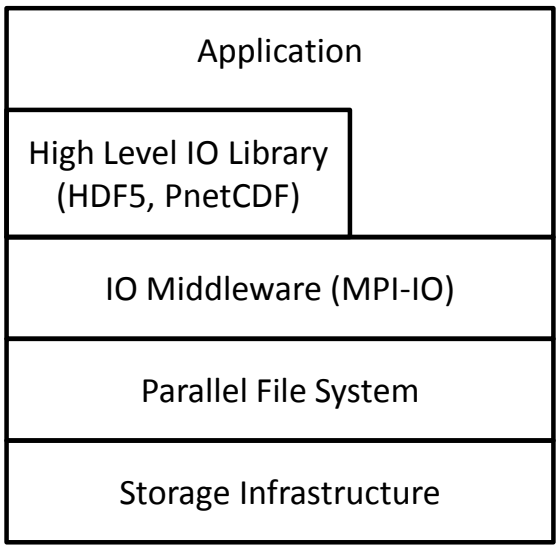


Figure 3.2: Parallel I/O Stack

it is limited in terms of scalability and portability when the number of processes is large. Shared files, on the other hand, are shared among all the participating processes and hence may have inter-process contentions or performance issues when the processes are not coordinated well. MPI defines a subset of programming interfaces to help address concurrent accesses to shared files, which are commonly referred to as MPI-IO. One popular implementation of the MPI-IO standard is ROMIO [60], which is incorporated in many MPI implementations including MPICH [61], Open-MPI [62], HP MPI, SGI MPI, IBM MPI and NEC MPI.

Application processes can use MPI-IO library or higher level I/O libraries built on top of MPI-IO (such as HDF5 and parallel netCDF) to access the shared files. The MPI-IO layer is an important software layer in the parallel I/O stack because it has the capability to coordinate participating processes and optimize their I/O requests in favor of parallel file system performance.

3.2.2 Access Patterns

File access patterns can be classified into strided and non-strided patterns. It is also common that reading/writing a shared file requires multiple I/O cycles to complete. In

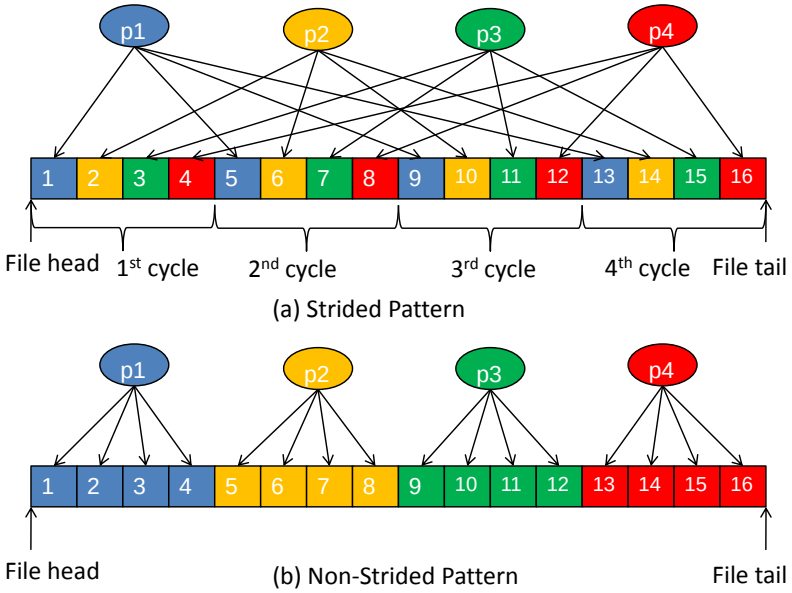


Figure 3.3: File Access Patterns

each cycle, only a portion of the file data is transferred. Figure 3.3 illustrates the two patterns using 4 processes. Assuming the file consists of 16 stripes and the request size is one stripe, it takes 4 I/O cycles for the processes to complete the whole file access in both patterns. In Figure 3.3(a), the 4 processes access stripes 1, 2, 3 and 4 respectively during cycle 1 and access stripes 5, 6, 7 and 8 during cycle 2, and so forth. This pattern is called strided because data belonging to the same process is separated by data of other processes. In Figure 3.3(b), however, the 4 processes access stripes 1, 5, 9 and 13 during cycle 1 and access stripes 2, 6, 10 and 14 during cycle 2, and so forth. Therefore, data belonging to the same process is contiguous or non-strided.

3.2.3 Parallel I/O Modes

The MPI-IO operations or APIs can generally be categorized into independent I/O[1] (e.g., `MPI_File_write()`) and collective I/O (e.g., `MPI_File_write_all()`). When performing independent I/O, participating processes access data independently and do not synchronize with each other when I/Os complete. In other words, the faster processes

[1] Independent I/O is also named as non-collective I/O in some literature.

do not need to wait for the slower processes and they can proceed to the next application instruction as soon as finishing their own I/O tasks. But accordingly, independent I/O operations lose I/O aggregation opportunities.

On the other hand, collective I/O operations utilize inter-process communications and synchronizations to provide collaboration between the participating processes and reorganize their I/O requests in the hope of achieving better aggregate performance. Importantly, two-phase I/O [40] is a collective I/O optimization technique in ROMIO which uses a subset of participating processes (named aggregators) to perform actual I/O communications with the underlying parallel file system on behalf of the rest of the processes. As a result, collective I/O can be further divided into collective I/O with aggregators and collective I/O without aggregators. Collective I/O without aggregators is essentially independent I/O with synchronizations and thus is between actual independent I/O and collective with aggregators. Therefore we do not discuss it in this chapter. Collective I/O means collective I/O with aggregators thereafter in this chapter.

Two-phase I/O or collective I/O contains a shuffle phase and an I/O phase. It will first gather request information from all processes and calculate the aggregate request range. This aggregate request range will be properly partitioned into several so called "file domains" which will be assigned to the aggregators. This partitioning and assigning process is called "file domain partitioning".

Once this is done, shuffle phase and I/O phase will start. The sequence of the two phases depends on the request type. Shuffle phase takes place first for writes and I/O phase takes place first for reads. During a shuffle phase, data will be exchanged between aggregators and other processes according to the assigned file domains. During an I/O phase, aggregators will issue actual I/O requests to the parallel file system.

There are two parameters in the MPI-IO library used to control the two-phase I/O activation: *romio_cb_read* and *romio_cb_write*. Both parameters default to value *automatic*, which means two-phase I/O will only be activated when some requirements are satisfied. For example, the aggregate request range should be contiguous. Note that two-phase I/O can also be enforced by setting the two values to *enable* or disabled by setting the values to *disable* [63].

3.2.4 File Allocations

Files in a parallel file system are striped over multiple I/O servers (or OSTs in Lustre terminology) to achieve better aggregate I/O throughput. There are two striping policies in Lustre file system. The default is Round-Robin allocation. Once the file stripe count (SC) and stripe size (SS) are set, a file will be striped across SC consecutive OSTs starting at a particular OST which is chosen according the OST space capacity usage. By default, an OST with lower usage is often chosen as the start OST so that all OSTs are guaranteed to have similar space usage. However, users are allowed to set customized striping policy for their working directories and files (including starting OST, stripe count and stripe size), which will make OSTs space capacities unevenly utilized. When the space usage difference is bigger than a threshold (20% by default), the second striping policy will be used which is called Weighted Random Allocation. The weighted allocation scheme will give higher priorities to OSTs with lower space utilizations to gradually even the space utilizations on all OSTs.

Parameters *striping_factor*, *striping_unit* and *romio_lustre_start_iodevice* can be set to desired values through MPI hints to control the striping count, stripe size and start OST respectively. Note that newly created files will inherit the striping scheme of the parent directory if these parameters are left unset.

3.3 Related Work

Process collaborations have long been considered as an effective method of improving I/O performance. Seamons et al. introduced the server-directed I/O scheme [64] where the I/O nodes collect requests from computing nodes to explore the I/O sequentiality in favor of disk performance. The most widely used optimization for MPI collective I/O is the two-phase I/O [40] algorithm. Two-phase I/O makes performance improvements by selecting a subset of the application processes to collect requests from individual processes and organize them into bigger contiguous requests. Two-phase I/O has been the foundation for many other approaches including ParColl [52], LACI/O [53] and view-based collective I/O [54], etc. Liao and Choudhary proposed several file domain partitioning algorithms [55] to improve two-phase I/O performance by reducing locking overhead in the underlying file system.

There are many parameters along the parallel I/O path (mostly in MPI-IO layer and parallel file systems) that will lead to undermined I/O performance when left to their default values. There are some work focusing on improving I/O performance by tuning these parameters include [56, 57, 58, 59].

For example, Chaarawi and Gabriel introduced a model [57] to choose the optimal number of aggregators at runtime based on factors including the file view, process topology, the per-process write saturation point, and the actual amount of data written in a collective write operation. However, their model does not take into account the importance of physical file layout.

Worringen implemented an approach in NEC's MPI implementation [59] to automatically determine the optimal setting for file hints related to collective MPI-IO operations. Their approach considers five parameters (cb_pros, cb_config_list, cb_read and cb_write and cb_buffer_size) and is specific to NEC's MPI-2 implementation (i.e., NEC/SX) and NEC's GFS file system. Their approach is inflexible in determining the parameters' optimal values. For example, their approach replies on phase change detection to adapt the buffer size.

Liu et al. evaluated several collective I/O and non-collective I/O related MPI parameters including romio_ds_read, romio_ds_write, and envisioned an automatic MPI-IO tuning tool based on Periscope Tuning Framework [57]. The shorting coming of this work is that they used a simplified strategy which explores next tuning parameter based on the best setting for previously explored parameters. This reduces the testing time but does not visit all the parameter permutations, which will result in failure of recovering correlations among I/O factors. Furthermore, this work only presented some parameter space exploration results without theoretical analysis.

McLay et al. developed a model [56] for understanding MPI collective write on Lustre and offered suggestions on selecting striping count to avoid "chasm" problem for Lustre file system. However, they did not study MPI collective read and non-collective MPI-IO workloads.

Behzad et al. proposed an empirical parameter training based approach [65, 66] to automatic parameter tuning for HDF5 applications. This approach searches for best parameter values by repeatedly running the application with different parameter value combinations which is termed as "parameter space search" before the actual running. The

set of best parameter values are stored as an XML file and will be loaded at runtime by the application. The downside of this approach is that some trainings can take up to 12 hours without a prediction model or 2 hours when using a prediction model proposed in [66]. This approach is also specific to HDF5 applications.

In contrast, the set of parameters crafted in our work is more complete than existing studies which including both MPI-IO and Lustre file system parameter. Besides, we provide more efficient ways for calculating the optimal parameter values based on a parallel I/O model that leads to a thorough understanding the relationships between these parameters and the workload patterns. We also implemented our approach, i.e., IO-Engine, in the MPI-IO library by extending the existing APIs to make it non-intrusive. IO-Engine can be used by any applications using MPI-IO library and/or higher level libraries built on top of MPI-IO library without modifying source code.

3.4 Motivation and Problem Definition

Our work is motivated by the fact that the current parallel I/O stack does not fully utilize the workload characteristics and exploit the correlations among the parallel I/O related factors described in the previous section. In order to maximize the parallel I/O performance, several conditions must be satisfied. First, I/O access modes must match the I/O access patterns. Second, logical I/O access from the application must match the physical data layout in the parallel file system or data striping across I/O severs. Third, I/O related parameters in both MPI-IO layer and parallel file system must be properly tuned. Fourth, file striping must be done according to the I/O characteristics. Last but not the least, large performance variations among the I/O servers must be taken into account for fast new file allocations. Failing to respect the above factors will lead to undermined parallel I/O performance. We formally define our problem as follows.

Goal: Given a HPC application or a parallel I/O workload in Lustre environment, design a non-intrusive tool that automatically optimizes I/O performance based on run-time workload characteristics. Performance improvement is defined as increased I/O throughput or reduced I/O latency.

General Inputs:

- Process topology: the total number of processes executed by the HPC application

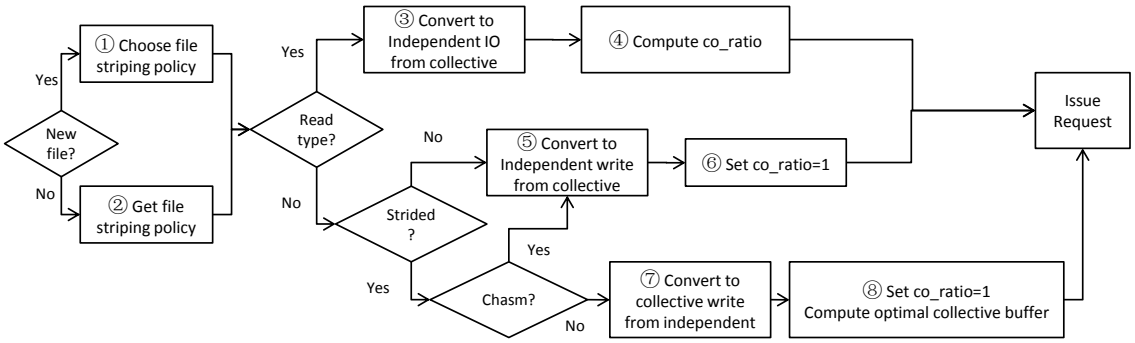


Figure 3.4: IO-Engine Heuristic Logic

(#processes), the number of computing nodes used (#nodes), and the number of CPU cores used on each computing node (#cores).

- I/O workload information: a sequence of I/O requests, each of which can be represented as a tuple (request type, request size, and request offset). Request type is defined by the MPI-IO APIs. Some APIs do not require explicit offset as an argument.
- Existing file striping information: striping count, stripe size for existing files. File striping information can be retrieved from underlying parallel file system through ADIO layer [67] which is a software layer inside ROMIO to support different underlying file systems. For example, we can use `ADIOI_LUSTRE_get_striping_info()` for this purpose in the Lustre file system.

Outputs: A set of automatic optimizations that lead to increased I/O throughput or reduced I/O latency.

3.5 Proposed Solution: IO-Engine

In this section, we describe the overview, design details and the implementation of IO-Engine.

3.5.1 Overview

Through an in-depth analysis of the current parallel I/O stack and extensive experiments, we organized several guidelines into a heuristic map as shown in Figure 3.4. IO-Engine follows this heuristic map to automatically optimize a given workload.

When the application opens a file, IO-Engine first determines whether this target file is a new file to be created or an existing file by checking the `MPI_MODE_CREATE` flag in the `MPI_File_open()` API. If this is a new file, IO-Engine will decide a striping policy (Step 1). If the accessed file is an existing file, IO-Engine will retrieve the striping information from the underlying parallel file system which will be stored in ADIO file handler of this file (Step 2). The striping information will be used later.

When the application issues an actual parallel I/O operation to access the file data, IO-Engine first checks the operation type. If it is a read operation, IO-Engine will decide to use independent I/O mode and thus will convert a collective read into an independent read[2] (Step 3). Meanwhile, IO-Engine sets the parameter $co\_ratio = min\{\#processes/\#OSTs, 4\}$ (Step 4). co_ratio controls Lustre parallelism which will be discussed in Section 3.5.2. On the other hand, if the operation type is write, IO-Engine will continue to check the access pattern by examining the request data ranges. If the access pattern is non-strided, IO-Engine will convert the collective write into an independent write (Step 5) and set co_ratio = 1 (Step 6). If the access pattern is strided, IO-Engine will decide to use collective I/O and convert an independent write into a collective write[3] (Step 7) and meanwhile set co_ratio = 1 (Step 8). IO-Engine will also fine tune the collective I/O by calculating an optimal collective buffer size and tuning the buffer size accordingly. When all the I/O tasks are done with the file, this file can be closed in the traditional way.

3.5.2 Heuristics Details and Justifications

In this subsection, we describe the details of each heuristic step. The justification for each step includes a theoretical analysis and test experiment results.

For test experiments, we run three popularly used benchmarks, IOR2 [18], MPI-IO

[2] If it is already an independent I/O operation, do nothing.
[3] If it is already collective write, do nothing.

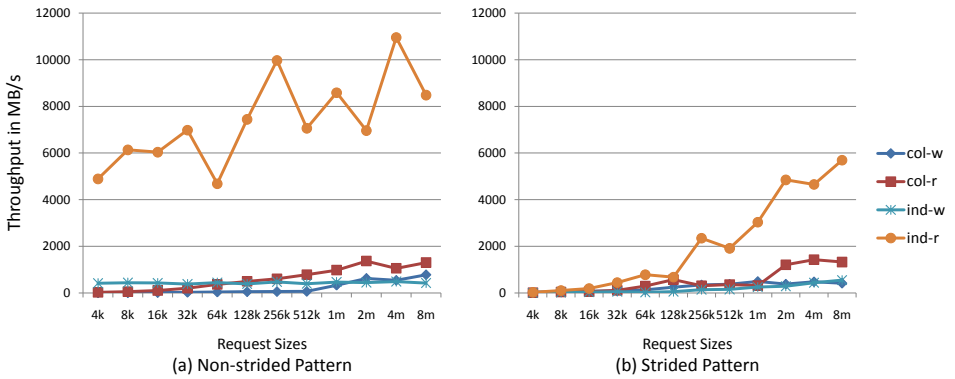


Figure 3.5: I/O Mode Comparisons

Test [24] and mpi-tile-io [68] on Itasca [45], a cluster system at Minnesota Supercomputing Institute [39]. We show the plottings for IOR2 only due to a large volume of plottings from experiments. IOR2 is part of the ASCI Purple Benchmarks developed at LLNL for evaluating parallel I/O performance. Itasca consists of more than 8000 compute cores and 24 TB of main memory. A large Lustre file system storage (1 MDT and 60 OSTs) of capacity over 500 TB is installed to serve as a large shared scratch space. In all of our test experiments, we run the benchmarks with 128 processes. When not separately stated, files accessed by the benchmark applications are striped over 8 OSTs and stripe size is 1 MB.

We now visit the details of the 8 steps in the following subsections.

Tuning File Striping (Steps 1 and 2)

For new files to be created, HPC application developers can either use the striping policy inherited from the parent directory or manually customize the striping policy by setting desired values to *striping_factor*, *stripe_size* and *start_iodevice*.

These parameter may have impacts on independent I/O and collective I/O performance. For example, McLay et al. showed that improperly configured striping count can create "chasm" problem [56] and have adverse impacts on collective write performance. This "chasm" problem will be introduced in Section 3.5.2 and the heuristic logic for step 1 is also described there because the discussion is strongly related to the collective I/O internals.

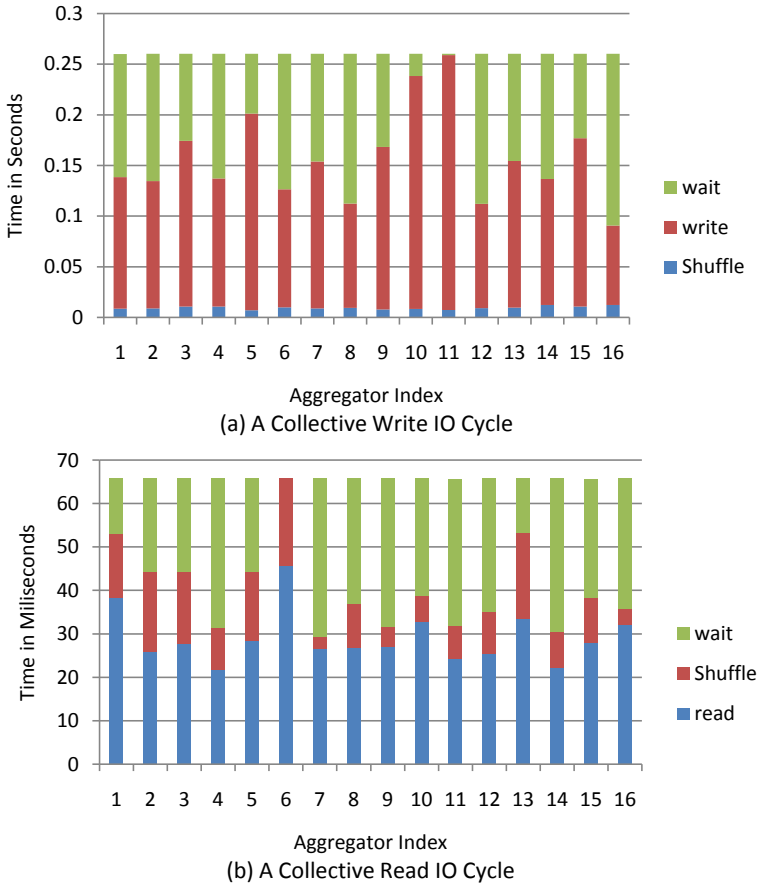


Figure 3.6: Collective I/O Breakdown

For existing files, IO-Engine will retrieve striping information from the underlying parallel file system which can be stored in the corresponding ADIO file handler data structure (step 2). The striping information will be used in later optimizations which can be retrieved with the `ADIOI_LUSTRE_get_striping_info()` API in Lustre.

I/O Mode Transformation (Steps 3-5)

As introduced previously, MPI-IO library provides different I/O modes including independent I/O (ind), collective I/O with aggregators (col). We identify suitable I/O access pattern for each of these modes and make corresponding heuristic rules for IO-Engine.

We conduct permutation experiments that covers different combinations of request

sizes, parallel I/O modes and file access patterns using the three aforementioned benchmarks and show the results for IOR2 in Figure 3.5. We can make several interesting observations which are detailed as follows.

Observation 1: All reads are faster than corresponding writes, especially obvious for the independent I/O case. This is mainly because of two following reasons. First, locking overhead for writes is more expensive than reads. Lustre uses the extent-based locking mechanism [69]. A write request usually will be granted a lock on an extent larger than the request in order to reduce the overall lock overhead. If a second write is accessing an extent that does not conflict with the first write request, then the first request will relinquish the corresponding extent to allow the second write to lock it. If a write does conflict with a current read or write request, its lock request will be enqueued until the previous request completes. However, concurrent read requests are allowed to have overlapped data section and do not have to go through the lock relinquish process. Second, read performance greatly benefits from read-ahead [69] caching scheme in Lustre.

Observation 2: Independent read is faster than collective reads for strided pattern. In order to understand the causes, we instrument the MPI Profiling Environment (MPE) [70] and install it to track the fine-grained time costs inside each I/O mode. Particularly, we break down the collective I/O execution time into shuffle cost, I/O cost and synchronization cost. We find that shuffle cost is minimal compared to the I/O and synchronization (or wait) costs as shown in Figure 3.6. Average shuffle cost is only about 10 milliseconds for both read and write. An important observation here is that great performance variations exist among the aggregators. Since collective I/O is synchronized, fast aggregators have to wait for slow aggregators in each cycle. When there are multiple I/O cycles, the total execution time is the sum of time costs of the slowest aggregators in each cycle.

We suspect that the performance variations between aggregators is mainly caused by the performance variations among the accessed OSTs. To verify that, we use 32 MPI processes to write to 32 OSTs in a one-to-one mapping with each process writing an individual 100 MB file starting at the same time. The write latency in shown in Figure 3.7. Some OSTs are significantly slower than others. Apparently, aggregators accessing slow OSTs will fall behind those accessing fast OSTs.

And there are at least three reasons for the OST performance variations. First, data is not evenly distributed because Lustre supports customized striping policy. Statistically, OSTs with larger utilization or more data will be more frequently accessed assuming data is evenly accessed. Second, file hotness or the file access frequency is different practically. Files or stripes stored on some OSTs may be more frequently accessed, resulting in different load on OSTs. Third, locking overhead and degree of inter-process contentions can be different, even if the same number of processes are accessing the OSTs. These conclusions can be utilized for future studies such as OST load aware file allocation in Lustre.

Observation 3: Independent read is much faster than collective read for non-strided access pattern. This can be explained easily given Observation 2. Assuming two-phase I/O is enforced, the aggregation benefit is less for non-strided pattern than for strided pattern because the aggregated request range is contiguous under strided pattern and thus file domain partitioning algorithms can chunk it based on stripe boundary to reduce locking overhead. This is impossible in non-strided pattern. If two-phase I/O is not used, then collective read without aggregators is essentially independent read with synchronization overhead. Therefore, independent read is faster than collective read here.

Observation 4: Independent write is faster than collective writes for non-strided pattern. The reason is similar to that for Observation 3. Moreover, independent write is suitable for non-strided pattern because requests are scattered and the processes will merely contend on the same stripe. Of course, there are occasions where moderate contentions can occur as discussed later.

Observation 5: Independent write is slower than collective write for strided pattern when request size is smaller than the stripe size. For write requests larger than a stripe, there is no big difference between the two modes. Strided pattern and small request size can form a larger aggregate request range which is perfect for file domain partitioning algorithm to do stripe boundary aligned chunking. Independent write, on the other hand, will suffer large locking overhead because multiple processes can contend on the same stripe.

Observation 6: Independent I/O performance for non-strided pattern is generally better than I/O performance for strided pattern. To explain this, we define a model to

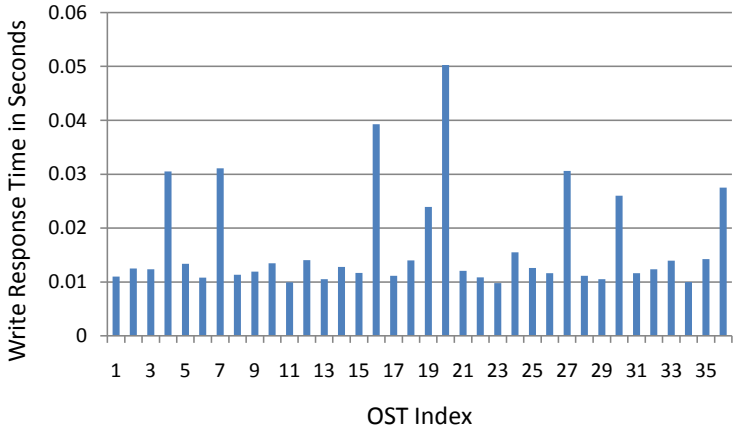


Figure 3.7: OST Performance Variations

better describe the access patterns. We use three concepts in this model: request size (RS), region size (RE) and stripe size (SS) as shown in Figure 3.8. The region size is only used in non-strided pattern and is defined as the size of the contiguous file region owned by a single process in the non-strided pattern. In the non-strided pattern, region size is always multiple times larger than a request size.

Based on the relationships among RS and SS, there are three cases for the strided pattern.

- RS > SS (illustrated as ③ in Figure 3.8(b)): A request will span multiple (at least two) stripes at the same time, possibly producing contention for some of them if RS is not a multiple of SS and the requests are not aligned with stripe boundary.
- RS = SS (illustrated as ② in Figure 3.8(b)): Stripe aligned requests will not generate any contention. Nevertheless, there will be moderate contentions on stripes if the request offset is shifted and not aligned with stripe boundary. Each stripe will be contended by two processes.
- RS < SS (illustrated as ① in Figure 3.8(b)): This will generate the most contentions among the three cases. At least two processes (or multiple processes in the worst case) will contend on every stripe. Collective I/O with aggregators is most desirable in this case because the aggregate request size for aggregators are

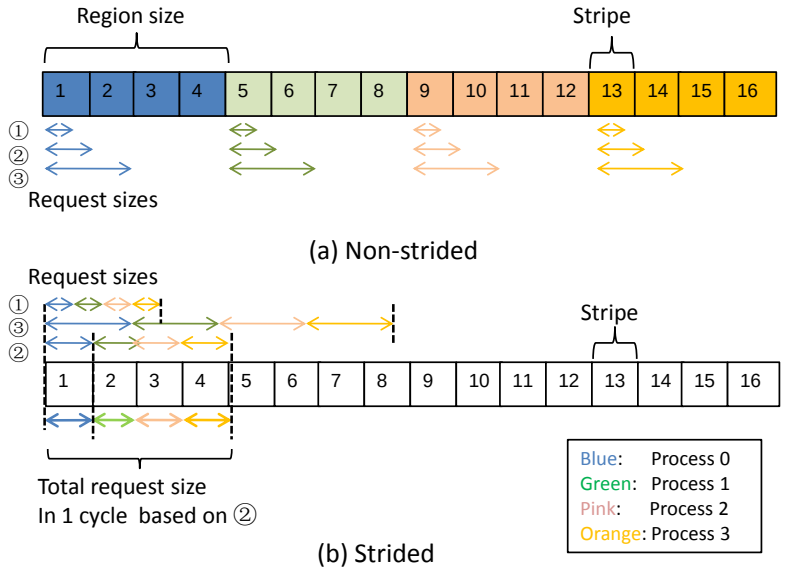


Figure 3.8: Access Pattern Modeling

possibly close to or larger than the stripe size, and appropriate file domain partitioning algorithms can make the aggregators' requests aligned with the stripe boundary. RS $<$ SS thus converts to RS $=$ SS or RS $>$ SS.

Based on the relationships among RS, RE and SS, there are also three cases for the non-strided pattern.

- SS $<=$ RS $<$ RE (illustrated as ② and ③ in Figure 3.8(a)): There will be only moderate contentions on the same stripe. Some data stripes can be contention free and at most two processes will contend on a single stripe at any time.
- RS $<$ SS $<$ RE or RS $<$ SS $<$ RE (illustrated as ① in Figure 3.8(a)): This will produce the best I/O performance because requests in an I/O cycle are scattered and processes are accessing separate data stripes. There is no contention on any stripe.
- RS $<$ RE $<=$ SS (not illustrated): This is the worst case in non-strided pattern. Every stripe will be contended by at least two processes or multiple processes in the worst case.

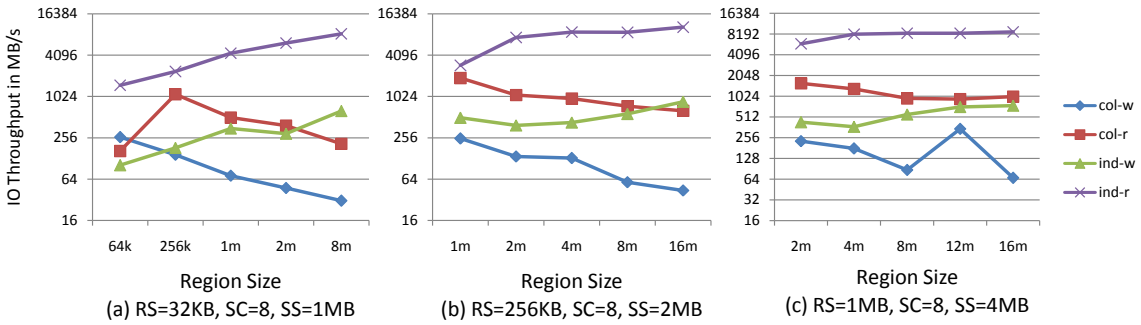


Figure 3.9: Access Pattern Modeling

As supplement to the theoretical analysis, we also study the RE size impact on the I/O performance with test experiments. The results are shown in Figure 3.9. We find that independent I/O performance generally increases as region size enlarges. This is because when region size is small, the read-ahead benefit is mostly consumed in the current I/O cycle while a larger region size can carry more read-ahead benefit to the following I/O cycles for independent read. Independent write also benefits from larger region size because larger region size can separate the independent write requests further away from each other to avoid contentions on same stripes. However, these impact is minimal when the request size is large. Particularly, we can observe that request sizes 64KB and 256KB in Figure 3.9(a) belong to the case of RS $<$ RE $<=$ SS and thus produce the worst independent I/O performance. The same thing happens for 1MB in Figure 3.9(b) and 2MB in Figure 3.9(c). On the other hand, we can observe collective I/O performance decreases as region size increases. It is because the further the I/O requests scatter, the harder for collective I/O to aggregate them.

We summarize all these discussions into two general guidelines in IO-Engine heuristics for properly choosing I/O modes for a given workload.

- If operation type is read, use independent I/O under both access patterns.
- If operation type is write, use independent I/O under non-strided pattern and use collective I/O under strided pattern.

Collective I/O Optimization (Step 8)

Assuming IO-Engine decides to use collective I/O mode, there are many parameters for further tuning the collective I/O performance. For example, *cb_buffer_size* (default to 16MB) and *cb_nodes* (default to the number of computing nodes) are most important two-phase I/O related parameters. Leaving these parameters to their default values will affect I/O performance.

cb_nodes controls the maximum number of aggregator processes. This value defaults to the number of computing nodes (#nodes) used to run this application. Note that ROMIO has another threshold for the number of aggregators which by default restricts the number of aggregators per node to 1. The parameter is called *cb_config_list*. These two conditions together make an upper bound of $U = min\{cb\_nodes, \#nodes\}$. Interestingly, ROMIO Lustre driver will set the actual number of aggregators to be the largest integer less than or equal to U that is a multiple or divisor of the striping count (SC). In other words, we can find this number by searching backward from U to 1 for the first integer that is either a multiple of SC or a factor of SC.

Although not often, this implementation sometimes will create a "chasm" phenomenon [56] if striping count and the upper bound U do not cooperate well. Assuming U is 12 and striping count is 13, then ROMIO Lustre driver will set the actual number of aggregators to 1, which significantly degrades the I/O performance because 1 aggregator cannot fully use the I/O bandwidth of 13 OSTs and the shuffle cost (data exchange between 1 aggregator and a large number of other processes) will be uselessly large. Chasm problem will not happen if $U \geq SC$ or SC is multiple of U.

The solution to this problem proposed in [56] was to carefully choose SC when creating a new file. It is suggested that prime numbers and numbers that are small multiples of prime numbers should not be used. We borrow this solution and implement it in step 1 in the heuristic map. When IO-Engine detects the manually configured or inherited striping count for a new file is a prime number or a number that is small multiples of a prime number, IO-Engine will automatically change the value to the nearest number that is powers of 2. This approach, however, cannot solve the "chasm" effect for the existing files. Our supplemental solution to this problem in IO-Engine is to transform the collective I/O to independent I/O once detecting the condition for a chasm when accessing existing files.

cb_buffer_size controls the collective buffer size for each aggregator. If this value is smaller than the aggregate request size handled by a single aggregator, each aggregator has to perform multiple pairs of shuffle phase and I/O phase to complete an I/O cycle, which is inefficient and time consuming. Moreover, it creates more RPC packets in the Lustre network which may also increase I/O latency. For example, Figure 3.10 shows the results for buffer size impacts. IOR2 runs 128 processes in both cases but fixing the number of aggregators to 4 in Figure 3.10(a) and 8 in Figure 3.10(b) respectively. The request size is 1MB. Striping count is 8 and stripe size is 1 MB. Access pattern is strided. The results show that when the collective buffer size is set to the aggregate request size for each aggregator, the performance is maximized for both read and write. So the optimal collective buffer size is 32 MB (= 1 MB * 32) for Figure 3.10(a) and 16 MB = (1MB * 16) in Figure 3.10(b). Setting buffer size smaller than the aggregate request size leads to suboptimal performance.

However, setting this value bigger than necessary is also harmful because of the memory limit on computing nodes and a higher probability of causing paging out. As a result, we apply an upper bound for the collective buffer size which is set to 64 MB in IO-Engine. At runtime, IO-Engine calculates the minimum aggregate request size based on the actual number of aggregators that will be set in the ROMIO Lustre driver and set the collective buffer size accordingly.

Lustre Parallelism Control (Steps 6-8)

Additionally, *romio_lustre_co_ratio* (default to 1) is the parameter used to control the maximum number of I/O clients for each OST [63]. When this ratio is set to 1 and multiple processes are going to access the same OST, they must be serialized. In this case only one process accesses the object, removing lock contentions from the whole object. On the downside, parallelism is reduced.

Our experiment results in Figure 3.11 show that read performance (both independent read and collective read) does increase with *romio_lustre_co_ratio* initially and becomes less impacted after a certain value (4 in our tests). This is mainly due to read taking advantages of the parallelism and merely suffering from current read lock overhead.

However, we do not see a big improvement of I/O performance for writes mainly because write requests have to go through the lock relinquish and sometimes need to be

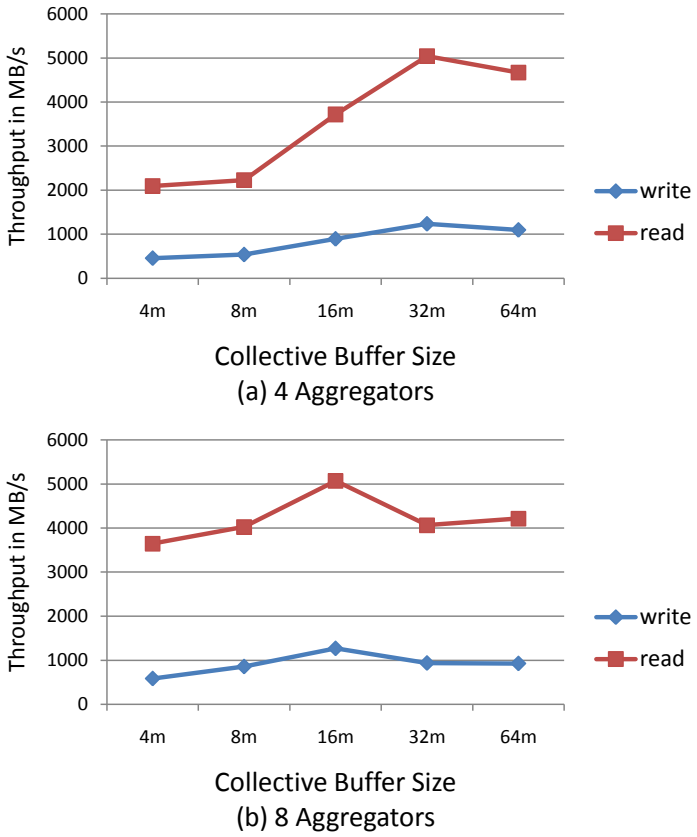


Figure 3.10: Collective Buffer Size Impacts

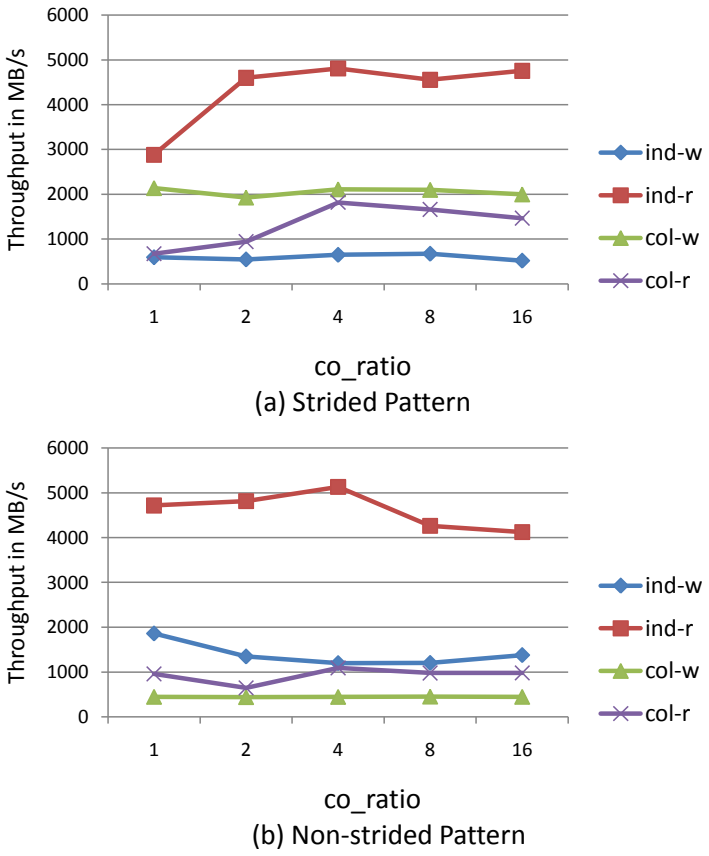


Figure 3.11: co_ratio Impacts

serialized if they contend on the same stripe. Therefore, when multiple aggregators are writing to the same OST, lock overhead can surpass the benefit of parallelism, causing random performance impacts.

As a result, IO-Engine will set *romio_ lustre_ co_ ratio* to $min\{\#process/SC, 4\}$ for reads. Since IO-Engine prefers independent I/O mode for reads, the average number of processes talking to each OST can be calculated with #process/SC. A proper co_ratio should respect an upper bound which is 4. On the other hand, IO-Engine will simply use default value for writes.

3.5.3 Implementation

IO-Engine is built into MPI-IO library by extending existing MPI-IO APIs. The transforming between collective I/O and independent I/O are performed inside corresponding APIs. API syntax is not changed. As a result, existing HPC application source codes do not need to be modified when using IO-Engine.

IO-Engine is logically a module over the native MPI-IO library as shown in Figure 3.12. When the application wants to read an existing file, its I/O flow follows Figure 3.12(a). ① The application opens the file in the traditional way, ② IO-Engine queries for the file striping information, ③ Parallel file system returns the striping information to be stored by IO-Engine, ④ The application issues a MPI-IO request in the traditional way⑤ IO-Engine transforms the request and tunes specific parameters based on the workload characteristics, ⑥ The native MPI-IO library sends the transformed request to the parallel file system⑦ The application can either perform another cycle of I/O request or close the file. The counterpart for writing new files is shown in Figure 3.12(b). It is generally the same as the read I/O flow except that IO-Engine does not query file striping information. However, a user-assisted IO-Engine that we are currently working on will have the capability of deciding proper file striping scheme based on some limited future I/O access knowledge.

3.6 Evaluation

In this section, we describe the design of our experiments and some discussions on the obtained results.

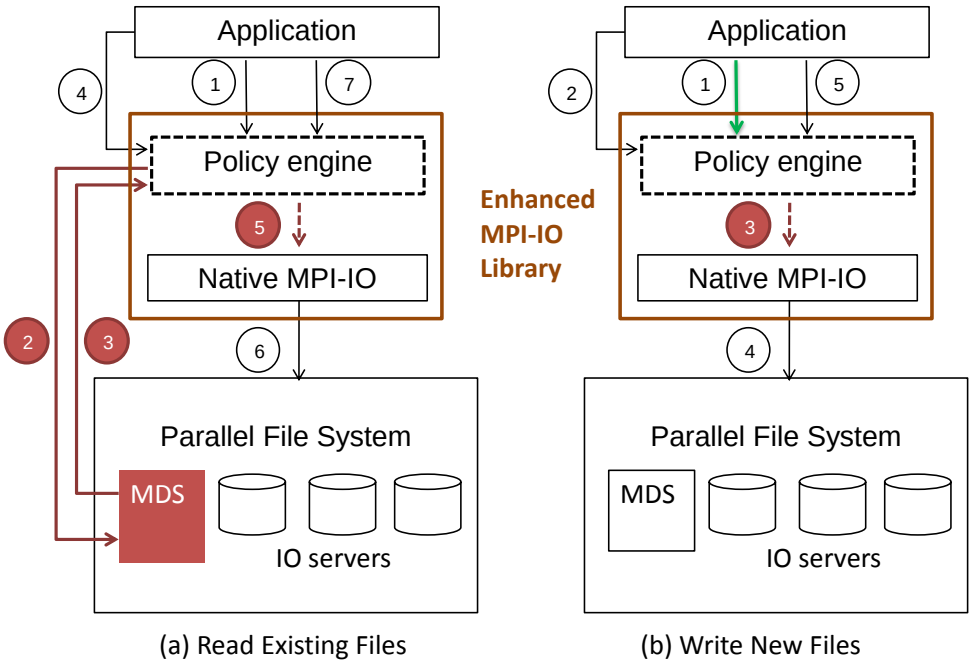


Figure 3.12: New Parallel I/O Flows with IO-Engine

We use three parallel I/O benchmarks: IOR2, MPI-IO Test and mpi-tile-io. For each of them, we compare the performance using a normal MPI implementation (mpich2 version 1.5 [61]) and an enhanced MPI implementation with IO-Engine. All benchmarks are run with 128 processes (16 computing nodes and 8 cores per node) on the cluster Itasca. The Lustre file system is installed at /lustre and consists of 1 MDT and 60 OSTs. Lustre version is 2.4. Again, our experiments cover all permutations using different file access patterns (strided vs. non-strided), request sizes, and I/O modes (independent I/O, collective I/O and IO-Engine mode).

The results are shown in Figures 3.13,3.14 and 3.15. Y axis is I/O throughput in MB/s in **log scale** and X axis is the request size. The results demonstrate that IO-Engine is able to constantly produce the best read and write performance as expected. Using its heuristics, IO-Engine can identify the best I/O modes for a given workload and tune the MPI-IO parameter based on the workload characteristics. For example, IO-Engine uses independent read mode for all reads, which greatly outperforms collective read mode

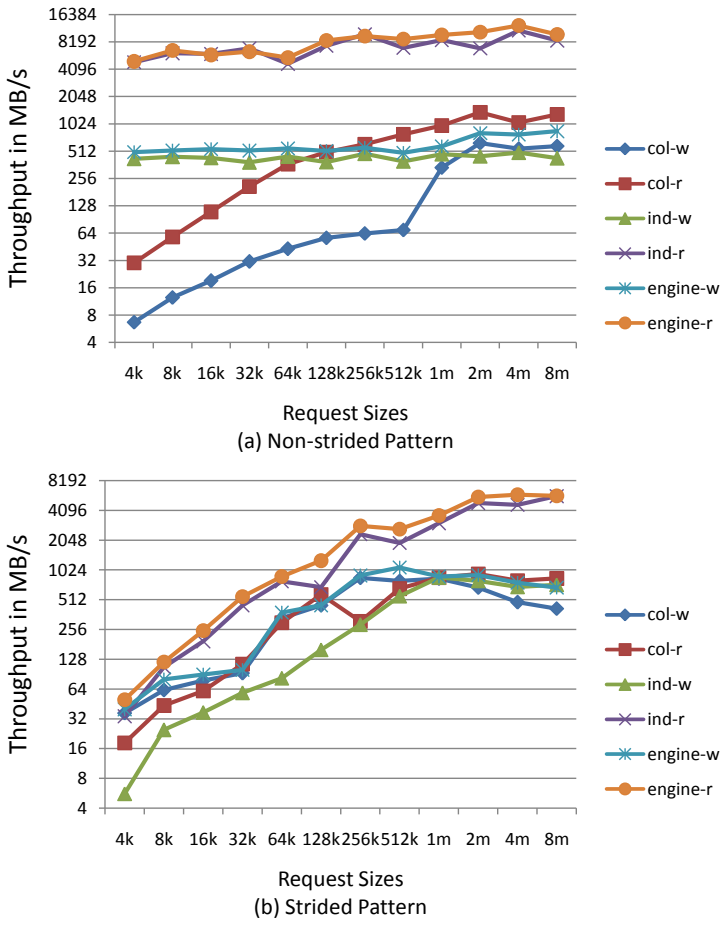


Figure 3.13: Performance Evaluation for IOR2

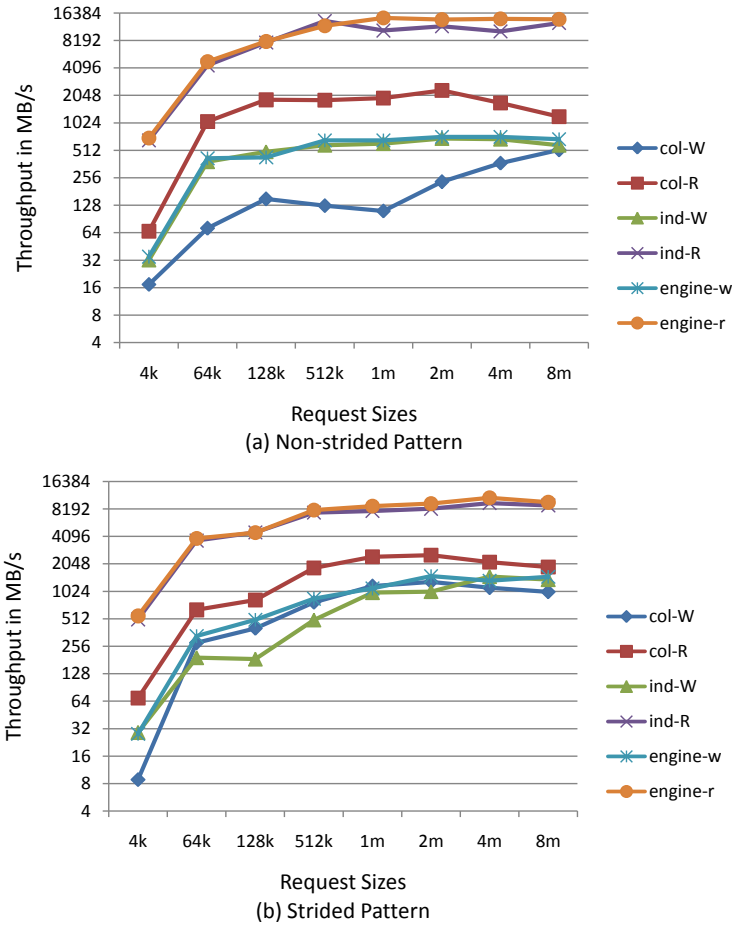


Figure 3.14: Performance Evaluation for MPI-IO Test

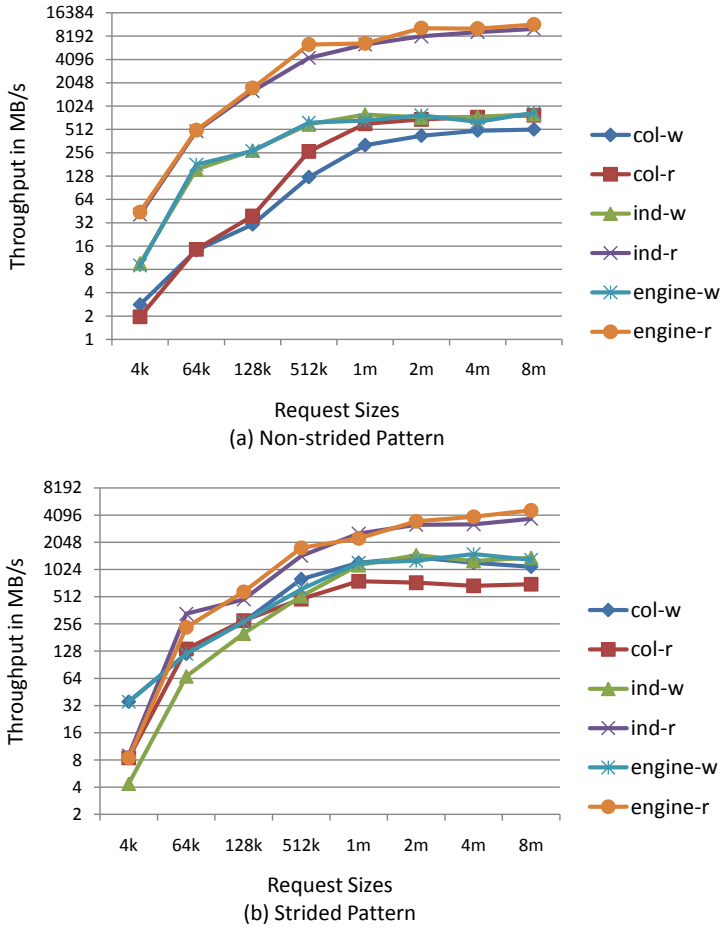


Figure 3.15: Performance Evaluation for mpi-tile-io

under all request sizes and both access patterns. By tuning *romio_lustre_co_ratio*, IO-Engine achieves better read performance than the default independent read performance. When handling writes, IO-Engine chooses collective I/O mode under strided pattern and independent I/O under non-strided pattern.

The benefit of choosing collective write under strided pattern is obvious especially when the request sizes are smaller than the stripe size (1 MB in our experiments). This is because collective writes aggregate small write requests from individual processes into larger requests using two-phase IO technique. Locking overhead therefore can also be minimized. This is also the reason that performance differences between the two modes start to decrease and independent writes are occasionally better as the request size gets larger. According to the process topology and the striping count of the files accessed by the benchmarks, 16 aggregators will be used each of which manages 8 processes. When the request size increases to 2 MB, the aggregate request size for each aggreagator is 16 MB which is equal to the default collective buffer size. However, when request sizes are set to 4 MB and 8 MB, the aggregate request size becomes larger than the collective buffer size, which requires multiple rounds of shuffle phases and I/O phases. This causes the default collective write performance fall below IO-Engine's performance as shown in the results.

On the other hand, IO-Engine uses independent writes under non-strided pattern to achieve better performance. The results show that independent write performance for IOR2 is constantly high which jumps to around 512 MB/s at 4 KB request size. The same thing happens for independent read for IOR2 which jumps to over 4096 MB/s at 4 KB request size. However, independent read and write performance increase slowly for MPI-IO Test and even more slowly for mpi-tile-io. The differences are caused by the settings of region size. Region size is set to 16 MB for IOR2, 100× the request size for MPI-IO Test and 10× the request size for mpi-tile-io. According to our I/O modeling, I/O performance increases with region size and good I/O performance can be expected when region size is larger than stripe size. As a result, independent read performance jumps to 512 KB/s before 128 KB request size and independent write performance jumps to 4096 KB/s as early as 64 KB for MPI-IO Test [4] . mpi-tile-io delayed both numbers to 512 KB because of a smaller ratio 10. We can also observe that I/O performance is

[4] 64 KB * 100 = 6.4 MB

less impacted by the region size when the request size is larger than a certain value such as 2 MB.

3.7 Extended Work

We are also working on a user-assisted IO-Engine discussed in Section 3.7 which requires some user inputs to facilitate more intelligent file striping. We define a new MPI file open API `MPI_File_open_s()` which asks for extra parameters to learn some basic characteristics of future I/O accesses to this file. One of the considered characteristics include read/write frequency because we observe that read performance tends to be more affected by stripe size while write performance tends to be more affected by striping count.

Our experiments in Figure 3.16 and Figure 3.17 show the impact of different striping counts and stripe sizes on collective I/O performance. In both cases, we use default two-phase I/O parameters.

In Figure 3.16, we fix the stripe size to 1MB and 4MB, and change the striping count from 1 to 16. Write performance generally improves as striping count increases due to two reasons. The first reason is that the average write lock overhead on a single OST will be smaller. For a simple example, if 20 stripes are accessed by 20 aggregator processes (aggregate request is aligned with the stripe boundary) during a write I/O cycle and are evenly distributed across 10 OSTs, then each OST has to perform a lock relinquish. However, if the 20 stripes are evenly distributed across 20 OSTs, then no lock relinquish needs to be done implying less lock overhead. The second reason is more OSTs provide larger aggregate I/O throughput. Read performance is less impacted by striping count because reads can be serviced with read-ahead cache in Lustre. As long as the bandwidth of OSTs are not saturated, I/O throughput for read will be less impacted by striping count.

However, read performance can benefit from relatively larger stripe size because of the read-ahead scheme in Lustre as shown in Figure 3.17, where we fix the stripe count to 4 and 8, and change the stripe size from 1MB to 16MB. Large stripe sizes can negatively impact writes because statistically more processes will contend on stripes causing more locking overhead.

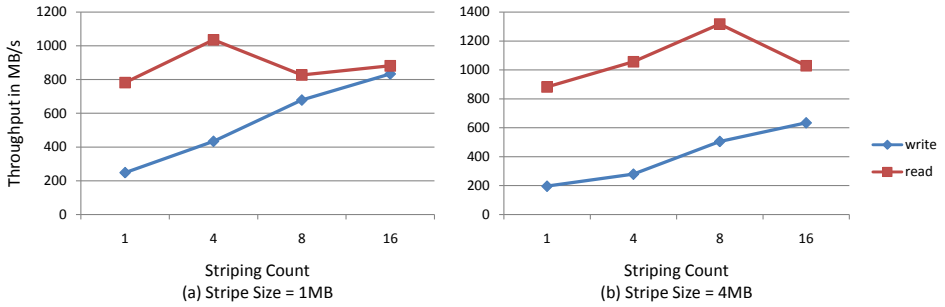


Figure 3.16: Striping Count Impact

As a result, a proper heuristic logic would be that for write intensive files, a larger striping count and a default stripe size should be used. For example, checkpointing files created by HPC applications are mostly written once and thus would prefer larger striping count. For read intensive files, a relatively larger stripe size and a default striping count should produce better performance. Of course, we need to carry out more comprehensive experiments and thorough technical research in order to validate this conclusion, which will also be our near future work.

Besides, another characteristic, the region size (the amount of contiguous data belonging to the same process), can also be used to help choose a stripe size such that RS $<$ SS $<$ RE situation can be formed which theoretically should produce the best result as discussed above.

Modifications must be made to MPI API syntax in order to pass these information. Therefore we do not include this feature in the current non-intrusive IO-Engine, but will integrate it in the user-assisted IO-Engine as part of step 1 in the heuristic map.

Moreover, this IO-Engine work can also be extended to optimize parallel I/O performance for other parallel file systems. We believe some heuristics can be shared among them but the rest are file system dependent.

3.8 Conclusions

Parallel I/O performance has been a challenge for HPC systems because of many factors along the parallel I/O path. In this chapter, we motivate ourselves by investigating

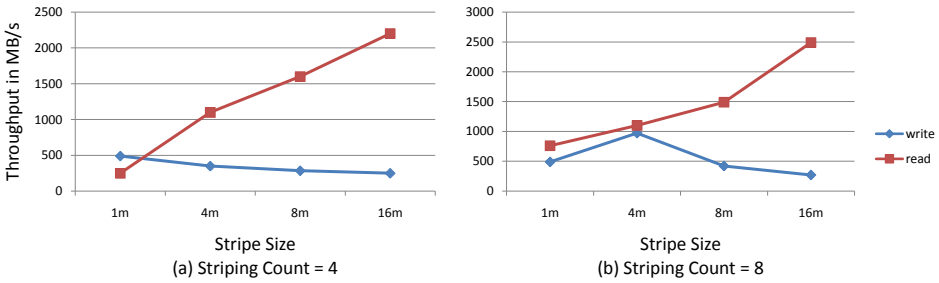


Figure 3.17: Stripe Size Impact

the parallel I/O stack and exploring the correlations among factors such as file access pattern, parallel I/O modes and specific system parameters. Based this knowledge, we propose IO-Engine, an intelligent I/O middleware module instrumented to the existing MPI-IO library that can transparently optimize HPC I/O workloads in Lustre system.

Chapter 4

In-place Update SWDs

In contrast to parallel I/O performance, storage capacity is another major I/O requirement in converged HPC systems. Traditional hard drives, which store over 80% of data in most of today's data centers and HPC systems [71], are reaching areal data density limit. New technology must be developed to keep up with the data growth pace.

Shingled Write Disks (SWDs), one of the most promising new technologies, increase the storage density by writing data in overlapping tracks. Consequently, data cannot be updated freely in place without overwriting the valid data in subsequent tracks if any. A write operation therefore may incur several extra read and write operations, which creates a write amplification problem. In this chapter, we propose several novel static *Logical Block Address* (LBA) to *Physical Block Address* (PBA) mapping schemes for in-place update SWDs which significantly reduce the write amplification. The experiments with four traces demonstrate that our scheme can provide comparable performance to that of regular *Hard Disk Drives* (HDDs) when the SWD space usage is no more than 75%.

4.1 Introduction

The low cost and big capacity make the traditional hard disk drives (HDDs) the most popular storage devices in the past decades. However, perpendicular magnetic recording technique used by traditional HDDs is reaching its areal data density limit. The data density is determined by the superparamagnetic effect (SPE) which restricts data density

in HDDs to be below 1 Tb/in^2 [72]. Therefore, several new recording technologies have been proposed to solve this problem including Heat-Assisted Magnetic Recording (HAMR) [73, 10, 74], Bit-Patterned Media Recording (BPMR) [11, 12] and Shingled Magnetic Recording (SMR) [14, 13]. Among these new techniques, SMR is the most promising technique because it does not require significant changes to either magnetic recording or manufacturing process. It can increase the data density to 2 Tb/in^2 and even to 10 Tb/in^2 when enhanced with 2-D readback [75].

SWDs increase the drive capacity by overlapping the neighboring tracks. It increases data capacity by overlapping the adjacent tracks and thus packing more data tracks into platters with the same physical dimensions. The asymmetric requirements for head width of read and write requests makes shingling technically feasible. Disk heads write a wide track but only need a narrow track for reading. Thus SMR works by writing a wide track then overwriting most of it when performing another write. The downside is what is called write amplification overhead. Assuming the write head is two-track wide, a write will now impact 2 tracks. That is, writing to a track may destroy the valid data in the following track. Consequently data is better to be written onto the tracks in a sequential manner. However, random read is still supported in SWDs.

In order for the shingled write disks to be adopted in the existing storage systems without significant performance degradation, it is necessary to mitigate or circumvent this write amplification problem. The solutions to this problem can be different depending on the specific type of SWD.

There are generally two types of SWDs: the autonomous SWDs and the host-managed/host-aware SWDs. Autonomous SWDs maintain a logical block addresses (LBAs) to physical block address (PBAs) mapping layer and therefore provide block interface to the upper level applications such as file systems and databases. On the other hand, host-managed/host-aware SWDs are simply raw devices and rely on specific upper level applications to interact with the PBAs directly. As an analogy, a solid state drive (SSD) without a built-in flash translation layer (FTL) has to rely on flash file systems to manage its physical space.

Depending on the update strategy, autonomous SWDs can further be classified into in-place update SWDs (I-SWDs) and out-of-place update SWDs (O-SWDs). To perform an update operation to a previously written track in an I-SWD, data on the following

tracks have to be safely read out first and then written back to their original positions after the data on the targeted track has been updated. To minimize this overhead, only a few tracks (4 or 5) per band are used. Besides, there will be enough separation between adjacent bands such that writing to the last track of each band will not destroy the valid data in the following band. An I-SWD maps LBAs to PBAs using a static address mapping and therefore it does not require any mapping table and GC operations. However, I-SWD space has to be organized into relatively small bands like 4 tracks per band in order to keep the write amplification overhead reasonably low as discussed in our previous paper [76]. This also means at least 20% of the total space has to be used as safety gaps between neighboring bands. Generally, bigger band size provides better space gain but worse update performance.

On the other hand, O-SWDs can provide much more space gain because larger bands (such as 100 tracks) are used. Only a neglectable amount of space is thus used for safety gaps. However, O-SWDs have their own disadvantages including the metadata overhead and the GC overhead. To perform an update operation in O-SWDs, the updated data blocks will first be written to a new place and the old data will be invalidated. A mapping table is necessary to keep track of these data movements. Besides, those invalidated data blocks must be reclaimed later by GC operations so they can be reused.

In this chapter, we discuss our design for I-SWDs. And O-SWDs will be discussed in the next chapter. Compared to O-SWDs, I-SWDs has its own advantage that it is possible to not use any GC operations and complicated mapping tables. We show that write amplification overhead of in-place update SWDs can be greatly reduced with novel static LBA-to-PBA mappings and these are simple mappings without incurring large overheads. Experiments with four traces demonstrate our proposed schemes can provide comparable performance to that of regular HDDs when space usage is no more than 75%.

The remainder of the chapter is organized as follows. Section 4.2 describes two major physical track layouts for SWDs. Section 4.3 discusses some related work and Section 4.4 shows the motivations for this work. Novel data mapping schemes with performance predictions are discussed in Section 4.5. Experiments and result discussions are presented in Section 4.6. Finally conclusion is made in Section 4.7.

4.2 SWD Layout

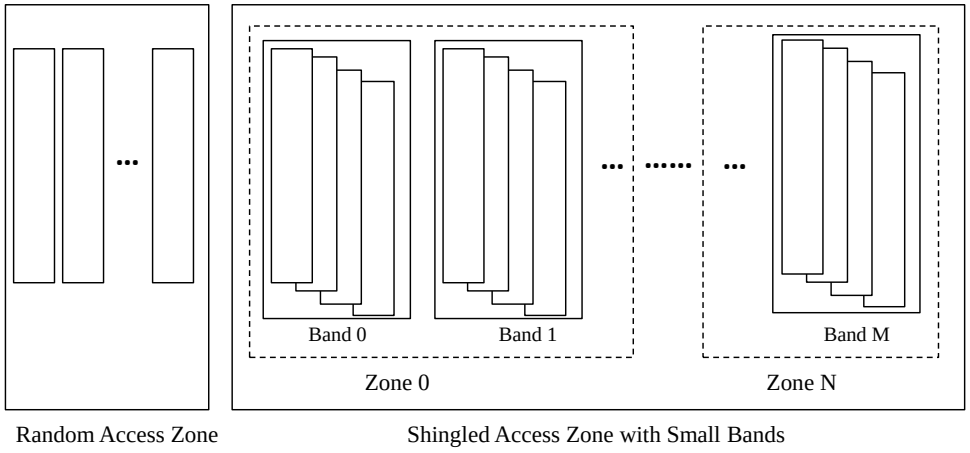
SWDs generally follow the geometry of regular HDDs except the tracks are overlapped. Similar to HDDs, each SWD may contain several platters. Physical data blocks are also addressed by Cylinder-Head-Sector (CHS). Obviously outer tracks are larger than inner tracks, so the SWD space is divided into multiple zones. Tracks in outer zones are larger than those in inner zones and have better performance too. Tracks in the same zone have the same size. Each zone can be further organized into bands if needed. A small portion (about 1% to 3%) of the total space is usually used as a random access zone (RAZ) for maintaining metadata [77, 78, 79].

I-SWDs and O-SWDs organize and use the bulk shingled access zone (SAZ) differently as shown in Figure 4.1. Autonomous I-SWDs usually organize the tracks into small bands in order to achieve a good balance between space gain and performance as discussed and evaluated in [76]. Figure 4.1a shows an example of using 4 tracks per band. However, bigger band size can be used for host-managed I-SWDs. For example, the shingled file system [77] sets the band size to be 64MB or about 100 tracks based on the track size.

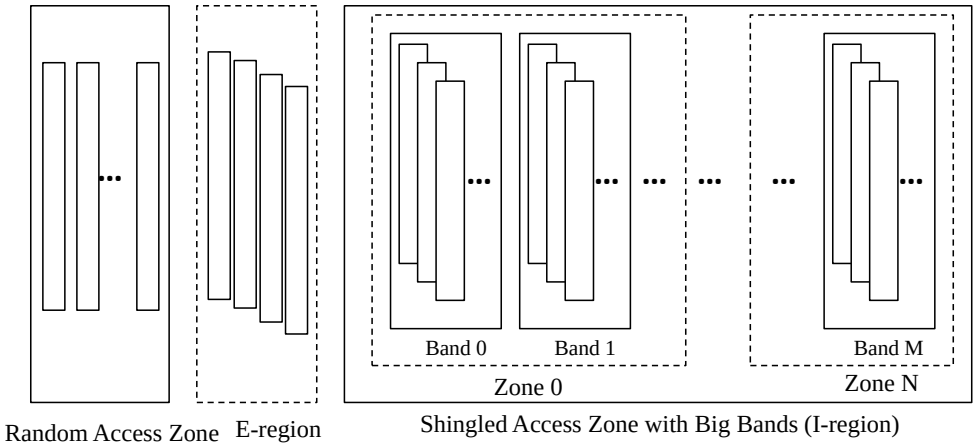
Most existing work on O-SWDs divide the shingled access zone into an E-region and an I-region as shown in Figure 4.1b. Sometimes multiple E-regions and I-regions may be used. E-region is organized as a circular buffer space and used for buffering incoming writes, while I-region is used for permanent data storage and organized into big bands. Obviously, writes to E-region and I-region have to be done in a sequential manner and GC operations are required for both regions. The E-region size is suggested to be no more than 3% [78, 79, 80, 81].

4.3 Related Work

Several studies have been done for out-of-place update SWDs. For example, Cassuto *et al.* proposed two indirection systems in [80]. Both systems use two types of data regions, one for caching incoming write requests and the other for permanent data storage. They proposed an S-block concept in their second scheme. S-blocks have the same size and each S-block consists of a pre-defined number of sequential regular blocks/sectors such as 2000 blocks as used in [80]. GC operations have to be performed in both data regions



(a) In-place Update SWD



(b) Out-of-place Update SWD

Figure 4.1: SWD Layouts

in an on-demand way. Hall *et al.* proposed a background GC algorithm [81] to refresh the tracks in the I-region while data is continuously written into the E-region buffer. The tracks in the I-region have to be sequentially refreshed at a very fast rate in order to ensure enough space in the E-region, which is quite expensive and creates performance and power consumption issues. Recently, Jin *et al.* proposed the HiSMRfs [82] which is a host-managed solution. HiSMRfs pairs some amount of SSD with the SWD device so that file metadata (hot data) can be stored in the SSD while file data (cold data)

can be stored in the SWD. HiSMRfs uses file-based or band-based GC operations to reclaim the invalid space created by file deletions and file updates. However, the details of the GC operations are not discussed. Aghayev *et al.* designed a tool framework called Skylight [83] to reverse-engineer a Seagate autonomous SWD. Skylight infers important information such as drive type, persistent cache size and GC types by measuring the latency of controlled I/O operations.

There are also some studies on in-place update SWDs. Wan *et al.* proposed two bold track and sector layouts to reduce space waste and write amplification overhead [84, 85]. The first is a wave-like shingling organization which lays out the tracks with partial overlap in two opposite radial directions like wave so the space waste on safety gaps can be reduced by about half compared to a traditional and practical shingling method. The second bold idea is called segment-based data layout which divide a region into segments in the radial direction such that the size of data rewritten can be limited to a segment instead of a whole region. The closest work to ours in this chapter is the shingled file system [77], which is a host-managed design for in-place update SWDs. The shingled file system directly works on SWD PBAs. The SWD main space is organized into small bands of size 64 MB. Files will be written sequentially from head to tail in a selected band. When a file is updated, impacted data in the subsequent tracks will be first read out to a block cache and written back to the original locations afterwards. However this work did not address the write amplification problem. Another drawback is that popular file systems (like EXT4 and NTFS) as well as other data management software have to be modified in order to use these SWDs. As a result, we do not make comparisons to this scheme. Our work improves the write amplification problem with novel address mapping schemes that make SWDs support general file systems in a drop-in manner.

4.4 Motivation

In this section, we discuss two factors that motivate our work, one is the intrinsic tradeoff in in-place update SWDs and the other is the conventional static address mapping used in regular HDDs.

4.4.1 Space Gain Tradeoff

Figure 4.1a shows the physical layout of an in-place update SWD. It uses a write head width of 2 tracks. There are k = 100 physical tracks in the random access zone, half of which are effectively used to construct the random access zone. There are also 10000 physical tracks in the shingled access zone which form m = 2000 bands with band size of 4 tracks. Totally 2000 tracks are used as safety gaps to separate the bands. The space efficiency is therefore 0.8 = 4m/(4m+m). As the write head width is 2 tracks, the actual space gain is 1.6 = 0.8*2. Although the outer tracks are bigger than inner tracks in a real disk drive, we assume a fixed track size of 100 blocks or sectors for simplicity in this example.

More generally, assume band size is N tracks and write head width is W tracks, then the *Space Gain* (SG) and the expected *Write Amplification Ratio* (WAR) for a single update request to a full band can be calculated according to Equation (1) and (2). Other discussions on areal density increase factor can also be found in [78, 86]. The WAR for a single update request is defined as the total number of requests associated with an amplified update request. Ratio 1 means no amplification is incurred. The equations clearly indicate that the bigger the band size is, the bigger space gain is but the larger write amplification overhead is created at meantime. We assume in this chapter that the band width is 4 tracks and the write head width is 2 tracks to balance this tradeoff. Other configurations, such as band width of 5 tracks with write head width of 3 tracks, can also be used as long as the tradeoff is balanced and manufacturing process allows.

$$SG = W\frac{N}{N+W-1} \tag{4.1}$$

$$WAR = \frac{1}{N}\sum_{i=0}^{N-1}(1+2i) = N \tag{4.2}$$

4.4.2 LBA-to-PBA mapping

Different from [77], the LBA-to-PBA mapping function is built into the in-place update SWDs in our design. As a result, sector-based file systems such as EXT4 and NTFS can be built on top of these SWDs nearly without any change. Write amplification management is transparent to the file systems.

Following conventional static address mapping used in HDD for in-place update SWD

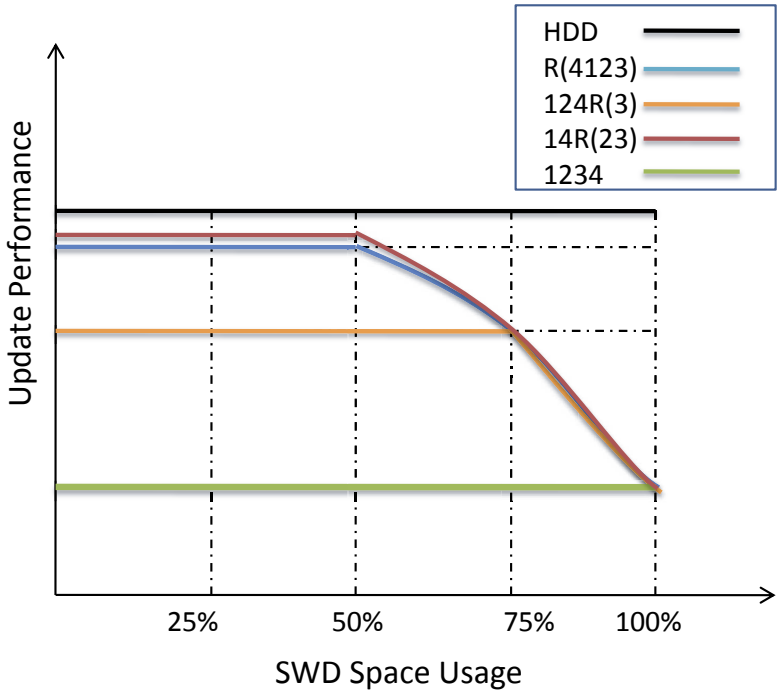


Figure 4.2: Update Operation Performance Prediction

and using Figure 4.1a for illustration, the conventional mapping scheme will sequentially map LBAs [1-100] to physical track 1, LBAs [101-200] to physical track 2 and so on. Physical track 5 is a safety gap so it is skipped. LBAs [401-500] will then be mapped to physical track 6.

This mapping scheme is noted as "1234" in this chapter as tracks are utilized in a left-to-right order. This scheme works fine for workloads with a small update percentage such as those "write once read multiple times" workloads or backup workloads. However, it will be expensive to make data updates because of significant write amplification overhead. As a result, better mapping schemes should be proposed for in-place update SWDs.

4.5 Novel Static Address Mapping Schemes

In this section, we describe several new address mapping schemes for in-place update SWDs and analyze their performance for update operations. The comparison result is

Trace	Inter-Arrival Time	Average Seek Distance	MAX LBA	MAX Request Size	Write Ratio
web_0	297.9411468	6245717.249	71116454	3200	0.70123
hp_c2247	14.19225897	273730.0428	2049836	134	0.488449
Financial2_0	0.06453672	591141.4663	2676179	3072	0.096978
SYN	50.00721344	0	2399999	8	1

Table 4.1: Trace Statistics

shown in Figure 4.2 and is validated later by experiments in Section 5.

4.5.1 General Principles

A band can be used more efficiently if we change the order of utilizing the tracks. Take one band as an example, the overall performance will be improved if the tracks are utilized in the order of "4123". In other words, the 4th track will be first used, followed by the 1st track, the 2nd track, and finally the 3rd track. By doing so, when the space utilization of this band is less than 25%, and if all data is made to be present only in the last track then all the data can be updated freely. When the space utilization is less than 50%, let data appear only in the first track and last track. The two tracks (2nd and 3rd track) between them will work as a safety gap, therefore allowing both first track and last track to be updated without incurring any extra cost. When space utilization is no more than 75%, with same principle, the 2nd track and last track are free to be updated. Only updates to the first track will incur 1 extra read and 1 extra write. However, when the space utilization becomes close to 100%, then the overhead becomes similar to the "1234" allocation. This observation triggers us to propose space allocation schemes that take SWD space utilization into consideration since the space in the SWD will be used (or allocated) gradually.

The story is similar for the entire SWD. The general principle is the third tracks of all bands should be delayed for use until the SWD is 75% full. Although several static LBA-to-PBA mapping schemes can be proposed using this principle, we will only present three representative new address mapping schemes which are indicated respectively by "R(4123)", "124R(3)" and "14R(23)".

4.5.2 Mapping Scheme "R(4123)"

Mapping scheme "R(4123)" maps LBAs to the tracks of all bands in a Round-Robin fashion. It maps the first 25% LBAs to the 4th tracks across all bands. Similarly, the second 25% LBAs are mapped to 1st tracks across all bands. The rest LBAs are mapped in the same Round-Robin manner to 2nd and 3rd tracks. Symbol "R" therefore means Round-Robin as a naming convention.

This mapping scheme makes sure no write amplification will be incurred when SWD usage is no more than 50%. When SWD usage becomes close to 75%, only 1 extra read and 1 extra write request will be incurred if an update request is made to the 1st tracks. However, SWD performance drops quickly when it is almost full.

4.5.3 Mapping Scheme "124R(3)"

Mapping scheme "124R(3)" is an alternate option, which maps the first 75% LBAs to the 1st, 2nd and 4th tracks in an ordered sequential manner but maps the rest 25% LBAs to the 3rd tracks in a Round-Robin fashion, as the name suggests.

This scheme preserves better LBAs spatial locality than scheme "R(4123)" so less seek overhead can be expected. However update requests may incur write amplification even when SWD usage is less than 50%. This actually indicates a tradeoff between amplification overhead and seek overhead.

4.5.4 Mapping Scheme "14R(23)"

This mapping scheme maps the first 50% LBAs to the 1st and 4th tracks in an ordered sequential manner and maps the next 25% LBAs to the 2nd tracks in a Round-Robin fashion. The last 25% LBAs will finally be mapped to the 3rd tracks in a Round-Robin fashion.

In terms of update performance, this scheme generally follows the prediction curve of "R(4123)"in Figure 4.2 but may perform slightly better when SWD usage is less than 50% because of a little better LBAs locality. The actual performance, however, also depends on the LBAs distribution in a given workload.

4.5.5 Performance Prediction for Updates

Assuming all factors are the same but the LBA-to-PBA mapping scheme difference, Figure 4.2 roughly predicts the average update performance for all the mapping schemes as the SWD space grows. This prediction will be validated later in our experiments.

4.6 Experimental Evaluations

The overall performance of SWDs with these new allocation schemes are evaluated with several realistic traces and then compared to that of a SWD with the conventional scheme and a regular HDD.

4.6.1 Enhanced DiskSim

We emulate the in-place update SWD with an enhanced DiskSim. We enhance the DiskSim with two components: one is the *address mapper* component and the other is the *write amplifier* component. The address mapper translates a given LBA into a PBA according to a specified static mapping scheme and the write amplifier converts a write/update request into a set of read and write requests if write amplification is incurred. Whether a write/update request will be amplified depends on the LBA and the current SWD usage.

We are simulating a SWD based on the parameters of an hp_c3323a disk drive in the DiskSim package. The SWD contains 3000 physical cylinders, each of which consists of 1000 blocks. Band size is 4 and write head width is 2. No obvious performance difference is observed when we configure to use 1 or 2 disk surfaces. The results we show below represents a single surface.

4.6.2 Traces

Four traces are used in our experiments, including one MSR trace (*web_ 0*)[87], one HP trace (*hp_ c2247*)[88], one Financial trace (volume 0 of *Financial2*)[89] and a synthetic trace (*SYN*). The characteristics of these traces, including inter-arrival time (IAT) and write ratio, are shown in table 5.2. Since write amplification is essentially caused by

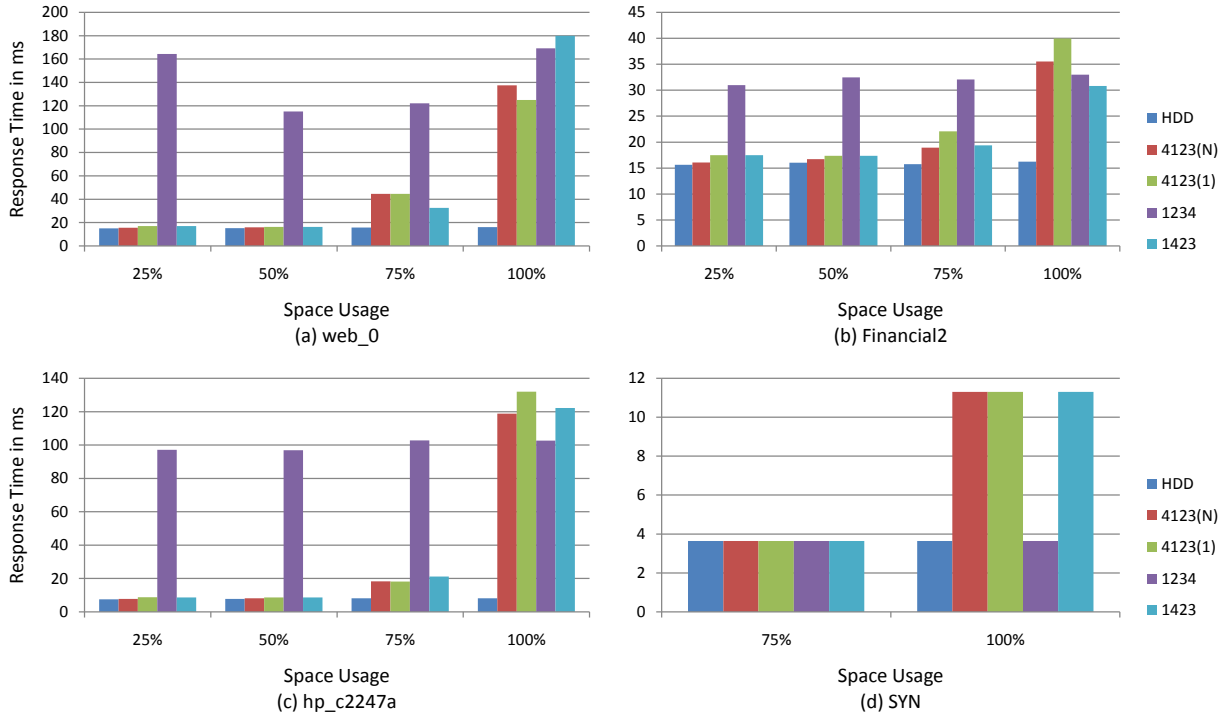


Figure 4.3: Average response time for four traces under different SWD space usages

update operations, these traces are picked according the write/update operation ratio[1] . For example, *web_ 0* is an update intensive workload, *hp_ c2247* is a moderate update workload, *Financial2* (read intensive) and *SYN* (cold sequential write) are light update workloads.

SYN is used to mimic a backup workload which continuously writes data to an empty SWD until the space is fully used. Therefore this is a cold sequential write workload with no update. Its average request size is 8 blocks and the inter-arrival time follows a normal distribution of which mean is 50 ms with standard deviation 10 ms.

4.6.3 Experiment Design

As Figure 4.2 indicates, update operation performance changes as the SWD space usage grows. We therefore choose 25%, 50%, 75% and 100% as the sampling points to make performance comparisons. The synthetic trace only requires 75% and 100% as sampling

[1] We logically convert writes into updates as shown in Section 5.3

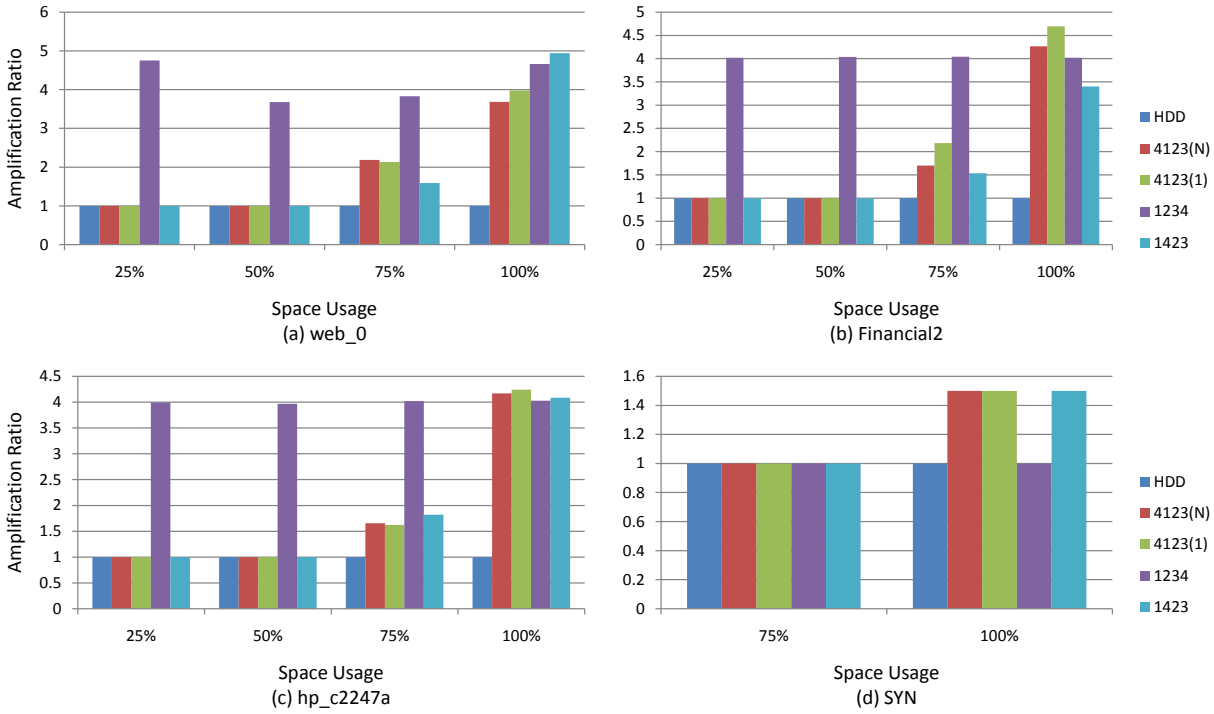


Figure 4.4: Write amplification comparison for four traces under different SWD space usages

points because none of the allocation scheme incurs write amplification overhead before 75% usage. We run 70 experiments in total, using different mapping schemes, different SWD space usages and different workloads combinations.

We run *web_ 0*, *Financial2* and *hp_ c2247* with our enhanced DiskSim 4 times and each time we pre-fill the SWD with data to a particular usage (i.e., 25%, 50%, 75% and 100%). This will logically convert all writes in the workloads into updates. These traces have to be adapted before input to the enhanced DiskSim. For example, LBAs larger than the specified SWD usage have to scale down with modulus operations. Besides, request arrival rate has to be scaled down in order not to saturate the emulated SWD because of two reasons. First, the traces we use represent workloads to the storage arrays with multiple HDDs which have much better performance than a single SWD. Second, write amplification in a SWD incurs extra read and write operations, which results in a much bursty workload to the SWD. Therefore, in our experiments, we increase the inter-arrival time by 200 times for web_0, similarly, 5000 times for *Finanical2* [2] and 5 times for *hp_ 2247*.

We run *SYN* twice. The enhanced DiskSim runs the *SYN* workload and writes data into an empty SWD until the SWD is 75% full in the first experiments. Data is continuously written into an empty SWD until space is 100% full in the second experiments. This is done to emulate a backup workload or cold sequential write workload.

4.6.4 Result Discussions

In this section, we make performance comparisons using average response time and average write amplification ratio.

When SWD Space Usage Is Less than 75%

The average response time for the four traces are shown in Figure 4.3 (a) (b) (c) (d) respectively. It can be observed that SWDs using the three new mapping schemes constantly outperform the SWD using "1234" scheme. The performance difference is especially significant for moderate update and update intensive workloads such as *hp_ c2247*

[2] *Financial2* has a huge variance in inter-arrival times. The workload is quite bursty from time to time but mean IAT is bigger than expected.

and *web_ 0*. Besides, SWDs using new mapping schemes can provide a similar performance to that of a regular HDD. Furthermore, "R(4123)" and "14R(23)" constantly outperform "124R(3)" for traces with updates when SWD space usage is no more than 50%, which indicates that write amplification overhead has more performance impact than seek overhead in our experiments. This is because write amplification incurs extra operations to the SWD, which increases the number of outstanding requests and consequently cause longer queuing time for other requests. In other words, the workload becomes more bursty because of the extra requests.

The performance difference can be well explained with the average write amplification ratio graphs as shown in Figure 4.4 (a) (b) (c) (d). For example, due to the nature of "1234" scheme, its average WARs always stay around 4 regardless of the SWD space usages and traces. Note that *SYN* is a special case because it is a cold sequential write trace and it contains no update request at all. Similarly, the average WARs for "124R(3)" always stay around 1.67 when SWD space usage is no more than 75%, the average WARs for "R(1234)" and "14R(23)" stay at 1 when SWD space usage is no more than 50% and become around 1.67 when SWD space grows to 75%. These observations are consistent with our theoretical performance analysis and prediction previously. A bigger average WAR simply means a more bursty workload is resulted.

SYN is used to show that for backup-like workloads, the "1234" scheme does outperform the new mapping schemes when SWD space usage is over 75% full, although nearly the same performance is achieved when usage is lower than 75%. This may indicate a possible future direction of adaptive mapping schemes for multiple workloads and multiple volumes case.

When SWD Space Usage Is Close to 100%

All SWDs produce an average WAR of 4 when SWD space usage is close to 100% regardless of the mapping scheme used. Therefore the performance drops quickly and significantly bigger response time can be observed. This implies that when space usage is over 75%, defragmentation should be performed to make more room in the 3rd tracks, which will practically make SWDs maintain good performance.

Another observation is that when SWD space usage is close to 100%, every mapping scheme including "1234" has a chance to win because the actual performance depends

on the LBAs distribution in the trace. For example, "R(1234)" works best for *web_ 0* but performs worst for *Financial2*. The reason is that more updates happen to take place in the 3rd and 4th tracks in *web_ 0* but more go to the 1st and 2nd tracks in trace *Financial2*.

4.7 Conclusions

In this chapter, we have presented several new address mapping schemes for in-place update SWDs. By appropriately changing the order of space allocation, the new mapping schemes can improve the write amplification overhead significantly. Our experiments with four traces demonstrate that new mapping schemes provide comparable performance to that of regular HDDs when SWD space usage is less than 75%.

Chapter 5

Out-of-place Update SWDs

Another approach to write amplification problem is the out-of-place update strategy, which means that data blocks will be written to new locations on updates and the original blocks will be invalidated. LBA to PBA mapping table has to be maintained to keep track the data movements. Besides, SWD will gradually become fragmented as the number of invalid data blocks increases. As a result, garbage collection operations are necessary to be performed to reclaim and reuse those invalid blocks.

In this chapter we introduce T-STL, a **T**rack-based **S**hingled **T**ranslation **L**ayer for autonomous Shingled Write Disks (SWDs). T-STL minimizes the write amplification overhead by utilizing two unique properties of SWDs to handle workloads differently according to the SWD space utilization. First, when SWD space utilization is less than 50%, T-STL turns the SWD into a HDD-like device by using every other track [1] . The unused tracks work as safety gaps to avoid data overwriting. Therefore, in-place updates are possible in this situation. The second property is a track-based mapping instead of a typical block-based mapping. When SWD utilization is over 50%, tracks that do not allow in-place updates are updated in a copy-on-write manner as out-of-place updates. In a track-based mapping, when a track is invalidated, it becomes free right away and can be immediately reused as long as the next track is free too, without triggering an explicit garbage collection (GC) operation. Only when the free SWD space becomes extremely fragmented, an explicit on-demand GC operation needs to be invoked

[1] We assume the write head is 2-track wide in our discussions. However, our schemes can be adapted for bigger write head width.

to create big contiguous free space by migrating some valid tracks.

Efficient track-level mapping and space management schemes are also designed to fully utilize these two properties. We implement the T-STL scheme and compare it with a regular HDD, an existing out-of-place update SWD (O-SWD) design and an in-place update SWD (I-SWD). The experiments with several realistic traces and one synthetic trace demonstrate that the T-STL scheme can perform much better than the existing SWD designs and even nearly as good as regular HDDs when SWD space usage is less than 50%.

5.1 Introduction

The decision on design tradeoffs are often based on many factors such as engineering difficulties, drive reliability, workload characteristics and storage requirement. As discussed in the previous chapter, I-SWDs have the advantage of no mapping table and garbage collection which reduce the potential reliability issues associated with all kinds of additional metadata management. On the other hand, O-SWDs have minimal cost on safety gaps and provides great space gain.

Similar to Solid State Drives (SSDs), the operation principles of out-of-place updates make mapping table and garbage collection algorithms are necessary for SWDs. One or more mapping tables can be used which tracks data movements caused by either new data writing, data updating or garbage collections. Garbage collections are used to reclaim the invalid data blocks or sectors created by out-of-place updates. Different from SSDs, tracks in SWDs do not have to be "erased" before being reused. In SSDs, however, blocks must be erased and turned into clean blocks before being reused due to the nature of flash. This unique feature of SWDs is one of its properties that we utilize to design our scheme.

In SSDs, a firmware layer called Flash Translation Layer (FTL) is used to manage the data mapping, garbage collection and wear-leveling. The three functions are tightly coalesced and data mapping level is usually the cut-in angle to compare different FTL schemes. Choices for FTL mapping levels include page level, block level and hybrid level. Nevertheless, FTLs can hardly be applied directly to SWDs without drastic modifications due to the intrinsic media properties.

Simiarly, data mapping for SWDs can be done at different granularities such as block or sector level, track level, S-block level [80] and band level. Block level mapping is generally not preferred as it generates a huge mapping table and it turns logically sequential reads into physically random reads, which performs awfully due to the high seek overhead in hard drives. Band level mapping, on the other hand, requires a much smaller mapping table. But it is very inflexible because it has high copy-on-write overhead. S-block level mapping and track level mapping are therefore more promising levels to design a Shingled Translation Layer (STL).

In order for O-SWDs to be adopted in the current storage systems, metadata overhead and GC overhead must be minimized. In this chapter we propose the T-STL scheme, a track-based shingled translation layer for autonomous SWDs. T-STL is motivated by two unique properties of SWDs. First, an SWD can be turned into a HDD-like device when the space utilization is less than 50% by using only every other track. The unused tracks serve as safety gaps to prevent destroying tracks with valid data. As a result, in-place updates can be performed in the used tracks. Second, an invalidated track in SWD is essentially a free track and can be immediately reused as long as the next track is free too. T-STL therefore takes advantage of this property to adopt an aggressive track update strategy which maintains a loop of track invalidation and reuse without triggering explicit on-demand GC operations. This greatly reduces the frequency of on-demand GC operations which are invoked only when the free SWD space becomes extremely fragmented.

Two major modules are designed in T-STL in order to fully exploit these two properties. One is a track level LBA-to-PBA mapping and the other is an efficient SWD space management scheme. The SWD space management module includes free track selection and on-demand GC operation. During a GC operation, valid tracks are migrated in an efficient way to create bigger contiguous free space.

We implement the T-STL scheme and compare it to a regular HDD, an existing O-SWD design (S-block based indirection system) [80] and an I-SWD scheme [76]. The experiments with several workloads demonstrate that the T-STL scheme can perform much better than the existing schemes.

The remainder of the chapter is organized as follows. The T-STL scheme is described in Section 5.2. Experiments and evaluations are presented in Section 5.3 and some

conclusion is made in Section 5.3.4.

5.2 The T-STL Scheme

In this section, we describe the T-STL internals. Although T-STL follows the O-SWD layout, it does not use an E-region and it allows in-place updates whenever possible. There are two function modules in T-STL: the LBA-to-PBA mapping (Section 5.2.2) and the space management scheme(Section 5.2.4).

5.2.1 Aggressive Track Update

T-STL handles update requests aggressively when SWD space usage is over 50%. If an update request is made to a track whose next track is free, then T-STL performs this update request in-place. However, if the next track is not free, T-STL will amplify this request into a track update even if this update request modifies only a partial of a track. In this case, T-STL will read the existing data on this track, modify the blocks in memory, write the track to a new track position and invalidate the original track. In other words, the track is updated in a copy-on-write manner. Note that a track can only be updated to a new position inside the same band due to track size differences from band to band. The mapping table will be updated accordingly after the migration.

Track update has been proven to be beneficial and affordable because it creates a track invalidation and reuse loop. When a track is invalidated, it actually becomes a free track and can be reused as long as its next track is free or as soon as its next track becomes free. As a result, track updates in T-STL continuously invalidate tracks and turn them into free tracks without triggering explicit on-demand GC operations. This greatly reduces the frequency of invoking on-demand GCs which compensates for the cost of the track update operations. Explicit on-demand GC operations are only invoked when the free SWD space becomes too fragmented as discussed in Section 5.2.4.

5.2.2 Track Level Mapping Table

The first main functionality of T-STL is the LBA-to-PBA mapping at a track level which enables SWDs to talk to upper level applications such as the file systems using the block interface. Therefore SWDs can be used in existing storage systems in a drop-in manner.

There is only a single track level mapping table for the whole SWD. The mapping table is updated when: 1) an used track is updated to a new location, 2) used tracks are migrated during an on-demand GC or 3) tracks are allocated for new data.

Given an LBA, T-STL will first calculate its corresponding logical track number (LTN) and its offset inside this track, based on the number of bands and the track size in each band. It then looks up the mapping table and translates LTN into physical track number (PTN). The final PBA can be easily computed with PTN and the offset inside the physical track.

Assuming an average track size of 1 MB, a 6 TB SWD requires at most a 48 MB track level mapping table (assuming 4 bytes for each LTN or PTN). Besides, the mapping table is empty in the beginning and gradually grows as the SWD space utilization increases. Considering the fact that the standard DRAM size for a 6 TB HDD on the market today is 128 MB, we claim the metadata overhead of the track level mapping table is reasonably low.

5.2.3 Space Elements

There are two types of tracks in an SWD: the used (or valid) tracks and the free tracks. When a used track is invalidated, it becomes a free track but it is not considered as a usable free track if its following track is not free. However, a free but unusable track can at least serve as a safety gap and allow its preceding track to be updated in place.

All the used tracks constitute the **used space** and all the free tracks constitute the **free space**. We call a group of consecutive used tracks or free tracks a *space element* which is used to describe the current track usage. For example, considering Figure 5.1, we say the used space includes elements [0, 1, 2, 3], [6], [10, 11, 12], [14, 15, 16, 17] and [20, 21] while the free space includes elements [4, 5], [7,8,9], [13], [18, 19] and [23, 24]. The **size** of a particular space element is defined as the number of tracks in it. The last track in each free space element is not usable and can not be written because writing to this last track will destroy the valid data on the following track which is a used track. Particularly, free space element of size 1 contains no usable free track such as element [13]. However, the number of elements and their sizes continuously change as incoming requests are processed. A free track that is previously unusable can become usable later as soon as its following track becomes free too. Accordingly the last track in a used

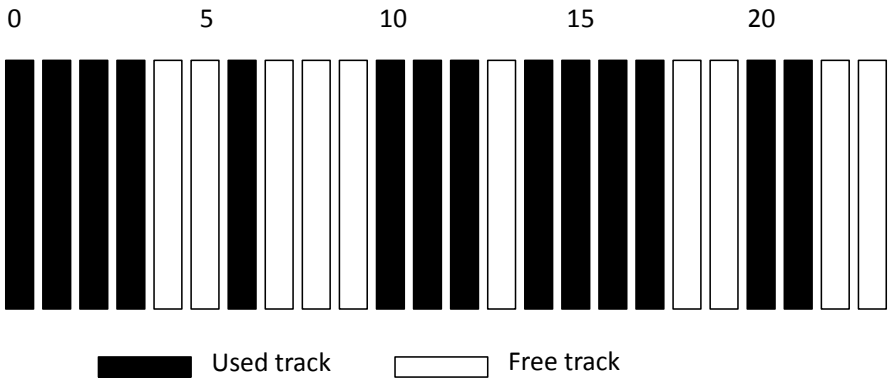


Figure 5.1: SWD Usage State

space element can be updated in place because its next track is a free track.

5.2.4 SWD Space Management

The second main functionality of T-STL is SWD space management which is responsible for free track selection and free space consolidation. Free space consolidation is essentially the on-demand GC operation for SWD that is mentioned previously. Space management is done for each band separately.

We first present a simple space management scheme named as "Greedy" and then describe another scheme called "Smart" in Section 5.2.4 which better utilizes the track level mapping strategy. Smart supports automatic cold data [2] progression and reduces GC overhead.

Free Track Selection

Free track selection means that when a track is updated, T-STL has to choose a new track position inside the same band from the available usable free tracks.

To choose new track locations upon track updates or writes, T-STL based on Greedy will search starting from the current SWD write head position in a greedy manner. It tries to find the nearest free space element with a similar size to the request size. If no free space element is found to be big enough to accommodate the data, multiple free

[2] We define the cold data as less frequently or not recently updated data.

space elements will be used with each selected in a greedy manner.

Fragmentation Ratio

We define the free space fragmentation ratio to help decide when to invoke on-demand GC in each band. Assuming the total number of **free tracks** in a selected band is F and the total number of **free space elements** is N, the free space *fragmentation ratio* (R) for this band can be computed according to Equation 5.1. In fact, the fragmentation ratio is the percentage of usable free tracks in all the free tracks. Fragmentation ratio of 0 means the free space is too fragmented. In fact, 0 means all free space elements are of size 1 and thus no track can be used.

$$R = \frac{F - N}{F}, where\ 1 \leq N \leq F \tag{5.1}$$

A big R ratio is not suggested either since frequent unnecessary GCs will be invoked even though the free space is not fragmented, which harms SWD performance. Furthermore, a larger ratio means a smaller N and thus a smaller number of tracks that support in-place updates. Accordingly a smaller ratio may suggest a bigger N and thus a bigger number of tracks supporting in-place updates. Our permutation tests suggest that 0.5 is a good option for the simulated SWD [3] . This way, T-STL can maintain relatively big contiguous free space and trigger a GC only if necessary.

Free Space Consolidation (GC)

The fragmentation ratio will be checked upon each incoming write request. If it is equal to or smaller than the threshold value, an on-demand GC operation will be invoked in the targeted band to migrate used tracks and combine the small free space elements into bigger elements. This will improve I/O performance for big writes and updates, as well as increase usable free space.

The GC operations move small used space elements and append them to nearby used space elements. Starting from the current SWD head position, T-STL based on Greedy will search in both directions (i.e., left and right) for the nearest used space element of

[3] Different physical drive layouts require different thresholds for best performance. A permutation test can help decide the value for a particular layout.

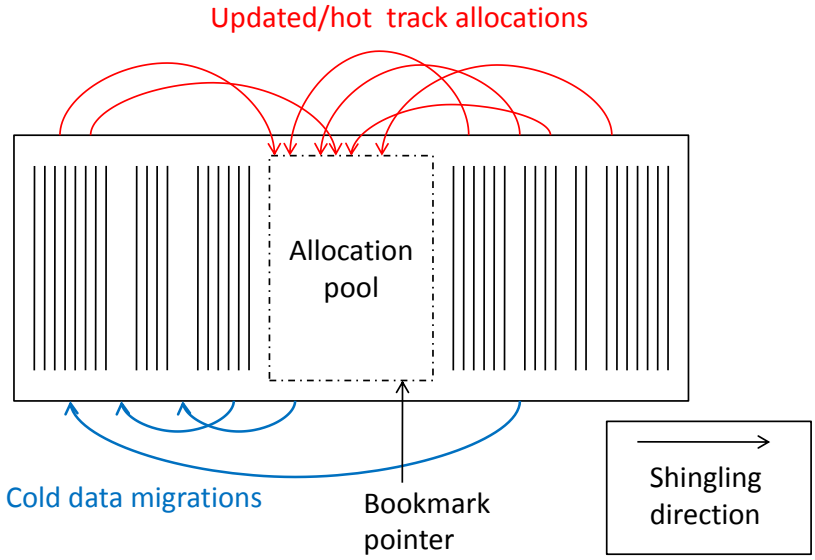


Figure 5.2: The Process of Smart

size smaller than a threshold W. W is initialized to be a small value based on the current SWD space usage U. It can be calculated according to Equation 5.2. In fact, W has a practical meaning and it stands for the used space to free space ratio. The theory behind this equation is that the average used space element size is bigger when the SWD space utilization is higher. For example, W will be initially set to 2 if the current SWD usage is 60% or 4 if SWD usage is 80%.

If no element is found to be smaller than W, then W will be doubled and T-STL will redo a search based on the new W. This will be repeated until a satisfying element is found.

T-STL will then read this element and append it to the nearest neighboring element. Usually the free space consolidation will immediately stop after one element has been moved so as to minimize the overhead of a single GC operation. This will therefore minimize the performance interference on serving the following requests. The fragmentation ratio sometimes remains below the threshold after the current GC operation. The next GC operation in this band will continue to improve the fragmentation. Multiple elements movement happens only when there is not enough usable free tracks to accommodate the updated track(s) or new files, which is infrequent in our experiments.

In Figure 5.1, R is 0.5 by Equation 5.1 and W is 2 by Equation 5.2. Assuming an on-demand GC is triggered and the current write head is at track 5, T-STL will select used space element [6] as the victim and append it to element [0, 1, 2, 3] since it is closer than appending to element [10, 11, 12]. Consequently, free space element [4, 5] and [7, 8, 9] will be merged into a single bigger element [5, 6, 7, 8, 9]. The resulting fragmentation ratio R is 0.6 (= (10-4)/10). T-STL will detect upon the next request that R is above the threshold and thus it will not invoke another GC operation.

$$W = \lceil \frac{U}{1-U} \rceil, where\ 0 < U < 1 \tag{5.2}$$

The Smart Scheme

Track level mapping provides a good opportunity of automatic cold data progression. This is achieved and embedded into the free track selection and free space consolidation of the Smart scheme, without using dedicated hot/cold data identification algorithms and data migration schemes.

Smart maintains a dedicated track pointer called "*bookmark pointer*" for each band. Assuming the shingling direction is from left to right, a bookmark pointer initially points to the leftmost free track of the largest free space element (named as "allocation pool"). The free tracks in this allocation pool are allocated to accommodate updated tracks in a sequential manner. A modified track is always written to the free track pointed by the bookmark pointer, which will be incremented accordingly. After all the usable free tracks are consumed, Smart will select the latest largest free space element to be the next allocation pool and update the bookmark pointer.

Free space consolidation or GC in Smart is triggered in the same way as the greedy T-STL and the victim used space element is also chosen in the same way. However, Smart always migrates a victim element to the leftmost free space element larger than the victim element[4] instead of appending it to the nearest neighbour.

Smart is illustrated in Figure 5.2. The allocation pool accumulates the recently updated tracks or hot tracks and naturally organizes them in large used space elements which are less likely selected as victims during GC operations. As a result, it is the cold data that mostly gets migrated against the shingling direction to the left by GC

[4] The used space element containing track 0 is a special case.

operations. Eventually the cold data will stay untouched at the left side of the bands and hot data gets updated and pushed to the right side of the bands which greatly reduces unnecessary cold data movements during GC operations.

5.2.5 T-STL for Cold Workload

To adapt for cold workload such as backup and archive workload, T-STL only needs to disable the alternating track strategy and enters the aggressive update mode directly. As a result, data will be written sequentially into the SWD just like a regular HDD and updated in the "copy-on-write" manner as needed. No changes to the space management scheme are needed.

5.2.6 T-STL Reliability

Since the mapping table is loaded and operated in the DRAM, it can be lost during a power failure. This problem can be solved by periodically checkpointing the mapping table to a copy in the random access zone (RAZ) on the SWD. During the checkpointing, all the dirty or updated mapping entries will be synchronized to the SWD.

Upon the completion of each checkpointing, T-STL will record the timestamp which will be used during recovery. Free track allocation and space management are the same as the basic T-STL schemes (Greedy and Smart) except that when writing a new track, a backpointer to the logical track number (LTN) along with the current timestamp will be stored together with the associated physical track. We assume there will be some tiny spare space associated with each physical track that can be used to store the LTN and the timestamp. Otherwise, we can simply reserve a sector or block in each physical track to be used for storing the LTN.

In order to recover from a power failure, T-STL will scan all the timestamps and identify those newer than the timestamp of the latest checkpointing. These newer timestamps indicate these tracks are updated or written after the latest checkpointing which are not reflected in the mapping table in RAZ yet. T-STL then reads their associated LTNs and PTNs to construct the corresponding LTN-to-PTN mapping entries which will be merged to the mapping table copy in RAZ so that the latest LTN-to-PTN table will be restored.

Further improvement for the recovery time can be achieved by pre-allocating free space to be used between two successive checkpointings [90]. This is a perfect for Smart scheme because Smart always reserves an allocation pool in advance. Applying the scheme here, a certain number of free tracks will be pre-allocated in each band whose physical track numbers (PTNs) will be recorded and stored in a secure place. Note that on-demand GC now can only be performed for tracks not pre-allocated. To recover from a power failure, only the pre-allocated tracks or the allocation pools have to be scanned which greatly reduces the recovery time.

5.3 Evaluations

In this section, we compare the performance of the T-STL scheme (based on Greedy and Smart respectively), an existing O-SWD design (S-block based Indirection System or IS), an I-SWD scheme and a regular HDD using several workloads.

5.3.1 T-STL Implementation

We implement these schemes on top of Disksim [88] to simulate SWDs and we simulate an SWD based on the parameters of a Seagate Cheetah 15,000 RPM disk drive [91]. This is the newest validated disk model that is available to Disksim. It has a capacity of 146GB (based on 512KB sector size) and 16MB of on-disk cache. We divide the total capacity into 3 parts: one 2GB randome access zone, one 2GB E-region and the rest 142GB for persistent storage space.

The random access zone and the E-region are left untouched if they are not used in a specific scheme. Specifically, both of the random access zone and the E-region are not used in HDD. And the E-region is not used in I-SWD, O-SWD.

5.3.2 Schemes to Be Compared

Since our objective is to design SWDs that can perform well under general workloads (not only for cold workloads) and can be used in existing storage systems, we choose the regular HDD as the baseline for comparison. We are hoping the performance of the new designs can be close to that of HDD.

Table 5.1: Tested Schemes

Scheme	Type	Effective Capacity	Band Size
HDD	HDD	140GB	N/A
Greedy	O-SWD	140GB	100
Smart	O-SWD	140GB	100
R(4123)	I-SWD	112GB	4
IS	O-SWD	140GB	N/A

We then choose the "R(4123)" scheme for I-SWD [76] as the second competitor. Band width is set to 4 and the write head width is set to 2. The main idea of "R(4123)" is to reduce the write amplification overhead for writes or updates by changing the order of utilizing the tracks. "R(4123)" statically maps the 4th tracks in all bands to the first 25% LBAs, the 1st tracks to the second 25% LBAs, 2nd tracks to the next 25% and finally the 3rd tracks to the last 25% LBAs. This scheme can performs as good as a regular HDD when effective SWD usage is no more than 50% but its performance drops quickly when usage is larger than 75% of the effective SWD space capacity. Due to space cost on safety gaps, the effective storage capacity of an I-SWD is 112GB[5] .

We also compare to an existing O-SWD design, the S-block based indirection system (IS) proposed in [80]. E-region in IS is named as cache buffer and I-region is named as S-block buffer, both of which are organized in a circular log fashion. IS uses block-level mapping for the cache buffer and S-block level mapping for the S-block buffer. IS also adopts three types of garbage collections algorithms: `cache_buffer_defrag`, `group_destage` and `S_block_buffer_defrag`. As the names suggest, `cache_buffer_defrag` manages the garbage collections in cache buffer, `group_destage` deals with the valid data migration from cache buffer to S-block buffer and `S_block_buffer_defrag` garbage collection the S-block buffer.

The original IS frequently saturates the underlying Disksim in our tests due to unnecessary valid block movements in the E-region. It destages data from E-region to I-region only when all blocks in the E-regions are valid. This can be improved by triggering data destaging once we detect that the total eligible blocks (invalid blocks plus the free space between head and tail pointers) is less than the sum of write request size and the safety

[5] $142GB \times 0.8 = 113.2GB$. We use 112GB for simplicity.

Table 5.2: Trace Statistics

Trace	I.A.T. (ms)	Average B.D.	MAX LBA	Average R.S.	Write Ratio
mds_0	499.41	968339427.25	36417159175	18	0.8811
usr_0	270.25	1682254833.62	17077784583	45	0.5958
stg_0	297.79	1332575418.89	11612643455	23	0.8481
web_1	3757.39	2627690359.97	72831761927	58	0.4589

gap size between head and tail pointers. We use this improved indirection system as the third competitor.

Information about all the tested schemes are summarized in Table 5.1 including the scheme name, the corresponding type of SWD the maximum effective capacity used (in GB) and the band size (in tracks if applicable).

5.3.3 Experiment Design

Four realistic MSR traces [87] and one synthetic trace are used in our experiments. The characteristics of the MSR traces are shown in Table 5.2 which include the average inter-arrival time (I.A.T.), the average block distance (B.D.), the maximum LBA, the average request size (R.S.) and the write ratio.

MSR traces were captured on storage arrays of multiple modern HDDs, so the inter-arrival time of these traces will have to be scaled properly. We increase the inter-arrival time by 10 times for all the MSR traces in our experiments. Besides, LBAs beyond the current SWD usage also have to be scaled down with modulus operations according to the SWD space utilization.

The synthetic trace (SYN) is generated to mimic a backup workload that continuously writes sequential data to the SWD. Its average request size is 8 blocks and the inter-arrival time follows a normal distribution of which the mean is 5ms and the standard deviation is 2ms.

5.3.4 Result Discussions

We use *overall average response time*, *write response time breakdown* and *write amplification ratio* as the main performance measurements. The overall average response time

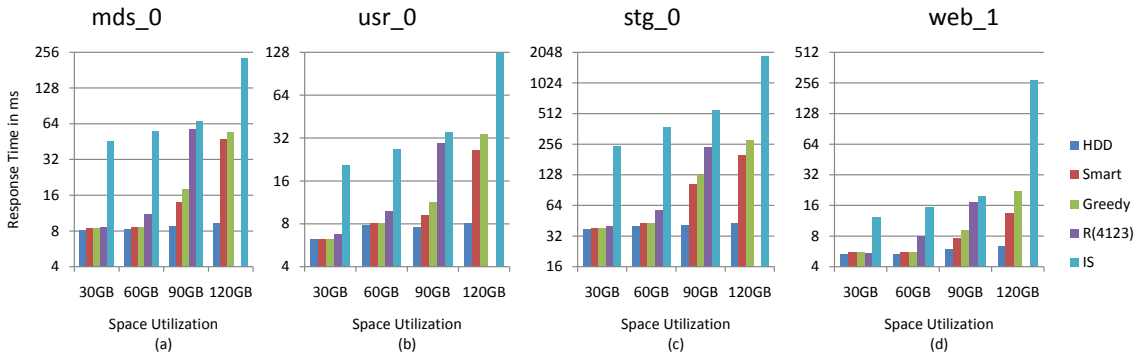


Figure 5.3: Average Response Time Comparisons

for different schemes under different workloads at different SWD space utilizations are shown in Figure 5.3. X-axis is the space utilization measured by GB and Y-axis is the overall average response time in *milliseconds* at **log scale**. Note that the total effective capacity of an I-SWD is only 112GB and therefore there is no plot for I-SWD at 120GB.

We also provide write response time breakdown in the case of 90GB space usage in Figure 5.4 to further understand the write amplification overhead and overall write performance. Unlike 30GB and 60GB space usage where the T-STL schemes do not invoke any GC operation, all schemes except the regular HDD are experiencing performance penalties caused by GCs or write amplifications when the space usage is 90GB. Y axis in the figure stands for the gross write response time which is the time between a write request getting queued and getting completed. Gross write response time consists of two parts: the write amplification overhead and the actual time to write the data. This is because if a write request triggers a GC operation, it will not be serviced until the GC completes.

Write amplification ratio (WAR) is defined as the ratio between the total number of incurred operations and the actual number of write operations. The results for WAR is shown in Figure 5.5. Since all schemes except IS have a ratio of 1 at 30GB space usage and I-SWD does not have a plot at 120GB space usage as explained previously, we only show the results at 60GB and 90GB space utilizations. WAR can partly explain the overall average response time results and the overhead associated with writes. Generally, a larger WAR value implies worse performance. Of course, more factors such as request size distribution and spatial locality of the extra operations incurred by GCs need to be

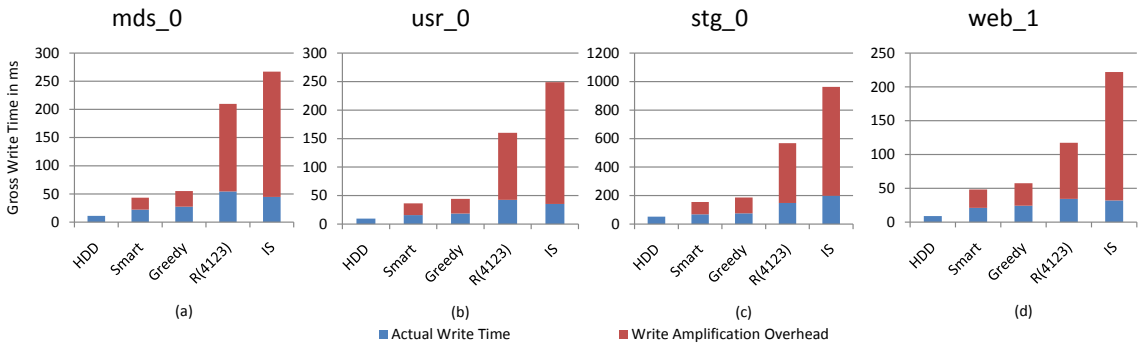


Figure 5.4: Gross Write Response Time Breakdown at 90GB Utilization

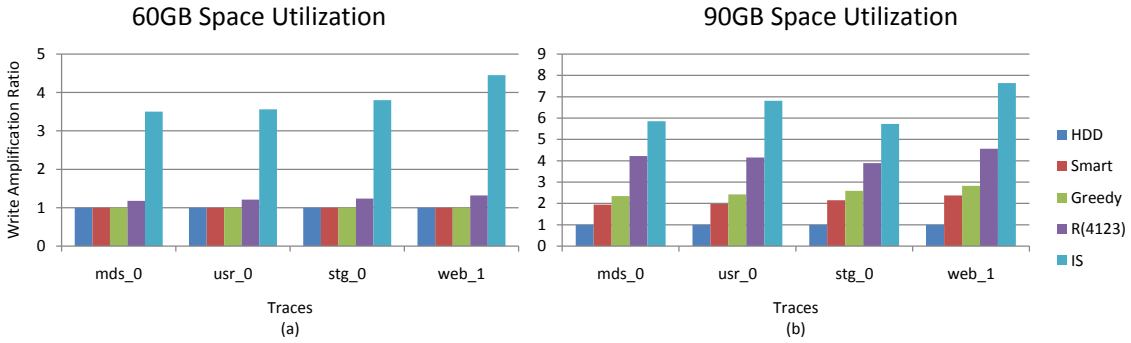


Figure 5.5: Write Amplification Ratio at Different Space Utilizations

considered for a completely comprehensive explanation.

Smart vs. HDD

Among all the SWD designs, Smart scheme provides the closest performance to HDD. Figure 5.3 shows that Smart can achieve HDD-like performance under MSR traces (representing primary workloads) when space utilization is no more than 50% or 70GB. When tested with the synthetic workload SYN, it constantly provides the same performance as HDD at all the tested space utilizations.

Smart vs. Greedy

These two schemes act exactly the same except that Greedy uses simple greedy algorithms for its space management while Smart better exploits the track level mapping.

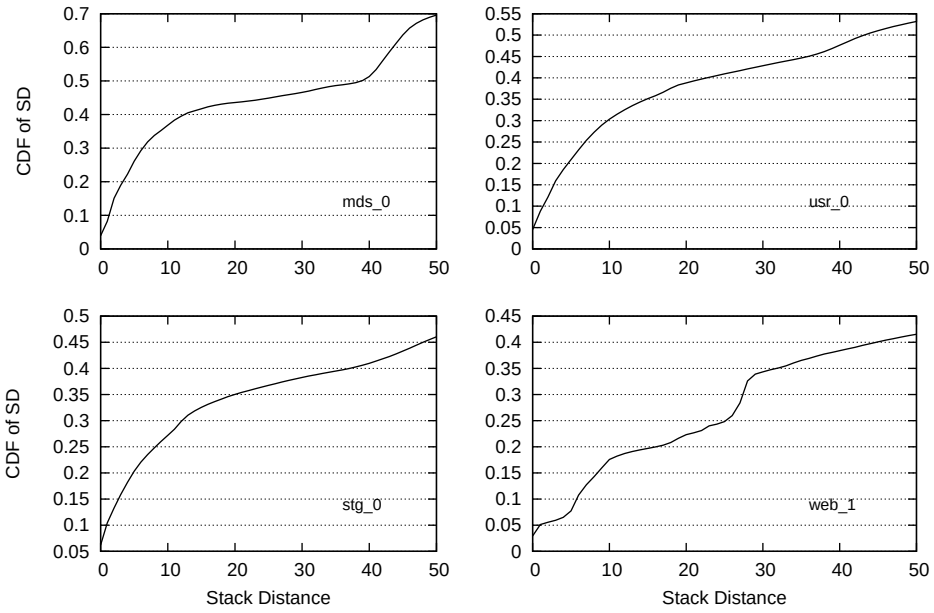


Figure 5.6: Stack Distance for Track Updates

By gradually separating the cold data from hot data and migrating the cold data to the left side of the bands, Smart reduces unnecessary cold data movement during GC operations and consequently the GC cost. We characterize all the available MSR traces and find that all of them exhibit a good degree of temporal locality measured by the stack distance between two successive updates to the same track, which indicates Smart is suitable for these primary workloads. The temporal localities of the tested MSR traces are shown in Figure 5.6. According to Figure 5.3, Smart improves the performance by up to 25% when space utilization is over 50% or 70GB.

To exam the effectiveness of Smart for workloads with low temporal locality, we test the two scheme using a synthetic workload with a uniform request offset distribution. In other words, data is evenly accessed and there is no hot or cold data. Again, its average request size is 8 blocks and the inter-arrival time follows a normal distribution of which the mean is 5ms and the standard deviation is 2ms. The result shows that the average response time for Smart is about 5% longer at 90GB and about 6% longer at 120GB. The main reason is that Smart incurs larger seek distances during GC operations while

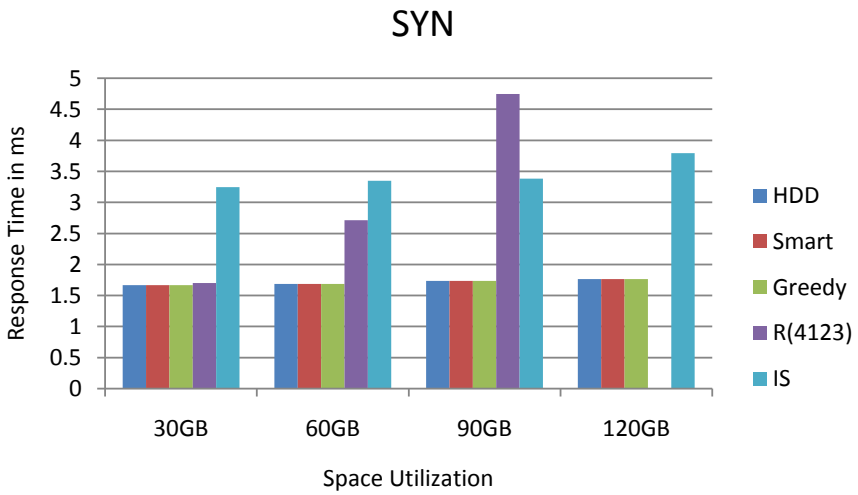


Figure 5.7: Performance Under SYN

Greedy only needs to append a victim to its nearest neighbour.

Smart vs. R(4123)

Our results show that the performance of Smart is more sustainable and better than R(4123) after 30GB space utilization. This is mainly because R(4123) provides less effective space than a Smart SWD and it suffers write amplification overhead much earlier. For example, 60GB is over 50% of a R(4123) based I-SWD ($112GB \times 0.5 = 56GB$) and thus it starts to incur write amplifications while 60GB is still less than 50% of a Smart SWD. This can also be verified by Figure 5.5 where the WAR for R(4123) is already around 1.3 while Smart is still 1 at 60GB space utilization. This difference is even more obvious when the space utilization is 90GB in our results.

Despite the fact that an I-SWD provides much less effective storage space and suffers write amplifications earlier, the bottom line is that it does not require any metadata such as a mapping table.

Smart vs. IS

Figure 5.3 show that Smart can perform much better than the improved S-block based indirection system (IS) under all space utilizations. The reasons are as follows.

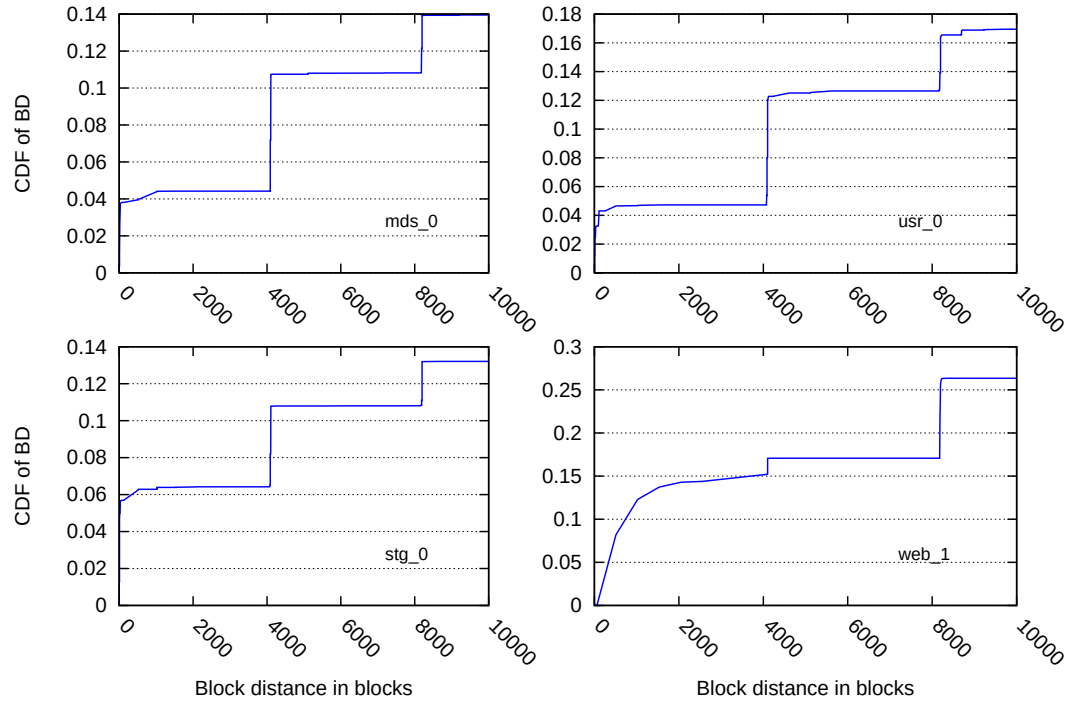


Figure 5.8: Spatial Localities of Write Operations

First, due to the S-block organization, the IS scheme is not able to take advantage of using alternate tracks when the space usage is no more than 50%. This means that IS has to handle write amplifications since an empty drive.

Second, IS uses an E-region in the hope of accumulating small writes in it and later destaging bulk data in the form of a (partial) S-block to the I-region during E-region GCs. However, E-region GCs themselves come with performance overhead, let alone small writes that belong to the same S-block are most likely scattered in the E-region and require multiple reads. In fact, whether an E-region will eventually benefit the overall performance depends on the workload locality.

Our workload characterizations on the MSR traces find that these traces contain a decent degree of spatial locality. Figure 5.8 plots the block distance between writes. A distance of 10000 blocks is considered small enough when compared to the used Seagate drive capacity. About 15% writes are within a distance of 10000 blocks for hm_0, usr_0, stg_0 and about 30% for web_1. However, the performance data in Figure 5.3 shows

Table 5.3: Scheme Comparison Summary

Schemes	E-region?	Performance	Sustainability	Metadata Overhead	Space Gain
HDD	no	best	best	no	best
Smart	no	good	good	low	good
O-SWD	no	good	good	low	good
I-SWD	no	good	fair	no	fair
IS	yes	not good	not good	high	good

that IS is still far behind Smart. The write amplification ratio of IS is also larger than Smart according to Figure 5.5. This indicates that accumulating small writes in the E-region does not offset the overhead of E-region GCs in our experiments.

Besides, E-region requires a block level mapping table to track the blocks in it. A 6GB E-region (0.1% of a 6TB SWD) produces a mapping table of 96MB which imposes additional challenges to the DRAM resource and metadata management. This is another major reason that we do not use an E-region in T-STL.

Comparison Summary

We summarize the comparison results in Table 5.3. The schemes are compared according to different metrics including performance, performance sustainability and metadata overhead. Smart is the most promising design which achieves a good balance among the listed metrics. The table also shows that all the SWD designs are impacted by the space utilization and their performance drops as the utilization grows (either faster or slower), which needs to be further improved in the future studies.

5.4 Conclusion

In this chapter we propose a T-STL scheme for efficient autonomous SWDs by exploiting two unique properties in the SWDs. This new scheme performs copy-on-write updates only when in-place updates is impossible. The track level mapping, when combined with a novel space management scheme (Smart) can automatically filter the cold data. The Experiments with realistic workloads and a synthetic workload demonstrate that the T-STL scheme can perform as good as regular HDDs under backup-like workload and

even under primary workloads when space usage is less than 50%.

Chapter 6

Conclusion and Discussion

An increasing number of organizations and researchers are migrating their big data workloads to HPC systems for higher data processing efficiency. Converged HPC systems emerge to combine Hadoop/MapReduce framework with HPC system infrastructure. Therefore, a mix of HPC workloads and big data workloads can be running in the same converged HPC system. There are two major I/O and storage requirements in these converged HPC systems: parallel I/O performance and storage capacity.

Parallel I/O performance is becoming more challenging as high performance computing techniques continuously evolve. Scalable and sustainable file system and storage systems must be designed to bridge the speed gap between the demanding CPU throughput and slower storage device performance. Efficient tools of I/O workload characterization and generations are therefore needed to help redesign and tune these systems.

Therefore in this work, we propose a complete solution (called PIONNER) to parallel I/O workload characterization and synthesizing which helps HPC system researchers and developers understand parallel I/O workloads better. Unlike existing work, we deal with several characteristics and challenges of parallel I/O workloads, including inter-process correlations, I/O library complexities and dependencies, as well as specific file access patterns. In PIONEER, we first condense a given original parallel I/O workload into a generic workload path by exploiting the inter-process correlations. Then we characterize and model the resulting generic workload path to extract a set of parameters describing all kinds of I/O characteristics. During this process, we build enforcement rules to preserve I/O request dependencies and we model file access patterns by profiling file

open sessions. Next, we use the extracted characteristics to construct a synthetic generic workload path based on desired parameter settings. PIONEER also includes a workload generation engine that can expand the synthetic generic workload path into a complete parallel I/O workload for a desired number of processes.

Parallel I/O performance is challenging for HPC systems mainly because of many factors along the parallel I/O path. In this work, we motivate ourselves by investigating the parallel I/O stack and exploring the correlations among factors such as file access pattern, parallel I/O modes and specific system parameters. We also propose a parallel I/O model which takes into account multiple I/O factors including request size, I/O access pattern, striping information to shed light on optimizing parallel I/O performance. This model can also be used by other researchers and developers to facilitate their work. Based on this knowledge, we propose IO-Engine, an intelligent I/O middleware module instrumented to the existing MPI-IO library that can transparently optimize HPC I/O workloads in Lustre system. IO-Engine can be extended to other parallel file systems using similar investigations. Moreover, IO-Engine can be further enhanced with a certain amount of future I/O access pattern or workload knowledge, which can be either provided by the developers/users or learned based on workload history using machine learning techniques.

Another major function of converged HPC systems is high performance big data analysis and one fundamental challenge in this aspect is the rapid data growth. The demanding storage capacity requires new storage techniques to break the areal data density limit in the traditional perpendicular magnetic recording HDDs. SMR is the most promising technique among several possible technologies due to its similar manufacturing process to the traditional HDDs. SMR increases the areal data density by overlapping neighbouring tracks and packing more tracks into disk platters with the same physical dimension. The nature of shingling prefers writes to be done physically sequentially which avoids overwriting tracks. In order to support random writes, two general approaches are investigated including the in-place update and out-of-place update, both of which have to handle the write amplification problem.

In this work, we first propose several new static address mapping schemes for in-place update SWDs. Tracks in an SWD device are organized into logical concepts called "Bands". One band is a group of neighbouring tracks. In-place update SWDs usually use

a smaller number of tracks per band to achieve a balance between space efficiency and overall performance. By appropriately changing the order of track allocations, the new mapping schemes can reduce the write amplification overhead significantly compared to a traditional track mapping scheme. Our experiments with four traces demonstrate that new mapping schemes provide comparable performance to that of regular HDDs when SWD space usage is less than 75%.

Next, we propose a T-STL scheme for efficient autonomous SWDs by exploiting two unique properties in the SWDs. This is motivated by the fact that in-place update SWDs spend considerable amount of space on safety gaps that prevent writing to the last track of a previous band from overwriting the tracks of the next band. T-STL solves this problem by adopting an out-of-place oriented approach and using large bands. This new scheme performs copy-on-write updates only when in-place update is impossible. The track level mapping, when combined with a novel space management scheme (Smart) can automatically filter and migrate cold data to minimize garbage collection overhead. The Experiments with realistic workloads and a synthetic workload demonstrate that the T-STL scheme can perform as good as regular HDDs under backup-like workload and even under primary workloads when space usage is less than 50%.

All the SWD schemes proposed in this work are designed for autonomous SWDs because they can be incorporated into existing storage systems in a drop-in manner. On the other hand, more research are needed to investigate the Host-managed and Host-aware SWDs because it is believed that disk I/O performance can be maximized when the host knowledge on workload characteristics are utilized for data management for SWDs. The challenges of Host-managed and Host-aware SWDs lie in offloading the physical data management from the drives and incorporating into upper software layers. Typical software layers include file systems, databases and software RAID. To accomplish this, an industrial standard must be proposed and agreed, new device commands need to be defined in order to take advantage of SWD characteristics, and the software in the I/O stack directly above SWDs must be revamped.

References

[1] Seref Sagiroglu and Duygu Sinanc. Big data: A review. In *Collaboration Technologies and Systems (CTS), 2013 International Conference on*, pages 42–47. IEEE, 2013.

[2] Infographic. `https://www.domo.com/blog/2014/04/data-never-sleeps-2-0/`.

[3] Jeffrey Dean and Sanjay Ghemawat. Mapreduce: Simplified data processing on large clusters. In *Proceedings of the 6th Conference on Symposium on Opearting Systems Design & Implementation - Volume 6*, OSDI'04, pages 10–10, Berkeley, CA, USA, 2004. USENIX Association.

[4] Infographic. `https://www.domo.com/blog/2014/04/data-never-sleeps-2-0/`.

[5] Alex Wright. Big data meets big science. *Communications of the ACM*, 57(7):13–15, 2014.

[6] Daniel A Reed and Jack Dongarra. Exascale computing and big data. *Communications of the ACM*, 58(7):56–68, 2015.

[7] Sidharth N Kashyap, Ade J Fewings, Jay Davies, Ian Morris, Andrew Thomas Thomas Green, and Martyn F Guest. Big data at hpc wales. *arXiv preprint arXiv:1506.08907*, 2015.

[8] I. Tagawa and M. Williams. High density data-storage using shingled-write. *Proceedings of the IEEE International Magnetics Conference (INTERMAG)*, 2009.

[9] Philip Carns, Robert Latham, Robert Ross, Kamil Iskra, Samuel Lang, and Katherine Riley. 24/7 characterization of petascale i/o workloads. In *Cluster Computing*

and Workshops, 2009. CLUSTER'09. IEEE International Conference on, pages 1–10. IEEE, 2009.

[10] Mark H Kryder, Edward C Gage, Terry W McDaniel, William A Challener, Robert E Rottmayer, Ganping Ju, Yiao-Tee Hsia, and M Fatih Erden. Heat assisted magnetic recording. *Proceedings of the IEEE*, 96(11):1810–1835, 2008.

[11] Elizabeth A Dobisz, Zvonomir Z Bandic, Tsai-Wei Wu, and Thomas Albrecht. Patterned media: nanofabrication challenges of future disk drives. *Proceedings of the IEEE*, 96(11):1836–1846, 2008.

[12] Akira Kikitsu, Yoshiyuki Kamata, Masatoshi Sakurai, and Katsuyuki Naito. Recent progress of patterned media. *Magnetics, IEEE Transactions on*, 43(9):3685–3688, 2007.

[13] Prakash Kasiraj, Richard MH New, Jorge Campello De Souza, and Mason Lamar Williams. System and method for writing data to dedicated bands of a hard disk drive, February 10 2009. US Patent 7,490,212.

[14] Garth Gibson and Milo Polte. Directions for shingled-write and twodimensional magnetic recording system architectures: Synergies with solid-state disks. *Parallel Data Lab, Carnegie Mellon Univ., Pittsburgh, PA, Tech. Rep. CMU-PDL-09-014*, 2009.

[15] T10 Standard. `http://www.t10.org`.

[16] Seagate Archive HDD. `http://www.seagate.com/products/enterprise-servers-storage/nearline-storage/archive-hdd/`.

[17] WD UltraStar Ha10. `http://www.hgst.com/products/hard-drives/ultrastar-archive-ha10`.

[18] IOR2. `http://sourceforge.net/projects/ior-sio/`.

[19] NPB. `http://www.nas.nasa.gov/publications/npb.html`.

[20] FLASH-IO. `http://www.mcs.anl.gov/research/projects/pio-benchmark`.

[21] Dan Feng, Qiang Zou, Hong Jiang, and Yifeng Zhu. A novel model for synthesizing parallel i/o workloads in scientific applications. In *Cluster Computing, 2008 IEEE International Conference on*, pages 252–261. IEEE, 2008.

[22] Qiang Zou, Yifeng Zhu, and Dan Feng. A study of self-similarity in parallel i/o workloads. In *Mass Storage Systems and Technologies (MSST), 2010 IEEE 26th Symposium on*, pages 1–6. IEEE, 2010.

[23] LANL-Trace Framework. `http://institute.lanl.gov/data/software/#lanl-trace`.

[24] MPI-IO Test. `http://institutes.lanl.gov/data/software/index.php#mpi-io`.

[25] Feng Wang, Qin Xin, Bo Hong, Scott A Brandt, Ethan L Miller, Darrell DE Long, and Tyce T McLarty. File system workload analysis for large scale scientific computing applications. In *Proceedings of the 21st IEEE/12th NASA Goddard Conference on Mass Storage Systems and Technologies*, pages 139–152, 2004.

[26] Philip Carns, Kevin Harms, William Allcock, Charles Bacon, Samuel Lang, Robert Latham, and Robert Ross. Understanding and improving computational science storage access through continuous characterization. *ACM Transactions on Storage (TOS)*, 7(3):8, 2011.

[27] Christopher Muelder, Carmen Sigovan, Kwan-Liu Ma, Jason Cope, Sam Lang, Kamil Iskra, Pete Beckman, and Robert Ross. Visual analysis of i/o system behavior for high-end computing. In *Proceedings of the third international workshop on Large-scale system and application performance*, pages 19–26. ACM, 2011.

[28] James Oly and Daniel A Reed. Markov model prediction of i/o requests for scientific applications. In *Proceedings of the 16th international conference on Supercomputing*, pages 147–155. ACM, 2002.

[29] Mengzhi Wang, Anastassia Ailamaki, and Christos Faloutsos. Capturing the spatio-temporal behavior of real traffic data. *Performance Evaluation*, 49(1):147–163, 2002.

[30] Christina Delimitrou, Sriram Sankar, Kushagra Vaid, and Christos Kozyrakis. Accurate modeling and generation of storage i/o for datacenter workloads. In *Proceedings of the 2nd Workshop on Exascale Evaluation and Research Techniques, EXERT, Newport Beach, CA (March 2011)*, 2011.

[31] G Horn, Amund Kvalbein, J Blomskøld, and E Nilsen. An empirical comparison of generators for self similar simulated traffic. *Performance Evaluation*, 64(2):162–190, 2007.

[32] Will E Leland, Murad S Taqqu, Walter Willinger, and Daniel V Wilson. On the self-similar nature of ethernet traffic (extended version). *Networking, IEEE/ACM Transactions on*, 2(1):1–15, 1994.

[33] Mark E Crovella and Azer Bestavros. Self-similarity in world wide web traffic: evidence and possible causes. *Networking, IEEE/ACM Transactions on*, 5(6):835–846, 1997.

[34] Priscilla ASM Barreto, Paulo HP de Carvalho, Jose AM Soares, and H Abdalla Jr. A traffic characterization procedure for multimedia applications in converged networks. In *Modeling, Analysis, and Simulation of Computer and Telecommunication Systems, 2005. 13th IEEE International Symposium on*, pages 153–160. IEEE, 2005.

[35] Christos Stathis and Basil Maglaris. Modelling the self-similar behaviour of network traffic. *Computer Networks*, 34(1):37–47, 2000.

[36] Maria E Gomez and Vicente Santonja. Analysis of self-similarity in i/o workload using structural modeling. In *Modeling, Analysis and Simulation of Computer and Telecommunication Systems, 1999. Proceedings. 7th International Symposium on*, pages 234–242. IEEE, 1999.

[37] Vern Paxson and Sally Floyd. Wide area traffic: the failure of poisson modeling. *IEEE/ACM Transactions on Networking (ToN)*, 3(3):226–244, 1995.

[38] Sriram Sankar and Kushagra Vaid. Storage characterization for unstructured data in online services applications. In *Workload Characterization, 2009. IISWC 2009. IEEE International Symposium on*, pages 148–157. IEEE, 2009.

[39] Minnesota Supercomputing Institute. `https://www.msi.umn.edu/`.

[40] Rajeev Thakur, William Gropp, and Ewing Lusk. Data sieving and collective i/o in romio. In *Frontiers of Massively Parallel Computation, 1999. Frontiers' 99. The Seventh Symposium on the*, pages 182–189. IEEE, 1999.

[41] Andy Konwinski, John Bent, James Nunez, and Meghan Quist. Towards an i/o tracing framework taxonomy. In *Proceedings of the 2nd international workshop on Petascale data storage: held in conjunction with Supercomputing'07*, pages 56–62. ACM, 2007.

[42] Swapnil V Patil, Garth A Gibson, Sam Lang, and Milo Polte. Giga+: scalable directories for shared file systems. In *Proceedings of the 2nd international workshop on Petascale data storage: held in conjunction with Supercomputing'07*, pages 26–29. ACM, 2007.

[43] iPic3D. `https://github.com/CmPA/iPic3D`.

[44] Hongzhang Shan and John Shalf. Using ior to analyze the i/o performance for hpc platforms. 2007.

[45] Itasca. `https://www.msi.umn.edu/hpc/itasca`.

[46] Gregory W Corder and Dale I Foreman. *Nonparametric statistics: A step-by-step approach*. John Wiley & Sons, 2014.

[47] S Donovan, G Huizenga, AJ Hutton, CC Ross, MK Petersen, and P Schwan. Lustre: Building a file system for 1000-node clusters. In *Proceedings of the Linux Symposium*, 2003.

[48] Robert B Ross, Rajeev Thakur, et al. Pvfs: A parallel file system for linux clusters. In *Proceedings of the 4th annual Linux showcase and conference*, pages 391–430, 2000.

[49] Frank B Schmuck and Roger L Haskin. Gpfs: A shared-disk file system for large computing clusters. In *FAST*, volume 2, page 19, 2002.

[50] HDF5 homepage. `http://www.hdfgroup.org/HDF5/`.

[51] Jianwei Li, Wei-keng Liao, Alok Choudhary, Robert Ross, Rajeev Thakur, William Gropp, Rob Latham, Andrew Siegel, Brad Gallagher, and Michael Zingale. Parallel netcdf: A high-performance scientific i/o interface. In *Supercomputing, 2003 ACM/IEEE Conference*, pages 39–39. IEEE, 2003.

[52] Weikuan Yu and Jeffrey Vetter. Parcoll: Partitioned collective i/o on the cray xt. In *Parallel Processing, 2008. ICPP'08. 37th International Conference on*, pages 562–569. IEEE, 2008.

[53] Yong Chen, Xian-He Sun, Rajeev Thakur, Philip C Roth, and William D Gropp. Lacio: A new collective i/o strategy for parallel i/o systems. In *Parallel & Distributed Processing Symposium (IPDPS), 2011 IEEE International*, pages 794–804. IEEE, 2011.

[54] JG Bias, Florin Isailă, David E Singh, and Jesús Carretero. View-based collective i/o for mpi-io. In *Cluster Computing and the Grid, 2008. CCGRID'08. 8th IEEE International Symposium on*, pages 409–416. IEEE, 2008.

[55] Wei-keng Liao and Alok Choudhary. Dynamically adapting file domain partitioning methods for collective i/o based on underlying parallel file system locking protocols. In *High Performance Computing, Networking, Storage and Analysis, 2008. SC 2008. International Conference for*, pages 1–12. IEEE, 2008.

[56] Robert McLay, Doug James, Si Liu, John Cazes, and William Barth. A user-friendly approach for tuning parallel file operations. In *High Performance Computing, Networking, Storage and Analysis, SC14: International Conference for*, pages 229–236. IEEE, 2014.

[57] Mohamad Chaarawi and Edgar Gabriel. Automatically selecting the number of aggregators for collective i/o operations. In *Cluster Computing (CLUSTER), 2011 IEEE International Conference on*, pages 428–437. IEEE, 2011.

[58] Weifeng Liu, Isaias A Urena, Michael Gerndt, and Bin Gong. Automatic mpi-io tuning with the periscope tuning framework. In *Parallel & Distributed Processing Symposium Workshops (IPDPSW), 2014 IEEE International*, pages 352–360. IEEE, 2014.

[59] Joachim Worringen. Self-adaptive hints for collective i/o. In *Recent Advances in Parallel Virtual Machine and Message Passing Interface*, pages 202–211. Springer, 2006.

[60] Rajeev Thakur, William Gropp, and Ewing Lusk. On implementing mpi-io portably and with high performance. In *Proceedings of the sixth workshop on I/O in parallel and distributed systems*, pages 23–32. ACM, 1999.

[61] mpich2 versions. `http://www.mpich.org/`.

[62] Open-MPI. `http://www.open-mpi.org/`.

[63] Rajeev Thakur, Ewing Lusk, and William Gropp. Users guide for romio: A high-performance, portable mpi-io implementation. Technical report, Technical Report ANL/MCS-TM-234, Mathematics and Computer Science Division, Argonne National Laboratory, 1997.

[64] Kent E Seamons, Ying Chen, P Jones, J Jozwiak, and Marianne Winslett. Server-directed collective i/o in panda. In *Supercomputing, 1995. Proceedings of the IEEE/ACM SC95 Conference*, pages 57–57. IEEE, 1995.

[65] Babak Behzad, Huong Vu Thanh Luu, Joseph Huchette, Surendra Byna, Ruth Aydt, Quincey Koziol, Marc Snir, et al. Taming parallel i/o complexity with auto-tuning. In *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*, page 68. ACM, 2013.

[66] Babak Behzad, Surendra Byna, Stefan M Wild, Mr Prabhat, and Marc Snir. Improving parallel i/o autotuning with performance modeling. In *Proceedings of the 23rd international symposium on High-performance parallel and distributed computing*, pages 253–256. ACM, 2014.

[67] Rajeev Thakur, William Gropp, and Ewing Lusk. An abstract-device interface for implementing portable parallel-i/o interfaces. In *Frontiers of Massively Parallel Computing, 1996. Proceedings Frontiers' 96., Sixth Symposium on the*, pages 180–187. IEEE, 1996.

[68] mpi-tile-io. `http://www.mcs.anl.gov/~thakur/pio-benchmarks.html`.

[69] Feiyi Wang, Sarp Oral, Galen Shipman, Oleg Drokin, Tom Wang, and Isaac Huang. Understanding lustre filesystem internals. *Oak Ridge National Laboratory, National Center for Computational Sciences, Tech. Rep*, 2009.

[70] Anthony Chan, William Gropp, and Ewing Lusk. User's guide for mpe extensions for mpi programs. Technical report, Technical Report ANL-98/xx, Argonne National Laboratory, 1998. The updated version is at ftp://ftp. mcs. anl. gov/pub/mpi/mpeman. ps, 1998.

[71] HDD Doinate Data Storage. `http://seekingalpha.com/article/1481641-how-long-will-the-hdd-dominate-data-storage-more-than-long-enough`.

[72] Ahmed Amer, JoAnne Holliday, Darrell DE Long, Ethan L Miller, J Paris, and Thomas Schwarz. Data management and layout for shingled magnetic recording. *Magnetics, IEEE Transactions on*, 47(10):3691–3697, 2011.

[73] WA Challener, C Peng, AV Itagi, D Karns, Y Peng, X Yang, X Zhu, NJ Gokemeijer, Y-T Hsia, G Ju, et al. The road to hamr. In *Magnetic Recording Conference, 2009. APMRC'09. Asia-Pacific*, pages 1–2. IEEE, 2009.

[74] Robert E Rottmayer, Sharat Batra, Dorothea Buechel, William A Challener, Julius Hohlfeld, Yukiko Kubota, Lei Li, Bin Lu, Christophe Mihalcea, Keith Mountfield, et al. Heat-assisted magnetic recording. *Magnetics, IEEE Transactions on*, 42(10):2417–2421, 2006.

[75] Roger Wood, Mason Williams, Aleksandar Kavcic, and Jim Miles. The feasibility of magnetic recording at 10 terabits per square inch on conventional media. *Magnetics, IEEE Transactions on*, 45(2):917–923, 2009.

[76] Weiping He and David HC Du. Novel address mappings for shingled write disks. In *6th USENIX Workshop on Hot Topics in Storage and File Systems (HotStorage 14)*. USENIX Association.

[77] Damien Le Moal, Zvonimir Bandic, and Cyril Guyot. Shingled file system host-side management of shingled magnetic recording disks. In *Consumer Electronics (ICCE), 2012 IEEE International Conference on*, pages 425–426. IEEE, 2012.

[78] Ahmed Amer, Darrell DE Long, Ethan L Miller, J-F Paris, and SJT Schwarz. Design issues for a shingled write disk system. In *Mass Storage Systems and Technologies (MSST), 2010 IEEE 26th Symposium on*, pages 1–12. IEEE, 2010.

[79] Chung-I Lin, Dongchul Park, Weiping He, and David HC Du. H-swd: Incorporating hot data identification into shingled write disks. In *Modeling, Analysis & Simulation of Computer and Telecommunication Systems (MASCOTS), 2012 IEEE 20th International Symposium on*, pages 321–330. IEEE, 2012.

[80] Yuval Cassuto, Marco AA Sanvido, Cyril Guyot, David R Hall, and Zvonimir Z Bandic. Indirection systems for shingled-recording disk drives. In *Mass Storage Systems and Technologies (MSST), 2010 IEEE 26th Symposium on*, pages 1–14. IEEE, 2010.

[81] David Hall, John H Marcos, and Jonathan D Coker. Data handling algorithms for autonomous shingled magnetic recording hdds. *Magnetics, IEEE Transactions on*, 48(5):1777–1781, 2012.

[82] Chao Jin, Wei-Ya Xi, Zhi-Yong Ching, Feng Huo, and Chun-Teck Lim. Hismrfs: A high performance file system for shingled storage array. In *Mass Storage Systems and Technologies (MSST), 2014 30th Symposium on*, pages 1–6. IEEE, 2014.

[83] Abutalib Aghayev and Peter Desnoyers. SkylightâĂŤa window on shingled disk operation. In *Proceedings of the 13th USENIX Conference on File and Storage Technologies, FAST 2015, Santa Clara, CA, USA, February 16-19*, pages 135–149, 2015.

[84] Jiguang Wan, Nannan Zhao, Yifeng Zhu, Jibin Wang, Yu Mao, Peng Chen, and Changsheng Xie. High performance and high capacity hybrid shingled-recording disk system. In *Cluster Computing (CLUSTER), 2012 IEEE International Conference on*, pages 173–181. IEEE, 2012.

[85] D. Luo, J. Wan, Y. Zhu, N. Zhao, F. Li, and C. Xie. Design and implementation of a hybrid shingled write disk system. *Parallel and Distributed Systems, IEEE Transactions on*, PP(99):1–1, 2015.

[86] Garth Gibson and Greg Ganger. Principles of operation for shingled disk devices. Technical report, Tech. Rep. CMU-PDL-11-107, Carnegie Mellon University, 2011.

[87] MSR Cambridge Block I /O Traces. http://iotta.snia.org/traces/list/BlockIO.

[88] DiskSim. http://www.pdl.cmu.edu/DiskSim/.

[89] University of Massachusetts Amhesrst Storage Traces. http://traces.cs.umass.edu/index.php/Storage/Storage.

[90] Youyou Lu, Jiwu Shu, Weimin Zheng, et al. Extending the lifetime of flash-based storage through reducing write amplification from file systems. In *FAST*, pages 257–270, 2013.

[91] Seagate Cheetah 15K.5 FC product manual. http://www.seagate.com/staticfiles/support/disc/manuals/enterprise/cheetah/15K.5/FC/100384772f.pdf.