UNIVERSITY OF OSLO
Department of Informatics

Performance Evaluation Of FileSystems Compression Features

Master Thesis In the field of

Network and System Administration

Solomon Legesse

Oslo and Akerhus University College

(hioa) In collaboration with

University of Oslo (UiO)

May 20, 2014

Performance Evaluation Of FileSystems Compression Features

Master Thesis In the field of

Network and System Administration

Solomon Legesse

Oslo and Akerhus University College (hioa) In collaboration with University of Oslo (UiO)

May 20, 2014

Abstract

The Linux operating system already provide a vast number of filesystems to the user community. In general, having a filesystem that can provide scalability, excellent performance and reliability is a requirement, especially in the lights of the very large data size being utilized by most IT data centers. Recently modern file systems has begun to include transparent compression as main features in their design strategy. Transparent compression is the method of compressing and decompressing data so that it takes relatively less space. Transparent compression can also improve IO performance by reducing IO traffic and seek distance and has a negative impact on performance only when single-thread I/O latency is critical. Two of the newer filesystem technologies that aim at addressing todays IO challenges are ZFS and Btrfs. Using high speed transparent compression algorithms like LZ4 and LZO with Btrfs and Zfs can greatly help to improve IO performance. The goal of this paper is threefold. 1st, to evaluate the impact of transparent compression on performance for Btrfs and ZFS, respectively. 2nd, to compare the two file system compression feature on performance. Thirdly studying the performance behavior of a more traditional filesystem implementation, ext4 as an additional reference point. The experiment is carried out by conducting an actual empirical analysis, comparing the performance behavior of Btrfs and ZFS under varying workload conditions, utilizing an identical hardware setup for all the benchmarks. The results obtained from the Iozone benchmarking tool show a large difference between Zfs and Btrfs compression feature performance. LZ4 compression option shows very high performance improvement on Zfs for single disk set up for all record and file sizes while LZO compression option shows no significant performance improvement on Btrfs for single disk set up. The compression feature of both file system has shown better performance improvement for for raid disk set up.

Acknowledgements

All things came into being through Him, and apart from Him nothing came into being that has come into being. John 1:3

My first and deepest gratitude goes to my beloved wife, Chachi Desalegn, to my beloved son Samuel Solomon and my beloved daughter Jael Solomon. You are my strength, my courage, and the love of my life. You gave me your precious time so that I can follow my childhood dream, the dream of learning. Chachi I would like to use this opportunity to express my sincere appreciation to your selfless personality.

Im heartily thankful to my supervisor, Ismail Hassan, for his great support, motivating discussions and encouragement. I also would like to extend my deepest gratitude to my other instructors professor Kyrre Begnum and professor Hårek Haugerud. Dear Kyrre, even though you didnt supervise me directly in this project, your influence has been so enormous in shaping the theme of this thesis and I also acquire the very basic and key principles and advanced way of doing research and how to write a thesis from your wonderful and high standard teachings.

A number of friends have contributed to the success of my academic and nonacademic life in Norway. Iam always blessed in having wonderful friends whose blessed heart imagine the success of others. My special thanks go to my best friends Frezer Teklu, Neftalem Woldemariam, Addisu Tesfaye, Eskedar Kefialew, Solomon Habtu, and my class mates. Thank you for being there for me.

Last, but not least; it is an honor for me to express my deepest gratitude to my parents for their special love and scarifies to their children. Dad and Mom you are so much loving parents, Im very proud to have been raised in that lovely family and always wonder how you able to create such strong bond between us.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

1.1 Motivation

A filesystem is the method and data structure that an operating system uses to keep track of files on a disk or partition [1]. The desire to develop a better performing filesystem is an issue that has been significant for decades.

Currently, the increase of data size in todays data centers makes it an even more crucial topic that needs due consideration. In general, having a filesystem that can provide scalability, excellent performance and reliability is a requirement for modern computer systems.

Breaking IO performance bottleneck is one of the focus in the design of the next generation file systems. According to moore's law [2] The computing power of cpu and memory size of computers are better solved than the still lagging disc IO throughput performance improvement. So the focus of todays file system design is evolving around this crucial topic.

Recently modern file systems has began to include transparent compression as main features in their design strategy. Transparent compression is the method of compressing and decompressing data so that it takes relatively less space. So it increasing space utilization on hard discs. Transparent compression can also improve IO performance by reducing IO traffic and seek distance and has a negative impact on performance only when single-thread I/O latency is critical.

Using high speed transparent compression algorithms like LZ4 and LZO with advanced next generation file systems like Btrfs and Zfs can greatly help to improve IO performance and contribute for fast data consumption and retrieval in todays data oriented society who is fuled by internet, mobile computing and social media applications.

Transparent compression can be done in fraction of seconds because of the gigantic power of cpu now a days and it is applicable to include it as part of on the fly operation in the file system. Over the years, the Linux operating system has provided different kinds of filesystems, beginning with the well known ext2, as its default base file system. More recent ones have added a variety of features and functionality having their own strengths and shortcomings especially those affecting file system(IO) performance.

ZFS on linux is the most Recent file system ported to linux in 2013 from free BSD which has the got the nick name the last word in file system [3]. It is very highly scalable file system and almost fail free file system which makes it to be highly reliable. ZFS is more than file system , it is actually designed to be storage manager.

The Btrfs filestem was developed beginning in the year 2007. It provides additional features over those in the ext4 file system. Btrfs was is designed to deliver significant improvements in scalability, reliability, and ease of management [4]. The Btrfs filesystem has built-in support for software RAID, including balancing multiple devices and recovering from corruption. It also supports live resizing and device addition and removal [5], as well as transparent compression, creation of snapshots and support for subvolumes.

The Ext4 filesystem was developed to addressing scalability, performance, and reliability issues faced by ext3 [6]. It provides support for large size filesystems and advanced features such as implementation of extents, delayed and multi-block allocations (in order to prevent file fragmentation), and persistent preallocation.

All these evolution and new development has been mostly in search of performance among other things. The aim of this research is to undertake an I/O performance investigation and comparison between ZFS on Linux, Btrfs with respect to transparent compression. The impact of transparent compression feature of both Zfs and Btrfs will be investigated on each filesystem and further the impact will be compared to each other. It will examine their general performance for a variety of tasks against compression feature.. It will also determine if there is a performance impact associated with the added features of compression and logical volume management which are part of ZFS on Linux and Btrfs and available via separate software for ext4.

1.2 Problem Statement

The research question described in this thesis is both investigation and comparison of the ZFS, Btrfs and , focusing on the following scenarios and questions:

- Does Real Time Transparent Compression In Filesystems improve IO performance ?
- Does Compression Improve IO Performance In Zfs And Btrfs FileSystems?

Performance: For this discussion, the term performance refers primarily to I/O throughput of the filesystem.

Transparent Compression: built in feature that compresses every read/write task on the fly transparently with out loss and with out the knowledge of the user.

Impact or Effective: these are measurable values in Kb/s.

This problem statement has been chosen because of the current challenge facing I/O performance. Transparent Compression has now becoming our savior tool to achieve this goal. So the above scenarios will be dealt properly by benchmarking the mentioned file systems with and without compression.

1.3 Research Goals and Contributions

Modern filesystems has come with excellent and advanced features like transparent compression. This is highly reflected on Btrfs and Zfs linux filesystem. Transparent compression is one of the latest innovation to be included as part of next generation filesystems which need to be tested in order to know how well it affects performance. LZ4 is a very fast lossless compression algorithm, providing compression speed at 400 MB/s per core, scalable with multi-cores CPU. It also features an extremely fast decoder, with speed in multiple GB/s per core, typically reaching RAM speed limits on multi-core systems. LZO also offers pretty fast compression and extremely fast decompression. There fore lz4 and lzo are very good for real time or near-real time compression, providing significant space saving at a very high speed and possibly positive performance impact for some workloads. It is still under heavy development and benchmarking and evaluation of this IO performance improviser is that makes this project important. Finally the findings of this study will suggest which record and file workloads are favourable to exploit the technology.

1.4 Thesis Outline

This paper is organized in the following manner:-

The first chapter provides the motivation of the research paper and specify research questions that needs to be addressed in this research paper.

The second chapter provides background information about filesystems in general, detailed feature design and structure of Zfs, Btrfs and Ext4 filesystems and also related works that have been done on benchmarking Zfs, Btrfs and Ext4 filesystems.

The third chapter explains the experimental setup , hardware and software specification as well as about the selected benchmarking tools.

The fourth chapter present results obtained from different benchmaking tools used for this project.

The fifth chapter present analysis based on the result obtained form the fourth chapter.

The six chapter present discussion based on the analysis obtained form the fifth chapter.

The seventh chapter is dedicated for conclusion and suggestion for future works.

Chapter 2

Background and Related Works

For better understanding of the subject matter The first part of chapter will discuss background information about local filesystems, architecture, evolution and features in short. The next part of this sections will provide detailed discussions of the features and design of the Zfs, Btrfs and Ext4 filesystems, and the last section will describe filesystem performance benchmarking, tools and finally related works.

2.1 Local Filesystems

Filesystems determine the way that the storage of data is organized on a disk. Linux operating systems have different kinds of filesystems with features that differentiate them from one another. Each type of filesystem has its own set of rules for controlling the allocation of disk space to files and for associating related data about each file (known as metadata) with that file. Metadata includes its filename, the directory in which it is located, its permissions and its creation and modification dates[ref1].

For Linux operating system, the important aspects of a file system is how the data is organised, e.g., in linked lists, i-nodes or B-trees, how many blocks there are in a sector, caching and block size, to name a few examples. The users are concerned with what files are, how they are protected and how they can be used. For example a B-Tree is a data structure in the form of a balanced tree. Balanced means that all leafs have the same distance from the root of the tree, which makes data look ups efficien [7].

The flexibility of the Linux operating system in supporting multiple filesystems arises from its implantation of abstraction in its low-level filesystem interface. This is possible because the Virtual Filesystem Switch (VFS), a special kernel interface level, defines a common, low-level model that can be used to represent any particular filesystems features and operation [8] [13]. In addition to this abstraction of the lowest-level file operation from the underlying

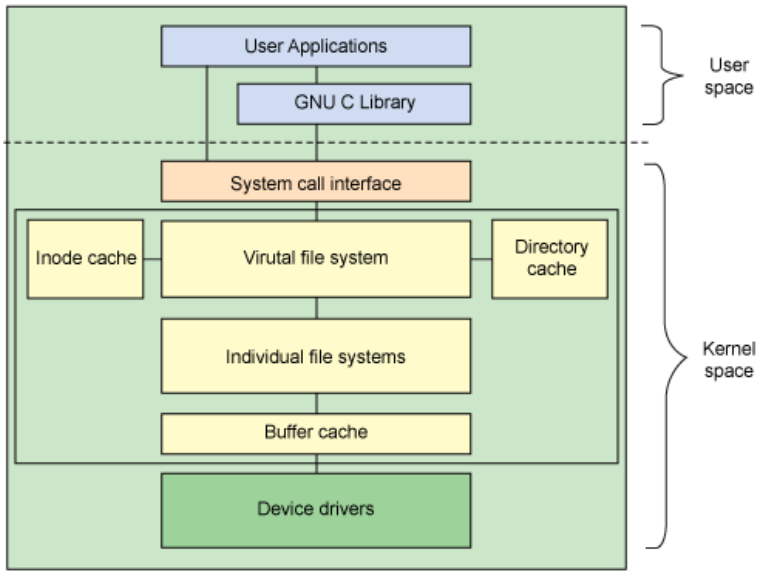


Figure 2.1: Architectural view of linux filesystem components

filesystem, the VFS also connects physical (block) devices to the actual filesystems that are in use.

2.1.1 Architecture

Linux views all filesystems as a common set of objects, which are categorized into four major parts. The first one is the superblock that describes the structure and maintains the state of filesystems. The second major object is the Inode (short for index node) which contains metadata that is used to manage objects and specify which operations are permitted on those objects. The third object type is the directory entry (dentry), which represents a directory entry as a single component of a path. The final major object is the file object, which represents an open file associated with a process [8].

Superblock

The Superblock is a structure that represents a filesystem as a whole, together with all required information that is necessary to manage the filesystem. This information includes the name, size and state of the filesystem, a reference to the underlying block device and filesystem metadata information.

Inode

An Inode is the data structure on disk that describes and stores a files attributes, including its physical location on disk. Inodes are created at the initial stage of filesystem creation. Historically, the number of Inodes equals the max-

imum number of files of all types that can exist in a filesystem[9]. Inodes hold information such as the type of file, its access permissions, its user and group owner ids, the time of the most recent modification done to the file, the size of the file and the disk address of the files data blocks. In general, Inodes store all information about the file except the name. The filename is stored in the directory where the file is located, together with the Inode number of the file.

2.1.2 Filesystem Evolution

The Berkeley Standard Distribution (BSD) fast filesystem is the traditional filesystem used all but the earliest Unix systems. It was designed to address the performance limitations of the original System V filesystem[9]. The BSD filesystem supports filesystem block sizes of up to 64KB. Even though the increased block size over System V improves performance, it will also creates internal fragmentation as a result of wasted space. In order to tackle this problem, the BSD filesystem additionally divides a single filesytem block into fragments, and each block can be broken down in to two, four or eight fragments, which can be addressed separately[9]. The BSD filesystem divides the filesystem partitions into cylinders groups, which are comprised of one or more consecutive cylinders. Each cylinder groups will have a copy of the Superblock, a fraction of the Inodes for the filesystem and data blocks, and the block map that describes available blocks in the cylinder group[9]. The Superblock is replicated in each cylinder group for the purpose of redundancy. Since each cylinder group contains a free block map, Inodes and blocks, together with the copy of Superblock, the occurrence of data loss on some part of the disk will not affect other cylinder groups that do not belong to the affected cylinder group. The BSD filesystem directory structure is a linear list which contains a length field and the file name whose length can be up to 255 bytes [10][16].

The major drawback of the BSD filesystem is its demand to perform filesystem checking at every boot, which takes a long time. This slowness is intolerable, especially with the huge storage devices of the current era.

The default Linux filesystem for many years was the Ext2 filesystem. Ext2 inherits most characteristics from BSD filesystem and makes changes to three basic features. The first change is the elimination of fragments. The increase in disk space and file size makes the demand of partitioning blocks into fragments less important[10]. As a result, the Ext2 filesystem provides a single allocation unit, the block size, for all allocations. The second change made by Ext2 is its usage of fixed size blocks instead of cylinder groups to divide the filesystem partition, since block size is more meaningful for newer hard disk types. The third and basic change made with Ext2 is utilization of buffer cache to store metadata until it is flushed to disk, in contrast to the BSD filesystem which writes out metadata immediately to disk[**?**].

The third extended file system (ext3) was a major advance in Linux file systems, even though its performance was less than some of its competitors. The ext3 file system introduced the concept of journaling to improve the reliability of the file system when the system is abruptly halted. And although competing file systems had better performance (such as Silicon Graphics' XFS and the IBM Journaled File System [JFS]), ext3 supported in-place upgrades from systems already using ext2. Ext3 was introduced in November 2001 and implemented by Stephen Tweedie.

Ext4 introduces numerous new advancements for performance, scalability, and reliability. Most notably, ext4 supports file systems of 1 exabyte in size. Ext4 was implemented by a team of developers, led by Theodore Tso (the ext3 maintainer), and was introduced in the 2.6.19 kernel. It is now stable in the 2.6.28 kernel (as of December 2008).

Ext4 borrows many useful concepts from a variety of competing file systems. For example, the extent approach to block management had been implemented in JFS. Another block management-related feature (delayed allocation) was implemented in both XFS and in Sun Microsystems' ZFS.

Today, Ext4 is the default Linux lesystem for some common Linux distributions. As it is an in-place replacement for Ext3, older lesystems can seamlessly be manages storage in extents. It uses an efcient tree-based index to represent les and directories. A write-ahead journal is used to ensure operation atomicity. Checksumming is performed on the journal, but not on user data. Snapshots are not supported internally, rather, the underlying volume manager provides that functionality.

BTRFS is a Linux filesystem that has been adopted as the default filesystem in some popular versions of Linux. It is based on copy-on-write, allowing for efficient snapshots and clones. It uses B-trees as its main on-disk data structure. The design goal is to work well for many use cases and workloads. To this end, much effort has been directed to maintaining even performance as the filesystem ages, rather than trying to support a particular narrow benchmark use-case. much effort has been directed to maintaining even performance as the filesystem ages, rather than trying to support a particular narrow benchmark use-case.

2.1.3 File System Internals evolution

Some file systems, like ZFS and Btrfs, support multiple block sizes on the same file systems. The advantage of using fix-sized blocks is that it is simple to implement, but as the block is the smallest unit, space is wasted if they are not used fully. The best block size depends on what type of data is going to be stored on the file system, called the workload. This has to be known in ad-

vanced in order to create a file system with the appropriate block size. If the files are large, large blocks yield better performance. Because the file system divide files in blocks, an important part of the file system is to keep track of which blocks are free to store new data. To keep track of free and used blocks, and which blocks belongs to which files, the file system use methods such as allocating files blocks contiguously, as linked lists or using i-nodes.

A. Allocation Methods

Filesystems use different kinds of allocation methods to allocate disk blocks for file storage. The type of allocation method selected and implemented in a filesystem is one of the determining factors for its overall performance since effective disk space utilization and quick access to a file depends on the space allocation technique used by the filesystem [11]. In general, there are three widely used allocation methods.

Contiguous Allocation

The contiguous allocation method requires a file to occupy a set of contiguous blocks on the disk[11]. The location of a file is defined by the disk address of the first block and the size of the file. Since all records are placed next to each other, sequential access of a file is fast. Moreover, random access is also fast as it only requires getting the starting block and size of a file, which is stored in the directory entry, to locate it.

The difficulty encountered with this allocation method is finding space for new file. Two common strategies, namely first fit and best fit, are used to select an unallocated segment for the requested space of the new file [11]. The former searches for a space until it finds one that is big enough to fulfil the requirement, while the latter searches for the smallest possible unallocated segment or hole that is big enough to hold the required size. Even though these strategies may help in locating the total amount of space needed for the new file, preallocation is still a major issue. Since a file can grow from time to time, the currently allocated space might end up being unable to fulfil the new size requirement, causing the file to require relocation. This is detrimental to performance and causes filesystem fragmentation.

Extent-based allocation

Extent-based allocation maintains all the advantages of contiguous allocation techniques while at the same time provides a solution to prevent this problem. Instead of allocating a single block, this technique initially allocate a contiguous chunk of space (an extent) that can be enlarged by adding another chunk of contiguous space as the demand arises. In extent based allocation, the location of a files block is recorded as a location and a block count, plus a link to the first block [11].

Linked Allocation

The linked allocation technique uses a linked list of disk blocks for each file. The directory entry for a file contains pointers to the first and last file blocks [11]. Each data block uses 4 bytes of its space for a pointer to the next block of the file. The the last block specifies an end-of-file value in this location. This scheme is effective for sequential file access, but it does not support direct access for a single block. Direct access is only possible if implemented with a table which stores all pointers to a file.

This technique also has the advantage that it eliminates external fragmentation and allows files to increase size easily. Its greatest shortcoming is reliability. Since disk blocks are linked by pointers, a problem occurring within a single pointer can make all the remaining blocks in the chain inaccessible without rebuilding the filesystem metadata.

Indexed Allocation

In this allocation method, an index block is allocated for each file that is created. The index block of a file contains pointers to all of the data blocks for that file, essentially an array of disk block addresses [11]. The directory entry for the file contains a pointer to this index block. Indexed allocation supports both sequential and direct access. It eliminates the occurrence of external fragmentation and also the problem of file growth exhibited by the contiguous block allocation technique.

However, one of the shortcomings associated with this technique is the occurrence of internal fragmentation as a result of a free space wastage on index blocks. The other issue is the the overhead associated with having an index block, which is most significant for small files.

B. Transparent compression

Transparent compression is a way of providing automatic, on-the-fly data compression for an entire filesystem without any user knowledge or intervention. The major advantage of compression is saving disk space but it also can provide reduced disk I/O operations, which in turn leads to improvement in the filesystems overall performance compared [12].

2.2 Zfs

ZFS is a relatively new general purpose file system for the Solaris and OpenSolaris operating systems, developed to reduce the complexity of storage management.

The traditional standard file system on Solaris, UFS, has a fragmented set of different tools for managing the file system, volume management, RAID and monitoring. ZFS has two programs, with many sub-commands, to manage everything. This implies that ZFS offer more than the file system layer of a storage system. ZFS is a 128-bit file system, which means that it has a maximum data capacity of 256 quadrillion ZB(ZettaByte). Directories has a maximum of 256 trillion entries, and there are no limit on the number of files a file system can store [3].

2.2.1 Storage Pool Model

ZFS does not use the concept of traditional volumes, but has its own model of storage pools. A storage pool is a collection of storage devices, whose physical blocks are distributed to file systems, on request, in the form of virtual disk blocks, analogous to the virtual memory abstraction. This means that one pool can have several file systems attached, which can grow or shrink by virtue of the virtual block concept.

2.2.2 Dynamic Block Sizes

ZFS supports dynamically changing block sizes in the range from 512 bytes to 128 KBs[13]. Analogous to stem cells, ZFS divides all storage into so-called meta-slabs. A slab consists of one or more pages of virtually contiguous memory carved up into equal-size chunks, with a reference count indicating how many of those chunks have been allocated [14], and is used to allocate memory in the kernel of Solaris. These meta-slabs are divided into different-sized blocks, and the most efficient block size for each file is calculated from its length [15].

2.2.3 Strong Data Integrity

ZFSs solution to the silent data corruption problem is a combination of end-to-end checksumming and self-healing. Every node in the internal data structure store a 256-bit checksum of its child node, i.e., the integrity of the whole path from the root of the tree to the node (each block has one) is verified when the

data is checked for errors, which is done regularly.

Writes are atomic, copy-on-write, where blocks are not overwritten, but written to a new location, followed by updating the pointer to the data only if the write was successful[15]. If an error is detected, ZFS can heal itself by replacing the bad block with a correct copy. The latter requires a setup with mirroring using RAID-Z [16]. When data is updated, the checksums are updated through the whole tree, up to the root.

2.2.4 Integrated Software RAID

ZFS has an integrated software RAID implementation called RAID-Z. RAIDZ is a type of RAID 5 which provides striping with distributed parity bits.

ZFS also implements its own flavour of RAID 6, called RAID-Z2. RAID 6 is similar to RAID 5, but has two parity schemes and is capable of losing two devices without destroying the array[**?**].

The advantage that RAID-Z has over other hardware or software implementations is that its integration with ZFS mitigates the so-called write hole problem. The write hole is the case where an interruptiony causes inconsistencies between the data in the RAID array and its parity bits, because two devices cannot be updated atomically. ZFS solves the write hole problem by using dynamic stripe widths and never overwriting live data[17]. Conventional RAID implementations use static stripe widths [18]. That the stripe width is static, means that the data is written to the medium in equally sized chunks, and the width cannot be changed in another way than recreating the array. This also has the disadvantage that the slowest device set the performance limit. Having dynamic stripe width makes ZFS able to scale the number of writes on each device, eliminating the previous problem. ZFS can write any data block anywhere, on any disk in the RAID-Z array, in dynamically sized blocks, and use this to implement dynamic striping, by letting each block be its own stripe. This makes every write to a RAID-Z a full stripe write, which in combination with transactional copy-on-write eliminates the write hole problem [15] [17]. A full stripe write is also faster than a partial stripe write, because the parity bits do not have to be read first, before the new bits can be computed [17].

2.2.5 Copy-on-write transactional model

The ZFS design represents a combination of a file system and a volume manager[3]. The file system commands require no concept of the underlying physical disks (because of the storage pool virtualization). All of the high-level interactions occur through the data management unit (DMU), a concept that is similar to

a memory management unit (MMU) for disks instead of memory. All of the transactions committed through the DMU are atomic, and therefore the data is never left in an inconsistent state.

In addition to being a transaction-based file system, ZFS also performs copy-on-write operations[19]. This implies that the blocks containing the data (that is in use) on disk are never modified. The changed information is written to alternate blocks, and the block pointer to the data in use is only moved once the write transactions are completed. This scenario holds true all the way up the file system block structure to the top block, which is labeled the uberblock[20]. In the case that the system encounters a power outage while processing a write operation, no corruption occurs as the pointer to the good data is not moved until the entire write operation completes. It has to be pointed out that the pointer to the data is the only entity that is moved. This eliminates the need for journaling or logging, as well as for an fsck or mirror resync when a machine reboots unexpectedly.

2.2.6 Compression

ZFS is built with the realization that in modern systems we typically have large amounts of memory and CPU available, and we should be provided with the means to put those resources to work[21]. Contrast this with the traditional logic that compression slows things down, because we stop and compress the data before flushing it out to disk, which takes time. Consider that in some situations, you may have significantly faster CPU and Memory than you have IO throughput, in which case it may in fact be faster to read and write compressed data because your reducing the quanity of IO through the channel.so compression isn't just about saving disk space. ZFS uses variable block sizes when compression is enabled so if a block of data is compressible, its compressed.

2.2.7 Zfs Artechiture

Zfs artechiture consists the following units:

- The Data Management Unit (DMU) provides the object based storage model. One interacts with the DMU to modify objects in a storage pool.
- The Dataset and Snapshot Layer (DSL) provides a wrapper for object sets that enables clones and snapshots.
- The Adaptable Replacement Cache (ARC) provides the primary caching layer in the ZFS stack.

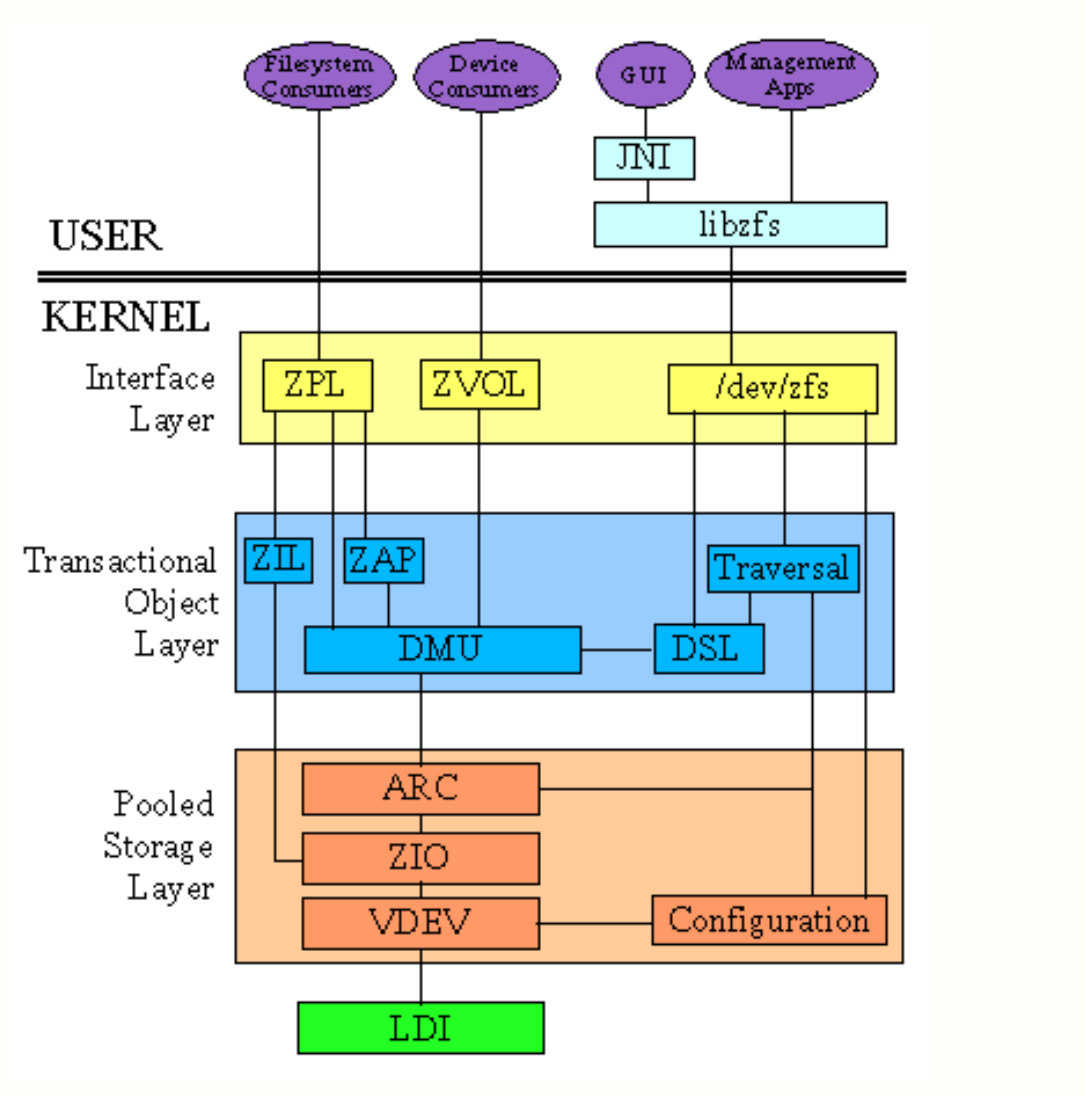


Figure 2.2: Zfs Artechitecture

.

- The ZFS Input Output framework (ZIO) provides a pipeliend I/O frame management framework for organizing the devices in a storage pool.
- The ZFS Attribute Processor (ZAP) provides a means of storing name value entries in the DMU objects.

2.2.8 ZFS Block Allocation

Block allocation is central to any filesystem. It affects not only performance, but also the administrative model (e.g. stripe configuration) and even some core capabilities like transactional semantics, compression, and block sharing between snapshots. So it's important to get it right[20]. There are three components to the block allocation policy in ZFS:

- Device selection (dynamic striping)
- Metaslab selection
- Block selection

By design, these three policies are independent and pluggable. They can be changed at will without altering the on-disk format, which gives us lots of flexibility.

The goal of device selection (dynamic striping) is to spread the load across all devices in the pool so that we get maximum bandwidth without needing any notion of stripe groups. This can be done in the fly in round-robin scheme by switching from one device to the next every 512K works well for the current generation of disk drives.

By dividing each device into a few hundred regions, called metaslabs, the one with the most free space and with the most free bandwidth the metaslab selection algorithm is fulfilled.

Having selected a metaslab, we must choose a block within that metaslab. The current allocation policy is a simple variation on first-fit. For keeping track of free space in a metaslab is a new data structure called a space map.

Every filesystem must keep track of two basic things: where your data is, and where the free space is. The most common way to represent free space is by using a bitmap but this doest scale up well for Zfs. Another common way to represent free space is with a B-tree of extents. An extent is a contiguous region of free space described by two integers: offset and length. The B-tree sorts the extents by offset so that contiguous space allocation is efficient.

Unfortunately, B-trees of extents suffer the same pathology as bitmaps when confronted with random frees. ZFS divides the space on each virtual device into a few hundred regions called metaslabs. Each metaslab has an associated

space map, which describes that metaslab's free space. The space map is simply a log of allocations and frees, in time order.

2.2.9 Arc

The ARC (Adaptive Replacement Cache) improves file system and disk performance, driving down overall system latency.

2.2.10 TXG

Transactions and Copy on Write are praised as being efficient in terms of fragmentation and data integrity. But this robustness comes with a cost in performance. Every modication of a node in the tree results in all parent nodes up to the top being modified. This means that even the slightest change in a single node scales to involve all nodes in a tree walk up to the root. When the weak performance of the disk speed is taken into account, this makes up a major problem. To solve this ZFS uses Transactions Groups. A Transaction Group is what ZFS commits to the disk. First when a Trans-action Group, TXG, is committed, the actual changes are Written to disk

2.3 The Btrfs Filesystem

Btrfs (the name stand for b-tree filesystem) is a copy-on-write (COW) Linux filesystem which is intended to address the lack of pooling, snapshots, checksums and integrated multi-device spanning in traditional Linux filesystems[5]. It has many features such as its support for snapshots of a live system, including rollback to a previous state, its capability to perform offline conversion of Ext3 and Ext4 filesystems, online block device addition and removal, and online volume growth and shrinking. Btrfs is designed to solve the problem of scalability that often occurs with large and fast storage[22]. As a 64-bit filesystem, Btrfs addresses up to 16 exabytes (16,384 petabytes), both in terms of the maximum volume size and the maximum file size[23].

2.3.1 Btrfs Design and Architecture

Btrfs uses b-trees to store generic objects of varying data types in a single, unified data structure. A b-tree is a tree data structure that allows tree nodes (also known as leaves) to have more than 2 child nodes. B-trees are designed for performance, and perform perations like searching, insertion and deletion in

logarithmic time.

Inside the b-tree, root nodes consists of two fields: the key, which holds information about the item contained in the leaves of a tree, and the block pointer, which provides information about the disk location of the next node or leaf in the b-tree[4].

Btrfs uses three types of on-disk structures, namely block headers, keys and items. The block header contains information about the block, including a checksum for the block contents, the universal unique identification (UUID) of the filesystem that owns the block, the level of the block in the tree, and the block number where this block is supposed to live.

Leaves of the tree hold the item and data fields, they grow toward one another. Items are combinations of keys and data, where the offset and size field of the item indicates the location of the item in the leaf. This way of storing the key with the data makes efficient use of space compared to the usual way of storing of only one kind of data in any given filesystem block[**?**].

Items are sorted by their 136-bit key, which groups related items together via a shared key prefix (and thus automatically optimizes the filesystem for large read and write operations). Small files can be stored directly in the tree leaves, while large files are allocated by extents. This technique both lowers the overhead and reduces fragmentation[4].

A key is divided into three chunks, which are the object id, type and offset fields. Each object in the filesystem has an object id, which is allocated dynamically on creation. The object id field allows all items for a given filesystem object to be logically grouped together in the b-tree. The offset field of the key stores the byte offset for a particular item in the object. The type field indicates the type of data stored in the item[24].

Btrfs component b-trees

A newly-created Btrfs filesystem contains five types of b-trees[23], as illustrated in Figure 2.2:

- The tree of root trees b-tree keeps track of the location of all the roots of the filesystem b-trees. It serves as a directory for all other tree roots.
- The extent tree holds information about extents allocated for the filesystem.
- The filesystem tree which contains the files and directory information.
- The chunk tree holds information about chunks of the device that are allocated and the type of data they hold.
- The checksum tree checksums of all data extents within the filesystem.

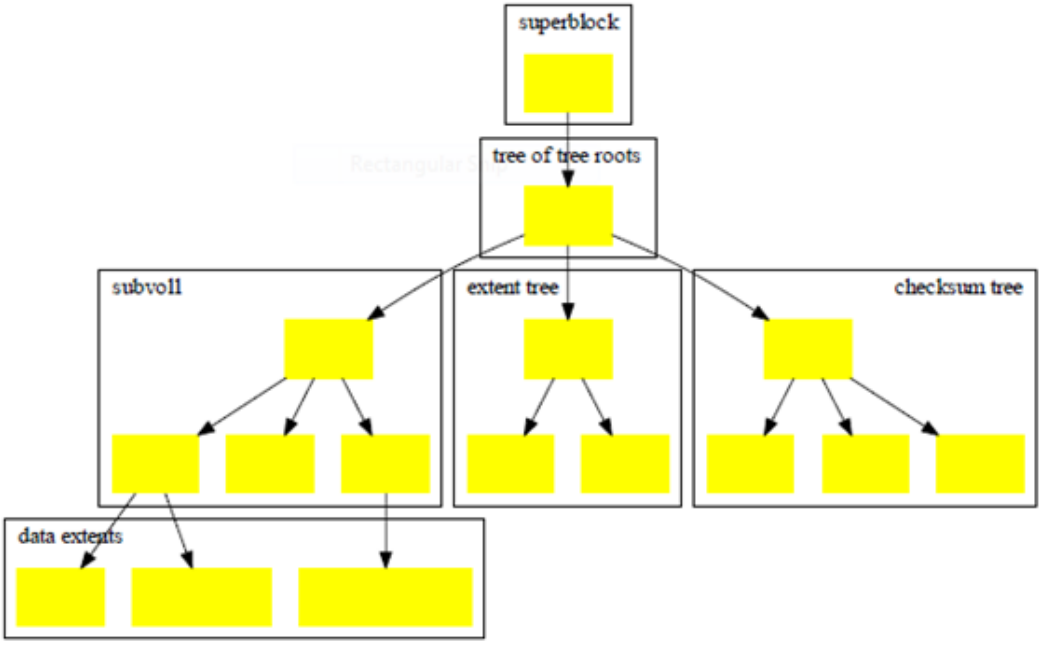


Figure 2.3: Btrfs Btree Structure

The Btrfs filesystem Superblock contains two pointers. The first pointer points to the tree of root trees, and the second pointer points to the chunk tree, which is responsible for device management[24]. Btrfs Inodes are stored in struct Btrfs Inode item. The Btrfs Inodes store the traditional Inode data for files and directories (as returned by the stat system call). The Btrfs Inode structure is relatively small, and does not contain any embedded file data or extended attribute data[4].

2.3.2 Dynamic Inode allocation

When creating the filesystem, only a few Inodes are established, rather than creating all Inodes that will ever exist at the very beginning. Based on the actual filesystem use, additional Inodes are created and allocated, which is suitable for data de-/compression in real-time. This means favoring speed over the best possible compression ratio.

2.3.3 Compression

Compression is implemented at the extent level. Btrfs implements transparent compression with two kinds of compression schemes, LZo and Zlib, with

Zlib being the default method[25]. This feature can be turned on at the mount option, and any new writes will be compressed. Moreover, Btrfs automatically identifies what should and should not be compressed to make this feature more efficient[26]. Both LZo and Zlib are of a lossless compression technique, i.e the original data can be recovered exactly from its compressed data counterpart.

- Lempel-Ziv-Oberhumer (LZO) compression is a data compression library that is suitable for data de-/compression in real time, and it which favours speed over compression ratio. It is a block compression algorithm that compresses a block of data into matches (using a sliding dictionary) and runs of non-matching literals[26]. Unlike Zlib, LZo supports a number of algorithms.
- The Zlib compression library provides in-memory compression and decompression functions, including integrity checks of the uncompressed data. It supports DEFLATE algorithm that provides good compression on a wide variety of data with minimal use of system resources[27].

2.4 IO Sub system and File System Workloads

The IO subsystem is the vital component of an operating system over which the filesystem is governing the layout of data and retrieval of the data from[28]. Performance is sometimes highly dependent of the Io subsystem on which case such IO activity is called IO bound. The Most known IO subsystem is hard disc which is managed by block level access abstraction through either SCSI or IDE controller channel. The access operations can be either read, write or seek. One of the primary goals of file system design is to hide the slow speed of the disk using techniques such as caching, prefetching, and delayed write-back.

2.4.1 IO Performnce

IO performance is important factor and several mechanisms are devised to increase the performance[29] . Some of which are the following:

- Load balancing and striping
- Buffering and Caching
- Data Compression
- Using Comcurrency using DMA
- Increase IO size and Rate

2.5 Compression

Compression is the new tool to Increase IO subsystem performance by directly affecting the size of data bound to IO channel. This will decrease the disc seek time by reducing amount of data sent to disc. Hence the write and read operations rate will increase directly.

2.5.1 What is Transparent Data compression?

Data compression, the process of encoding digital information using fewer bits [30]. In data compression, transparency is the ideal result of lossy data compression. If a lossy compressed result is perceptually indistinguishable from the uncompressed input, then the compression can be declared to be transparent. In other words, transparency is the situation where compression artifacts are nonexistent or imperceptible. It is commonly used to describe compressed data bitrates. All lossless data compression methods are transparent, by nature.

Types of Compression Techniques

The two new transparent compression techniques are LZO and LZ4

What is LZO ?

LempelZivOberhumer (LZO) is a lossless data compression algorithm that is focused on decompression speed. The LZO library implements a number of algorithms with the following characteristics [26]:

- Compression is comparable in speed to DEFLATE compression(LZ77 algorithm and Huffman coding)
- Very fast decompression Requires an additional buffer during compression (of size 8 kB or 64 kB, depending on compression level).
- Requires no additional memory for decompression other than the source and destination buffers.
- Allows the user to adjust the balance between compression ratio and compression speed, without affecting the speed of decompression.

LZO supports overlapping compression and in-place decompression. It is a block compression algorithmit compresses and decompresses a block of data. Block size must be the same for compression and decompression. LZO compresses a block of data into matches (a sliding dictionary) and runs of nonmatching literals to produce good results on highly redundant data and deals

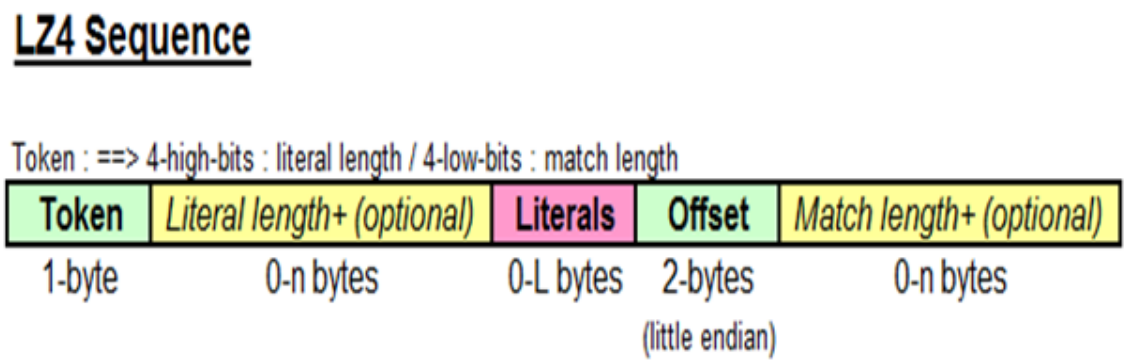


Figure 2.4: Lz4 Compressed Data Format

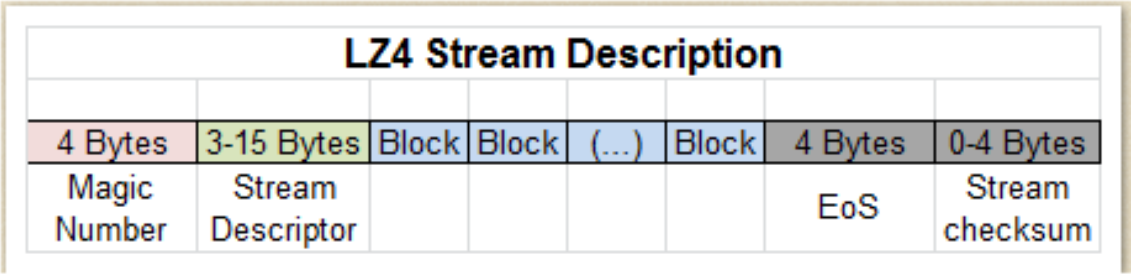


Figure 2.5: Lz4 Uncompressed Stream Data Format

acceptably with non-compressible data, only expanding incompressible data by a maximum of 1/64 of the original size when measured over a block size of at least 1 kB.

What is LZ4 ?

LZ4 is a very fast lossless compression algorithm, providing compression speed at 400 MB/s per core, scalable with multi-cores CPU [31] [32]. It also features an extremely fast decoder, with speed in multiple GB/s per core, typically reaching RAM speed limits on multi-core systems. LZ4 was also implemented natively in the Linux Kernel 3.11. The BSD implementation of the ZFS filesystem supports the LZ4 algorithm for on-the-fly compression.

Compressed Data Format
the format below describes the content of an LZ4 compressed block [33]. But a file, or a stream, of arbitrary size, may consist of several blocks. Combining several blocks together is the scope of another layer, with its own format.

2.5.2 Standard lossless data compression corpora

Silesia compression corpus

The intention of the Silesia corpus is to provide a data set of files that covers the typical data types used nowadays for researchers in the universal lossless data compression field. The sizes of the files are between 6 MB and 51 MB [34]. The chosen files are of different types and come from several sources. Nowadays the two fastest growing types of data are multimedia and databases. The former are typically compressed with lossy methods so we do not include them in the corpus. The database files, osdb, sao, nci, come from three different fields. The first one is a sample database from an open source project that is intended to be used as a standard, free database benchmark. The second one, sao, is one of the astronomical star catalogues. This is a binary database composed of records of complex structure. The last one, nci, is a part of the chemical database of structures. The sizes of computer programs are also growing rapidly. The projects are composed of hundreds or thousands files, so it is a common habit to compress it all together. We often can achieve a better compression ratio if we compress a concatenated file of similar contents than the small separate ones. This trend is reflected in including a samba file. Besides the source codes, there is also a need to store the executables. There are also types of images that cannot be compressed loosely,the medical images. The standard corpora contain text files.

2.6 Bench Marking Tools: Iozone

I. The choice of Iozone:

In general, IOzone has the following especial features and advantageous compare to the other available benchmarking tools. And, that is why it is chosen to benchmark file system. It works for all types of file systems (local, network, and distributed file systems)[35].

- It is easy to use and it works under many platforms (or operating systems)
- which includes Linux and Windows.
- It assumes its execution is bottlenecked by storage devices to avoid the
- significant effect of CPU speed and RAM size specifications.
- It is Compatible for very large file sizes.
- It is Compatible for multi-process measurement.

- It is Compatible for both single and multiple stream measurement.
- It is Compatible for POSIX Asynchronous I/O
- It is Compatible for POSIX Threads, or Pthreads.
- Its I/O Latency plots feature.
- Its processor cache size configurable feature.
- Excel importable output for graph generation feature.
- Compared to bonnie++, IOzone has more features and generates more detailed outputs than the common read and write speeds. It measures many file systems operations (files I/O performance), like: read, write, re-read, re-write, read backwards, read strided, fread, fwrite, random read/write,

II. Installing Iozone:

It is also possible to install IOzone on the Linux command line by typing:

```
$ apt-get install iozone3
```

Since the file system benchmarking result is highly influenced by the size of the systems buffer cache, before running IOzone one need to know the following requirments [35]:

I. For accuracy the max size of the file going to be tested should be bigger than buffer cache. If the buffer cache is dynamic or confusing to know its size, make the max file size bigger than the total physical memory which is in the platform [35]

II. Unless the max file size is set very smaller than the buffer cache, you must see at least the following three plateaus:

- File size fits in processor cache.
- File size fits in buffer cache.
- File size is bigger than buffer cache.

III. Use -g option to set the maximum file size value. Refer manual page of IOzone command (man iozone) for more information.

IOzone Command Line Options:
For simple start use the automatic mode:

```
$ apt-get install $ iozone -a
-a    Run in automatic mode; it generates output that
covers all  tested file operations for
record sizes of 4k to 16M for file sizes of 64KB to 512MB.
```

```
-b filename
Iozone will create a binary file format file in Excel compatible
 output of results.
-e
Include flush in the timing calculations.
-f filename
Used to specify the filename for the temporary file under test.
-g #
Set maximum file size (in Kbytes) for auto mode.
-i #
Used to specify which tests to run. (0=write/rewrite,
1=read/reread,  2=random-read/write
3=Read-backwards, 4=Re-write-record, 5=stride-read,
6=fwrite/re-fwrite,  7=fread/Re-fread,
8=random mix, 9=pwrite/Re-pwrite, 10=pread/Re-pread,
11=pwritev/Re-pwritev, 12=preadv/Re-preadv).
-s Sets file size in KB for the test. It also accepts MB and GB
which needs to be explicitly specified
-R Generate Excel report.
```

2.7 Related Works

Jan Kara and co-workers [36] undertook a comparative study of the Btrfs, Ext4,XFS and Ext3 filesystems. The experiment was performed on a two-core CPU in a single SATA drive running the 2.6.29 Kernel and with a RAID system. They made the performance comparison without including any of the features that makes Btrfs unique except that of the copy-on-write feature. One of the results of the test performed on a single SATA drive shows that Btrfs takes 10 percent less time than Ext4 to perform the task of creating 30 kernel trees. Another test on similar setup, reading 6400 files within a directory, Btrfs shows better results than that of Ext4, although it was not as good as XFS. The third test done on the single disk setup shows that Btrfs outperforms Ext4 in a 100 thread synchronous writing of 100 files. They also reported that, in the RAID setup experiment, turning on the copy-on-write feature of Btrfs causes the performance to degrade; with a test of random writes using 16 threads using the default copy-on-write feature of Btrfs, Ext4 outperformed Btrfs .

Dominique A. Heger [37] made a performance comparison among the Btrfs, ZFS and Ext4 filesystems by using the Flexible FileSystem Benchmark(FFSB) IO benchmarking set. The experiment was done on both a single disk and a RAID setup consisting of 8 Seagate driver with (Linux kernel 2.6.30). One of the major findings was that the Ext4 filesystem outperforms the others on the sequential read and mixed workloads for the single disk. Ext4 showed similar performance results with that of Btrfs for the sequential read, sequential write, random read/write and mixed tests conducted. The paper also stated that conducting the test with the nodatacow and nodatasum features of Btrfs,

which turn off COW and data checksums, gained only a small improvement on the achieved throughput.

Meaza Taye [38] has also under took a general comparative study of Btrfs and ext4 filesystem in which Compression feature on Btrfs was taken into account to see it performance impact on Btrfs eventhough ext4 has no compression feature yet enabled. In her study she both used synthetic and real world application benchmarking tools. In addition the logical volume management features of btrfs with Ext4 in combination with Linux LVM were compared. The result of the study shows that a large difference between Btrfs and Ext4 for synthetic tests performed where as the real application tests shows more or less the same result. The other interesting result from her test is the impact of compression feature on btrfs filesystem where compression has significant performance impact on btrfs operation.

Sakis Kasampalis [39] in his study of copy On Write Based File Systems Performance Analysis And Implementation specifically on the two most known file systems Btrfs and zfs where he used one application emulator for emulating macro-benchmarks, Filebench, one trace/workload generator, IOzone, and two micro-benchmarks, bonnie-64 and Bonnie++ on both single disk and mirrored raid by using the default mount option this means that in ZFS metadata are compressed and data are uncompressed, In Btrfs, both data and metadata are uncompressed has tried to get some analysis. The performance analysis between ZFS and Btrfs has shown the strengths and weaknesses of each file system. Because of its simplicity and tight integration with GNU/Linux, Btrfs performs better than ZFS on single disk systems when the caches are bypassed, and as soon as it becomes stable, it seems to be the right choice for all but metadata creation intensive applications on single disks. Btrfs should also be the preferred choice for NFS file servers and applications which rely on creating, reading, and randomly appending files. Note that all these apply only when the caches are bypassed. When the caches are utilised, ZFS seems to outperforms Btrfs in most cases, but he cannot make any conclusions since in his micro-benchmark tests he bypassed the caches. ZFS is the right choice for both single disks and mirroring for applications which use exclusively 64 KB buffering. he believes that when the caches are utilised, ZFS can outperform Btrfs no matter the size of the buffers. ZFS is also more scalable than Btrfs, which means that it is more appropriate to use it on mail servers, database servers, and applications that require synchronous semantics (fsync).

Chapter 3

Approach and Methodology

The operationalization of the problem statement stated above needs to investigated the impact of transparent compression on IO performance of Zfs and Btrfs file systems under a variety of tasks or different loads and further compare the impact between two file systems namely Zfs and Btrfs. The impact of compression will be more investigated if the set up could include both single disk and raid.
In order to attain the best possible result out of this approach the following approaches are followed

- Single disk setup (for both default and compression mount options).
- Software Raid5 setup (for both default and compression mount options).
- Using Ext4 filsystem for single and raid5 disk with out compression as baseline reference to measure how much improvement the modern filesystems has shown.
- Using better benchmarking tool (iozone and additional tools) to simulate different loads and benchmark the performance.
- Collecting data: while the benchmarking tools are running on a specific disk or raid, output data will be collected using log files and iostata command and other scripts. To increase the predictability of the experiment each experiment will be repeated a number of times as necessary.
- Analysing: Finally r studio and Microsoft Excel will be used to analyse the data mainly the Average out put and its distribution in the form of standard deviation and standard error of mean. To perform the I/O throughput performance comparison, one should use a filesystem benchmarking tool that is capable of showing how both filesystems perform under different work load. Basically there are two options:
  - real application
  - synthetic benchmarking tools

Real applications are more advantageous to use, especially if it is the type of application that is intended to be used with the filesystem since this will imitate the exact situation in the real environment. However, the problem associated with this type of benchmarking is the difficulty of finding such a representative real application [40].

The second alternative is using synthetic benchmark tools that are designed to simulate different workloads. Synthetic benchmarks are mostly flexible and have different parameters that can be adjusted for specific requirements. However, the problem with synthetic benchmarks is that they do not measure any real work. For example, the synthetic benchmark might add overhead that does not exist in a real application. On the other hand, a real application might incur overhead not modelled in the benchmark [40]. Wasim Ahmad Bhat et al. [41] specify that the ideal benchmark for file and storage systems combines the ease of use of synthetic benchmarks with the representativeness of real workloads.

Combining both Synthetic and real application benchmarking for filesystem I/O throughput measurements will produce a more representative result rather than solely depending on either of the two types of benchmarking tools. This project implements both synthetic benchmarking tool Iozone as well as real application tests by file and Directory read-/write tests .

Even though performance is a broader concept, file system performance measurement is mainly about how fast the file system is able to write and read. Accordingly and as per the description of the problem statement above, the general performance of Zfs and Btrfs will be measured against compression feature. The plan is to measure read/write speed using a benchmarking tool under varying loads (i.e. different file sizes and record lengths).

3.1 Experimental Setup

According to the plan of the approach above The experiment set is divided into the following main types.
1. first the setup is divided according to the three filesystem.
2. Then the setup has single and raid disk setup again which will give us a total of six set ups.
3. Next is test of performance with compression and and with out compression which will make the number of setup ten.

4. Then it is intended to study performance by scaling up file size and record size which will make the total number of variables to be studied

at this point to be 20 different variables.

5. Finally in total for the above ten set up 30 file sizes and 40 record sizes will be studied.
6. For these 30 files and and 40 records four write and read operations will be taken and becomes 160 record operations and 120 file operations.

7. Totally from the above 10 setup 280 write and read operations will be investigated individually.
The above seven procedures will be repeated for:

- Iozone file and record scalablity test.
- Iozone throughput mode for process scalablity
- Iostat disk and cpu monitoring test

But studying the impact of compression on record and file size is not enough. The impact of compression on throughput as the number of thread increases is necessary to investigate the relationship b/n Compression and cpu usage. So for one file and one record size when the number of thread scales from one to five will be studied. Which means ten Iozone throughput commands will be run and data will be collected.

Additional 5 dd commands and 5 linux compile commands will be run to see additional impact of compression other than iozone one file copy and compiling time as alternative approach to iozone. Depending one kind of tool to bench mark will mislead the result. So additional 20 commands will be run.

Finally a number commands and data collection scripts will be used, around 38 scripts will be used to get result for the necessary tests mentioned above.

In the plan of data collection procedure, mainly MS excel will be used to collect the sample data for the two kind of iozone commands, one for the record and file combination and the other for throughput.

Last but not least of the approach and setup chapter is how the disk and file system configuration looks likes. First 5 benchmarking disks of 73GB size are available for this experiment. The first step is to prepare the disks for single disk setup. Then all the three filesystem will be configured first with out compression and all the planed commands will be run. Next to umount the default option of the file system and mount the compression option and do all the tests again.

Then the disks has to be configured for raid set up and the above tests has to be repeated again for both compression and default mount options.

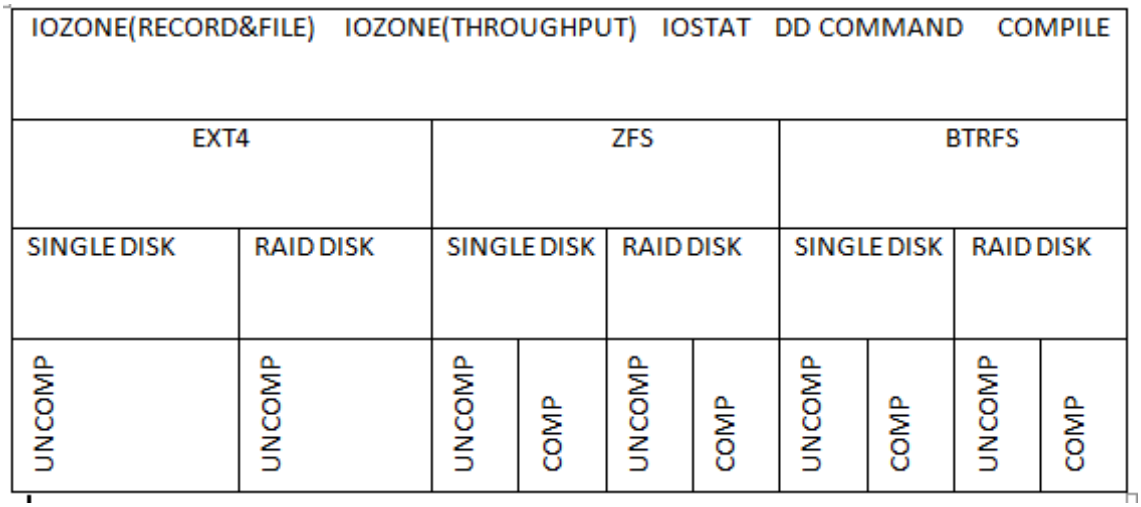


Figure 3.1: Summary of setup and Neccessary tools to be used

.

The Benchmarking is to be taken under different conditions and using number of options.
The overall hardware specifications and the topology of the file systems used in the process of the experiment are stated in this chapter.
The experiment is organized according to the type of conditions sets in the benchmarking process. The IOzone options used are discussed in this chapter. Finally, the Perl scripts used in the process of benchmarking and data collections are discussed

The experiments were conducted on a Dell PowerEdge 2850 Rack Mount Chassis Server with 64bits two dual-core Intel(R) Xeon(TM) CPU with a clock speed of 3.00GHz each, 16KB L1 Cache, 2MB L2 Cache and 3GB SDDR2 memory. The Ubuntu 13.10 Operating system was used with kernel 3.11 The system contains six hard disks one with 146GB(10k rpm) and the rest 73GB(15k rpm). The first hard disk (sdb) is used only to host the operating system while the other hard disks (sdc, sdd, sde, sdf and sdg) are used entirely for the experiment. The following table shows details about the hardware and software environments used for the experimental setup.

Device-Component	**Model**
Computer	PowerEdge 2850
Cpu	2 Intelxeon CPU 3.00GHZ
L1 and L2 cache	16KiB and 2MiB
Memory	3GiB DDR2
System HDD	146GB scsi disc 15rpm
Benchmarked HDD	Benchmarked HDD 5 73GB scsi discs 10rpm

Table 3.1: Hardware Specifications

Name	**Version**
Ubuntu 13.10	Kernel 3.11
Ext4	Rw
Btrfs	Rw, lzo, Btrfs v0.20-rc1
ZfsonLinux	Rw lz4
Iozone	v.397

Table 3.2: Software Specifications

Disc 0 146GB	/dev/sdb	Os, Ext4
Disc 1 73 GB	/dev/sdc	Zfs, Btrfs, Ext4
Disc 2 73 GB	/dev/sdd	Zfs, Btrfs, Ext4
Disc 3 73 GB	/dev/sde	Zfs, Btrfs, Ext4
Disc 4 73 GB	/dev/sdf	Zfs, Btrfs, Ext4
Disc 5 73 GB	/dev/sdg	Zfs, Btrfs, Ext4

Table 3.3: Experimental Hard Disk Partition Layout

The following subsections discuss the selected benchmarking tools and necessary commands used for this project in detail.

Iozone	IOstat	DD Command	COMPILE
Records, File size and Process Scaling	cpu and disk usage monitoring	time and throughput	compiling time

Table 3.4: Bench Marking Tools

A. Combined file sizes with record sizes:
Iozone is used to test the I/O throughput for sequential read, sequential write and re-write, random read and random write test types. All of the selected test types are executed with record sizes of 256K to 2048K and file sizes of 4GB, 8GB and 16GB. These tests are done for a single disc as well as for raid disc of both filesystems with and with out Compression mount options.

B. Throughput test using iozone t:
By increasing the number of threads from one to five and see the effect of multithreaded processes on the compression efficiency and see cpu utilization as well.

C. Iostat:
To measure IOPS, CPU and disk usage while running Iozone benchmark
D. Using DD command to simulate big file read/write tests
File copy operations are used to perform sequential read and write test by reading and writing an 6GB file

E. Compiling linux 3.14 kernel:
This will enable us to measure of the impact of compression on the big linux kernel directory reading. **F. APerl script is designed to Automate the benchmarking process.**

3.1.1 Iozone benchmarking tool and options used

According to the target of this project the suitable Iozone command options are used to get the following two benchmarking conditions

1. Running Iozone in semi automatic mode to scale up the needed individual record lengths and file sizes for the different Read/write combinations.

2. Running Iozone in throughput mode to investigate the impact of compression on throughput when the number of processes scaled through one to five threads.

IOzone is discussed in the last section,the default or automatic mode uses 13

record sizes from 4KB to 16MB (which is 4, 8, 16 . . . 8192, and 16384 in KB) for each file size test from 64KB to 512MB (which is 64, 128, 256, 512 . . . 262144, 524288 in KB). Again, it measures write, rewrite, read, reread, random read, random write, etc. . . in total 13 different measures; and so 13 outputs for each combination of record size and file size. This is one of the very interesting features of IOzone if one would like to test a file system in many aspects.

In the experiment in the fulfillment of Type 1 condition above, one standard or common IOzone command will be used in all benchmarking tests which has file sizes of 4GB, 8GB and 16Gb and record sizes of 256KB, 512KB for read/reread and write/re-write performance as it is described above. The first common IOzone command to be used is:

```
iozone -a -i 0 -i 1 -i 2 -r 256 -r 512 -r 1024 -r 2048 -s 4g -s 8g
-s 16g
the following script is used to run the command
#!/usr/bin/perl
mkfs.btrfs /dev/sdg
mkdir btrfs_single_uncomp
mount /dev/sdg  /btrfs_single_uncomp/
touch btrfs_single_uncompressed

for ( $i=0; $i <= 5; $i++) {
system(" iozone -a -i 0 -i 1 -i 2 -r 256 -r 512 -r 1024 -r 2048
-s 4g -s 8g -s 16g -f /mnt/test  >> out_put$i ");
}
iozone -i 0 -i 1 -i 2 -+u -l 1 -u 5 -r 2048 -s 8g \
-F /btrfs_single_uncomp/thrpt1
/btrfs_single_uncomp/thrpt2 /btrfs_single_uncomp/thrpt3 \
/btrfs_single_uncomp/thrpt4
/btrfs_single_uncomp/thrpt5
umount /btrfs_single_uncomp
mkdir btrfs_single_comp
mount -o compress=lzo /dev/sdg  /btrfs_single_comp/
mount-o compress=lzo /dev/sdg /btrfstestcompress
iozone -a -i 0 -i 1 -i 2 -r 256 -r 512 -r 1024 -r 2048 -s 4g -s 8g \
-s 16g -f /btrfstest/btrfs_single_compressed$i
iozone -i 0 -i 1 -i 2 -+u -l 1 -u 5 -r 2048 -s 8g \
-F /btrfs_single_comp/thrpt1
/btrfs_single_comp/thrpt2 /btrfs_single_comp/thrpt3  \
/btrfs_single_comp/thrpt4 /btrfs_single_comp/thrpt5
umount /btrfs_single_comp
mkfs.btrfs -f -m raid5 -d raid5 /dev/sdd1 /dev/sde1 /dev/sdf1
mkdir /btrfsraid5comp
mount -o compress=lzo /dev/sdd1 /btrfsraid5comp/
```

```
mkdir /btrfs_raid5_compressed
touch btrfs_raid5_compfile
./iozone_test.pl&
```

The iostat command:

```
#! /usr/bin/perl
system(" iostat -c -d -x -t -m /dev/sdg 1  >> iostat.out& ");
system(" iozone -i 0 -i 1 -i 2 -r 128k -s 6g -f /mnt/test >> \
iozone_out");
```

For experiment Type 2 another common IOzone command in throughput mode was used for a single 8GB file size by scaling thread size from 1 to 5 for record size of 2048KB . The second IOzone common command to be used in experiment type 2 is:

```
iozone  -i 0 -i 1 -i 2 -+u -l 1 -u 5 -r 2048 -s 8g  -F /mnt/test1
/mnt/test2 /mnt/test3 /mnt/test4 /mnt/test5 >>throughput_out
```

data collection script

```
grep "Avg" single_zfs_disk_ucomp_throughput >> \
single_zfs_disk_ucomp_throughputAvg
grep "CPU" single_zfs_disk_ucomp_throughput >> \
single_zfs_disk_ucomp_throughputcpu
grep "Avg" single_zfs_disk_compr_throughput >> \
 single_zfs_disk_compr_throughputAvg
1599  grep "CPU" single_zfs_disk_compr_throughput >> \
single_zfs_disk_compr_throughputcpu
grep "Avg" raidz1_zfs_ucomp_throughput >> raidz1_zfs_ucomp_throughputAvg
grep "CPU" raidz1_zfs_ucomp_throughput >> raidz1_zfs_ucomp_throughputcpu
grep "Avg" raidz1_zfs_comp_throughput >> raidz1_zfs_comp_throughputAvg
grep "CPU" raidz1_zfs_comp_throughput >> raidz1_zfs_comp_throughputcpu
less iostatdataset2all.out |grep sdc >> iostatdataset2IOPS
less iostatdataset1all.out |grep sdc >>iostatdataset1IOPS
```

Large file copy script

```
dd if=/dev/zero of=speetest2 bs=128k count=46875 conv=fdatasync
dd if=speetest2 of=/dev/null bs=128k
```

Linux kernel compile script

```
wget https://www.kernel.org/pub/linux/kernel/v3.x/linux-3.14.tar.xz
cd linux-3.11.0/
echo "Clean"; make clean >/dev/null 2>&1 ;
echo "Defconfig"; make defconfig >/dev/null 2>&1 ;
echo "Build";  time ( make  >/dev/null 2>&1 );
```

3.1.2 Modeling and the Environment

The state of the system during the benchmarking process can have a significant impact on the obtained result of the benchmark. Traeger et al[9] states that some of the major factors that can affect results are cache state, filesystem ageing and non-essential processes running during the benchmarking process.

To avoid cache impact of filestems in the Iozone benchmark test, mounting and unmounting of the tested filesystem is done for every consecutive tests. Similarly for file and directory read/write tests and also for the compression tests, a reboot is performed. Both filesystems are mounted by their respective default mount option except for compression tests, which require enabling compression feature. Moreover, all non-essential process are stopped during the test.

Repetitions for the various test were performed as follows:

The selected test types of Iozone are repeated six times on both single disk and volumes and the average is taken

3.1.3 Package Installation and Configuration

Installing the Btrfs filesystem

The latest version of the file system is used

```
apt-get install btrfs-tools
```

Creating btrfs File Systems:

```
 mkfs.btrfs  /dev/sdg
```

Btrfs file system detail:

```
btrfs filesystem show /dev/sdg
Label: none  uuid: 9438fad3-e12f-4b54-a56f-35206a6d6487
Total devices 1 FS bytes used 28.00KB
devid    1 size 68.37GB used 2.04GB path /dev/sdg
Btrfs v0.20-rc1
```

Mounting btrfs File Systems

```
mount /dev/sdg /btrfstest
```

Detail of the mount point

```
btrfs filesystem df /btrfstest
```

```
Data: total=8.00MB, used=0.00
System, DUP: total=8.00MB, used=4.00KB
System: total=4.00MB, used=0.00
Metadata, DUP: total=1.00GB, used=24.00KB
Metadata: total=8.00MB, used=0.00
```

Using Compression With btrfs

```
mount -o compress=lzo /dev/sdg /btrfstest
```

Installing the Zfs file system

Installing zfs on ubuntu 13.10

steps

1. apt-add-repository –yes ppa:zfs-native/stable

2. apt-get update

3. apt-get install ubuntu-zfs

Formating Zfs filesystem

```
Zpool create f  tank /dev/sdc
zfs create tank1/datasetuncomp
zfs create tank1/datasetcomp
zfs set compression=lz4 tank1/datasetcomp
zpool create tank1 raidz1 /dev/sdd1 /dev/sde1 /dev/sdf1
zfs create -o compression=lz4 tank1/datasetcompred
```

zpool status

```
  pool: tank
 state: ONLINE
  scan: none requested
config:
 NAME                      STATE     READ WRITE CKSUM
  tank                      ONLINE       0     0     0
   /dev/sdc               ONLINE       0     0     0
```

zfs create -o mountpoint=/tank/dataset1
zfs set compression=lz4 tank/dataset1
zfs list

```
NAME            USED  AVAIL  REFER  MOUNTPOINT
tank           11.7G  55.3G    32K  /tank
tank/dataset1  11.5G  55.3G  11.5G  /tank/dataset1
tank/dataset2   206M  55.3G   206M  /tank/dataset2
```

3.1.4 Expectation of this experiment

From this experiment result and analysis the obtained data will be expected to answer the following questions:

- How is the performance of Zfs is affected when compression is applied varying the record size and the file size for both sequential and random read/write operations.
- How is the performance of Btrfs is affected when compression is applied varying the record size and the file size for both sequential and random read/write operations.
- How is the impact of compression on Zfs and Btrfs when compared.

Chapter 4

Results

The performance impact of compression on Btrfs and Zfs has been measured and investigated according to the criterias discussed in Chapter 3. A lot of benchmarking tests under different conditions were taken and results have been collected.
Every benchmarking tests are iterated a number of times so as to evaluate and interpret the results (sample outputs) as correctly as possible using statistical calculations (like: average or mean, outliers, standard deviation, max, min, median, confidence interval, standard error of mean, etc of the raw outputs). Box plot and MS Excel are used for the statistical calculations and graphs of the sample outputs of each benchmarking test results.

4.1 Performance benchmarking test Results for Zfs Single Disk

This section presents the results obtained from sequential read/write and random read/write tests of Iozone. The results are the output of 6 runs and are plotted below.

For the results presented in this section, results are limited to four record sizes: 256KB, 512KB, 1024KB and 128KB.
The graphs below show both sequential and random operations performance of a single zfs disk for uncompressed and compressed features.
Here the Y axis represents the amount of kilobytes read or written to disk in thousands. The X axis represents the number of iterations. Excel graphs (line and bar graphs) are used to show the pattern of the raw data values.

4.1.1 Performance benchmarking test Results for Zfs Single uncompressed

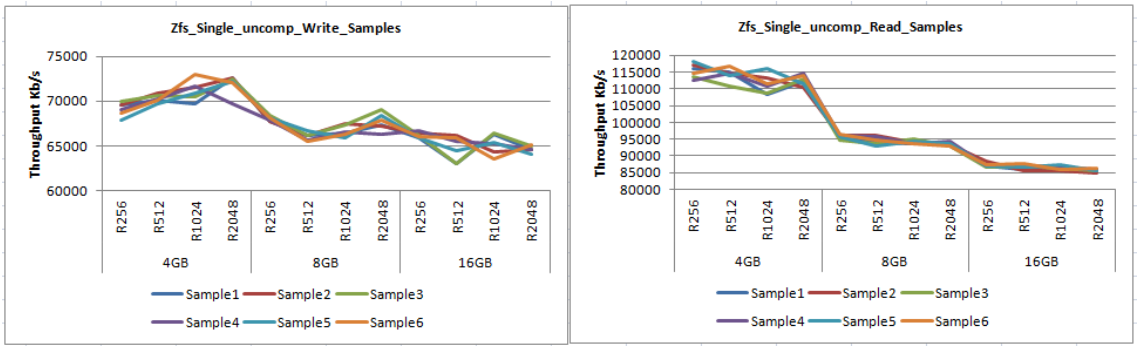


Figure 4.1: Sequential read/write operation for uncompressed Zfs Single

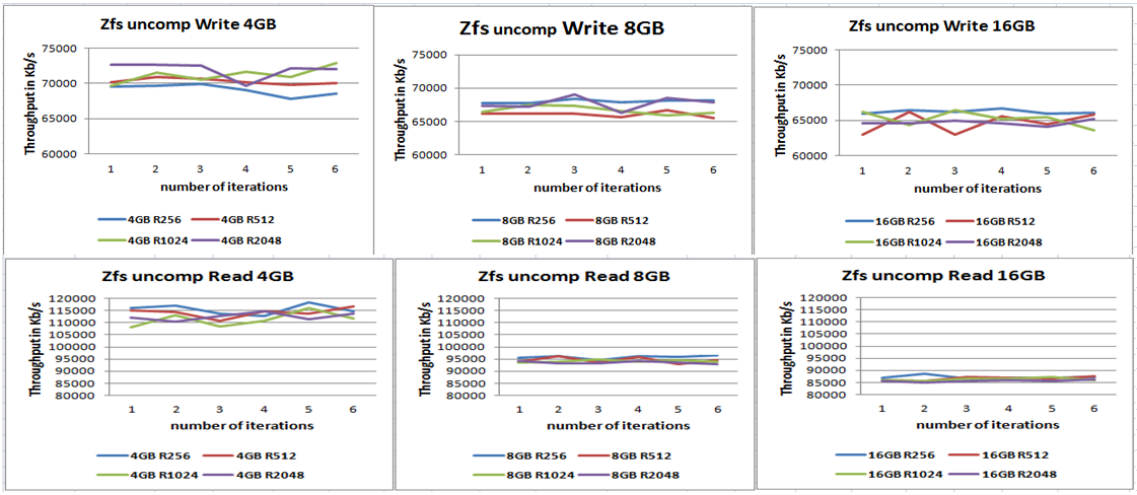


Figure 4.2: Sequential read/write operation for uncompressed Zfs Single per file size

.

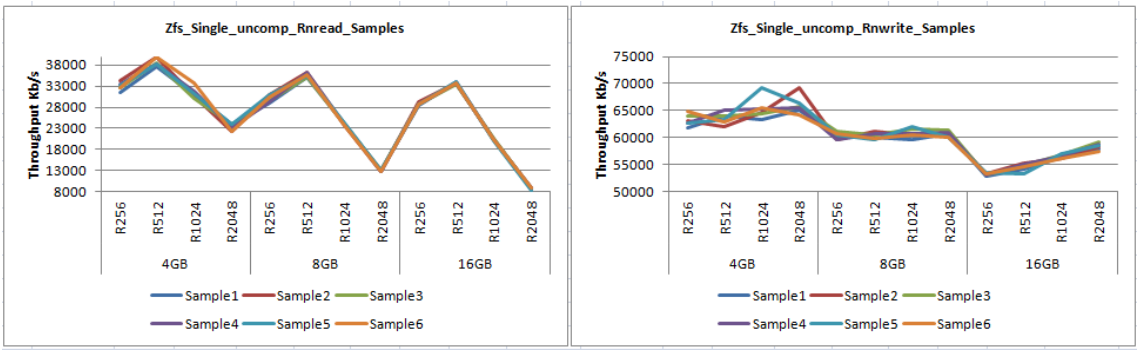


Figure 4.3: Random read/write operation for uncompressed Zfs single

.

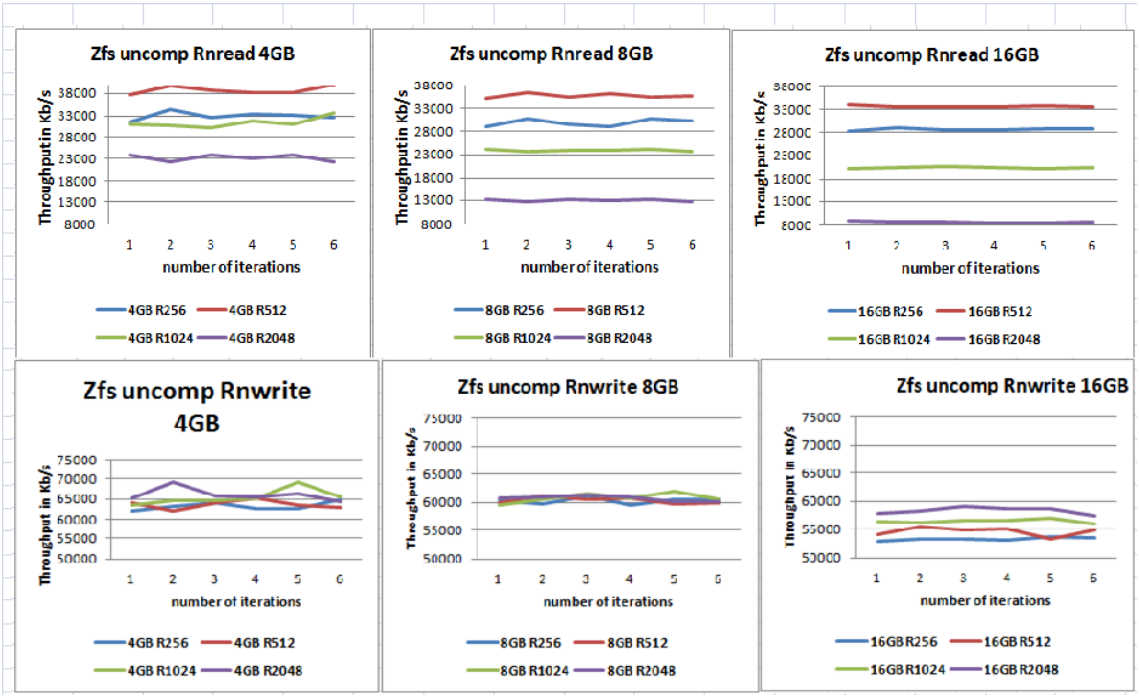


Figure 4.4: Random read/write operation for uncompressed Zfs Single per file size

.

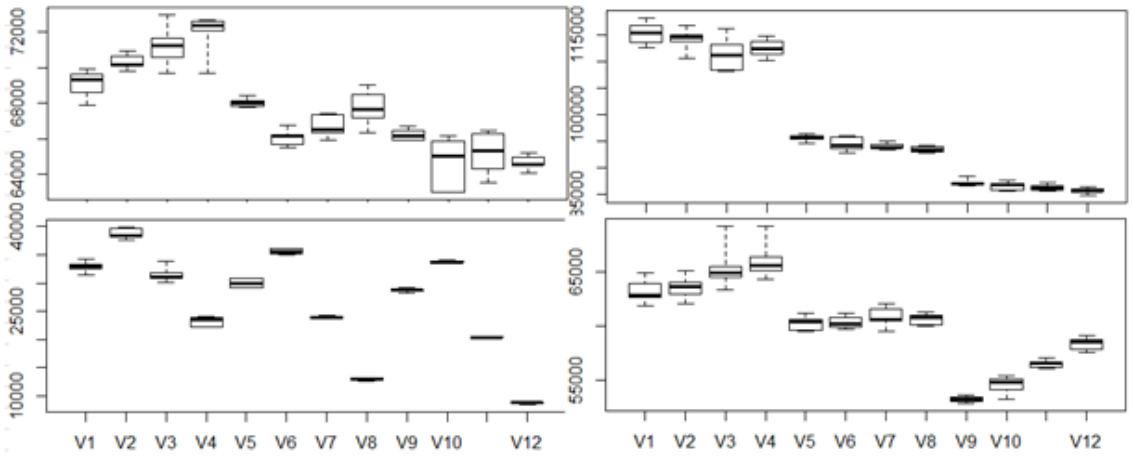


Figure 4.5: write/Read operation for uncompressed Zfs Single in boxplot

.

The boxplot presentation needs some explanation. Here the Y axis represents the amount of kilobytes read or written to disk in thousands and The X axis represents the variables from V1 to V12.
These variables are represents the four records multiplied by three files which has 12 combinations.
256, 512, 1024 and 2048 records times 4, 8, and 16 GB file sizes. There are four boxes from left to right. The first box is sequential write, the second is sequential read, the third is random read and the fourth is random write. Boxplot

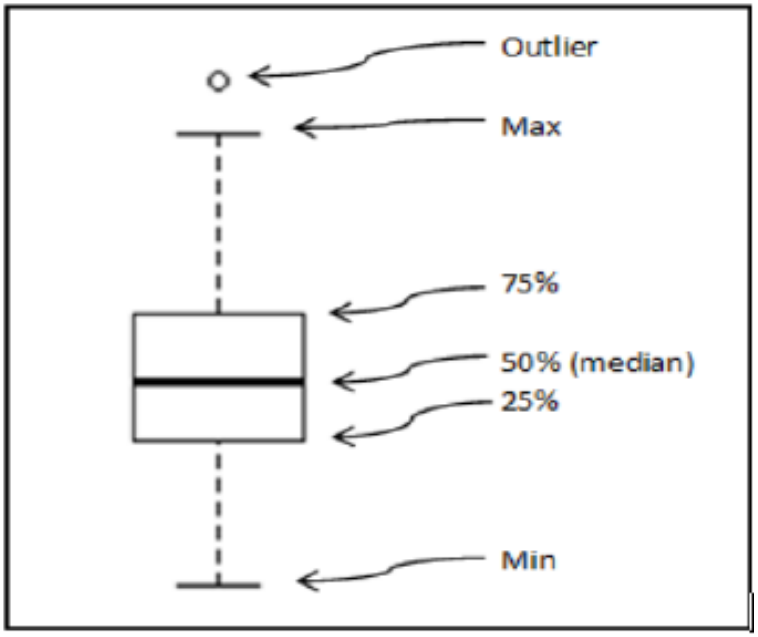


Figure 4.6: Boxplot values

.

shows where the 25, 50, and 75 percent quartiles of the values are concentrated plus the data min, max, and outliers if any. Outliers are values in the data which are very distant from the rest of the data.

4.1.2 Performance benchmarking test Results for Zfs Single compressed

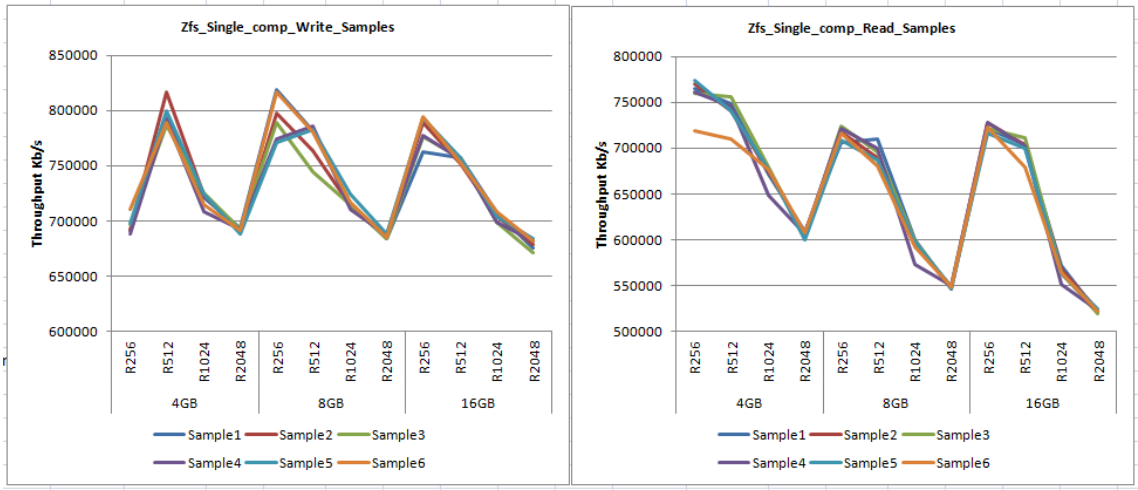


Figure 4.7: Sequential read/write operation for compressed Zfs Single

.

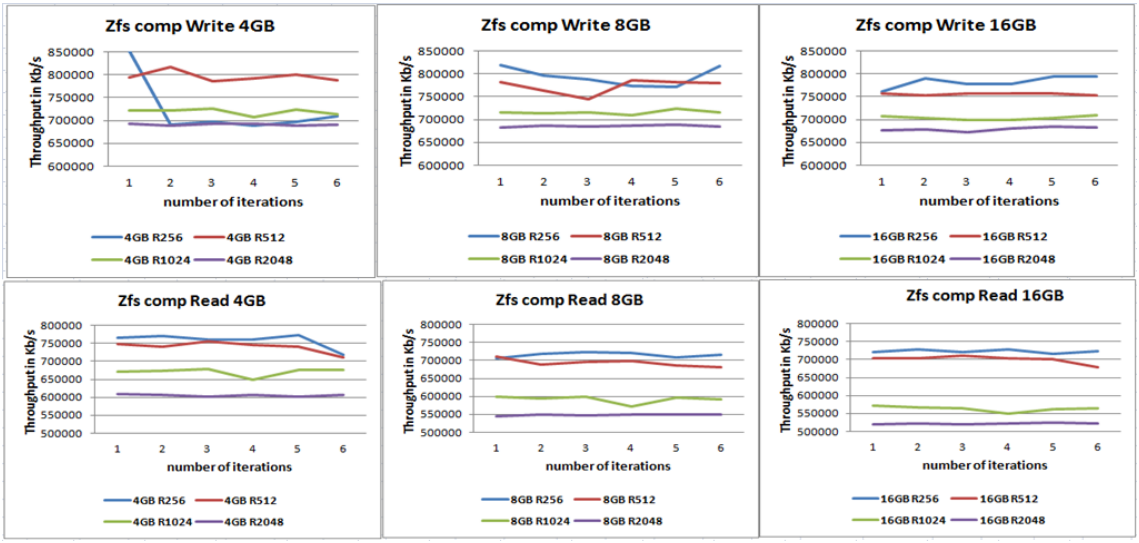


Figure 4.8: Sequential read/write operation for compressed Zfs Single per file size

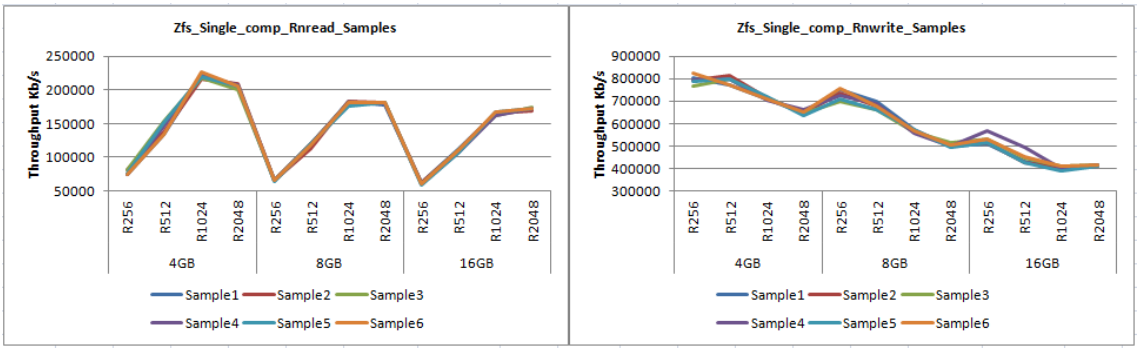


Figure 4.9: Random read/write operation for compressed Zfs Single

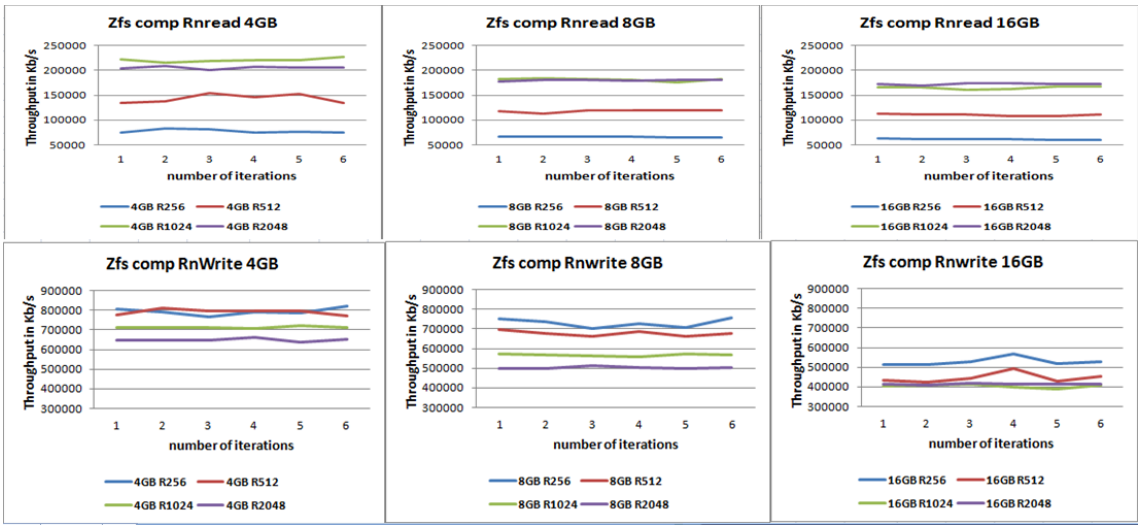


Figure 4.10: Random read/write operation for compressed Zfs Single perfile size

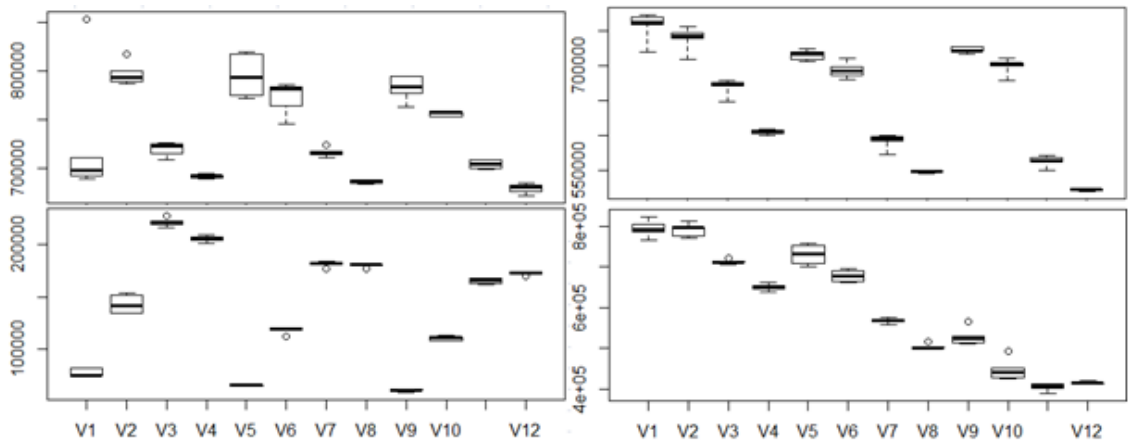


Figure 4.11: Sequential and Random write/Read operation for compressed Zfs in boxplot

.

4.2 Performance benchmarking test Results for Btrfs Single

This section presents the results obtained from sequential read/write and random read/write tests of Iozone. The results are the output of 6 runs and are plotted below.
For the results presented in this section, results are limited to four record sizes: 256KB, 512KB, 1024KB and 128KB.
The graphs below show both sequential and random operations performance of a single Btrfs disk for uncompressed and compressed features.
Here the Y axis represents the amount of kilobytes read or written to disk in thousands. The X axis represents the number of iterations. Excel graphs (line and bar graphs) are used to show the pattern of the raw data values.

4.2.1 Performance benchmarking test Results for Btrfs Single uncompressed

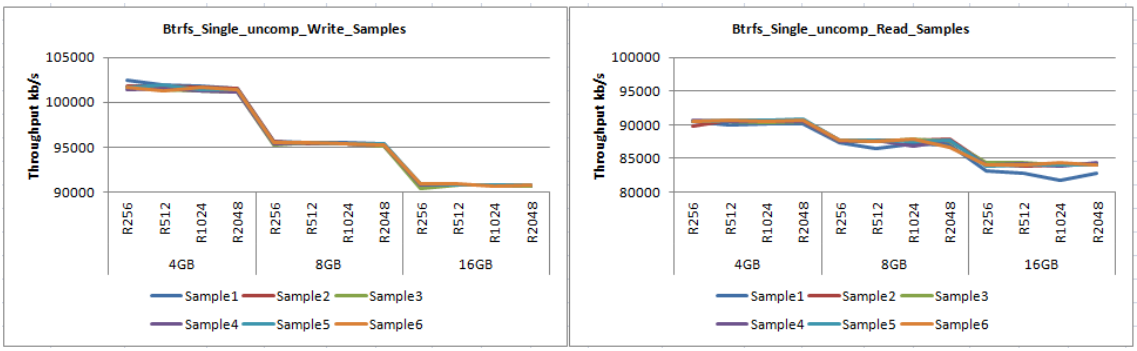


Figure 4.12: Sequential read/write operation for uncompressed Btrfs Single

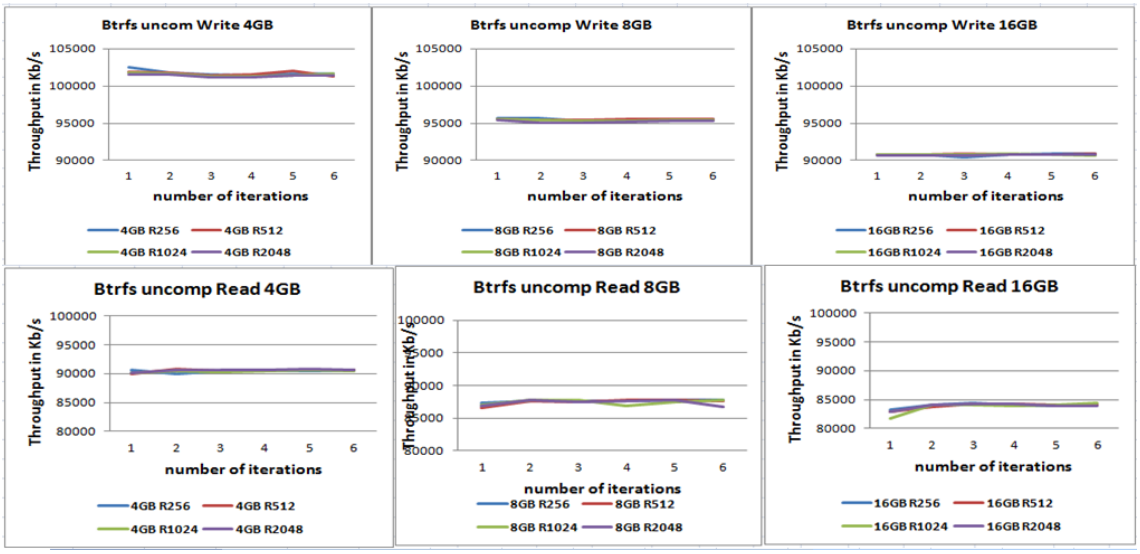


Figure 4.13: Sequential read/write operation for uncompressed Btrfs Single per file size

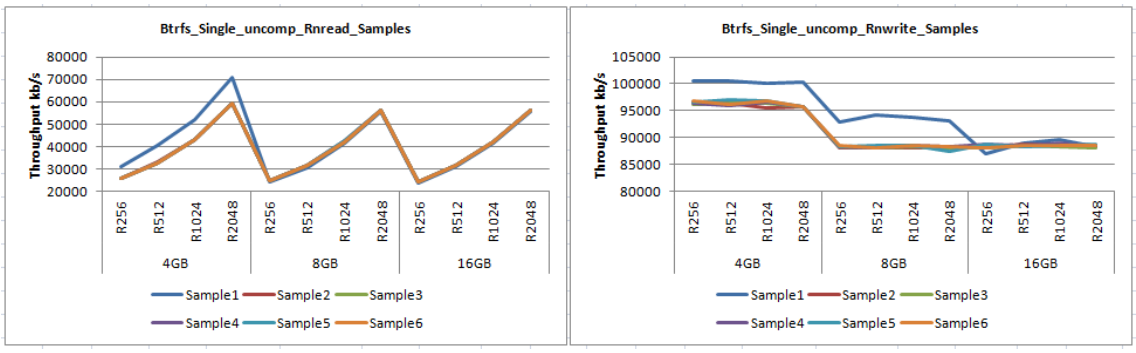


Figure 4.14: Random read/write operation for uncompressed Btrfs Single

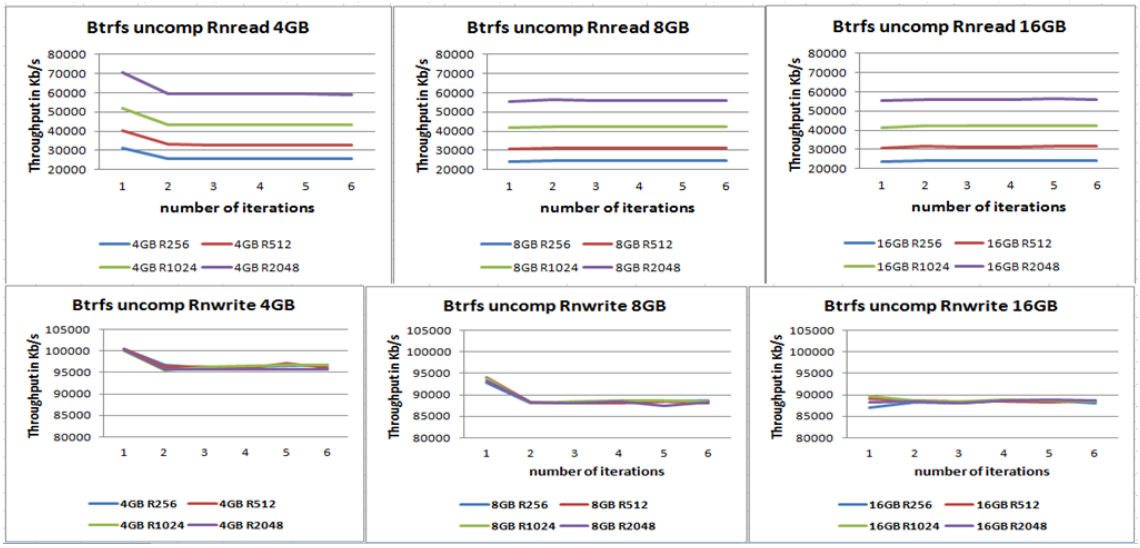


Figure 4.15: Random read/write operation for uncompressed Btrfs Single per file size

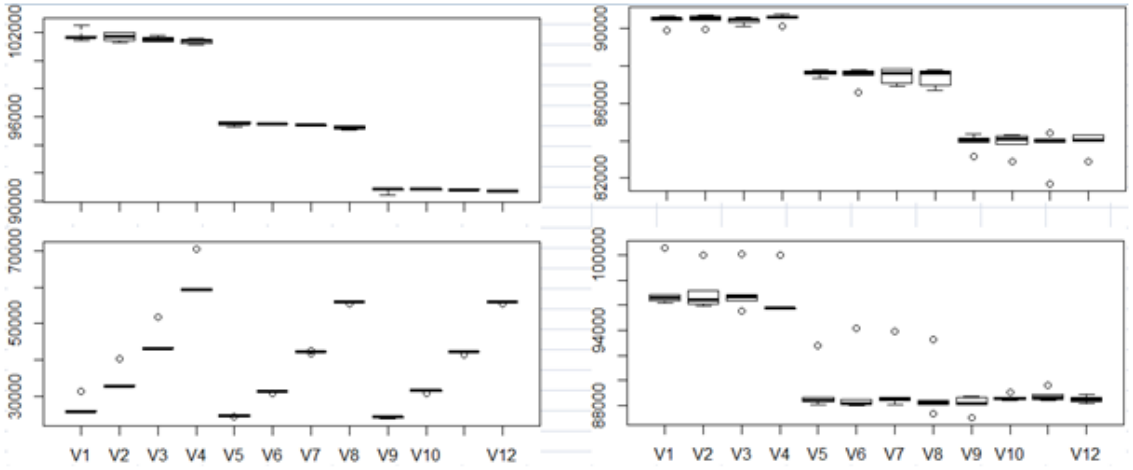


Figure 4.16: write/Read operation for uncompressed Btrfs in boxplot

The boxplot presentation needs some explanation. Here the Y axis represents the amount of kilobytes read or written to disk in thousands and The X axis represents the variables from V1 to V12.
These variables are represents the four records multiplied by three files which has 12 combinations.
256, 512, 1024 and 2048 records times 4, 8, and 16 GB file sizes. There are four boxes from left to right. The first box is sequential write, the second is sequential read, the third is random read and the fourth is random write.

4.2.2 Performance benchmarking test Results for Btrfs Single Compressed

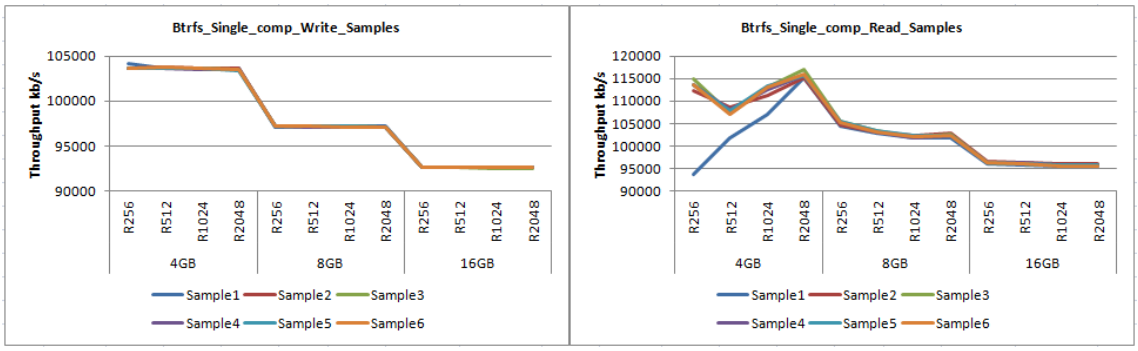


Figure 4.17: Sequential read/write operation for compressed Btrfs Single

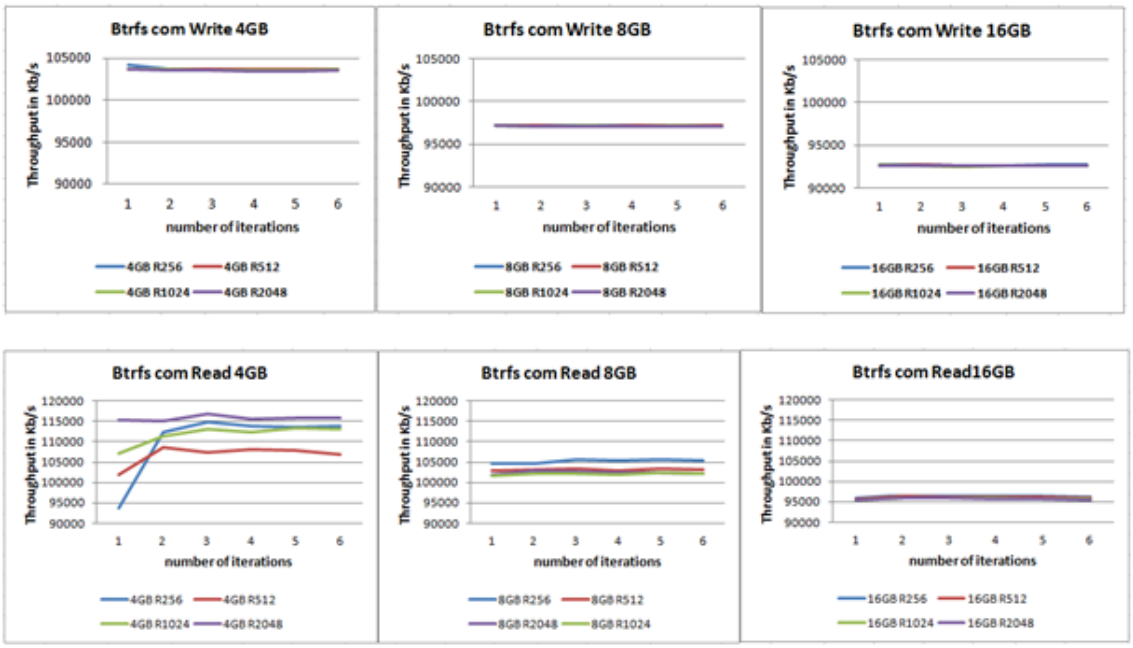


Figure 4.18: Sequential read/write operation for compressed Btrfs Single per file size

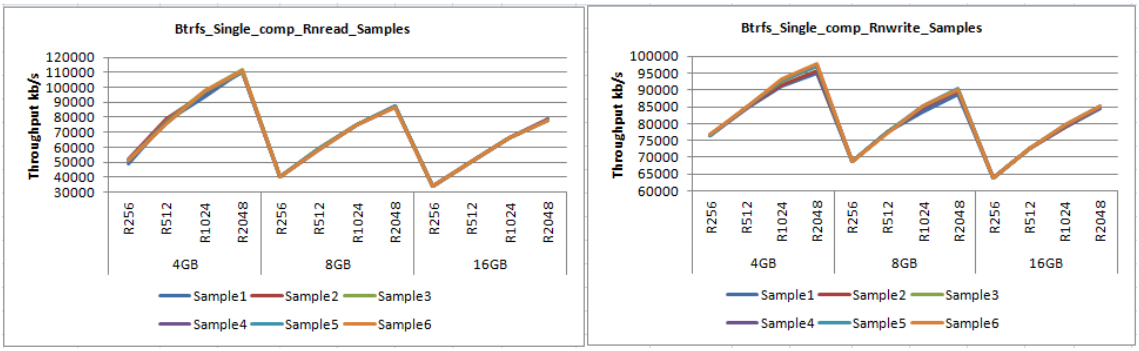


Figure 4.19: Random read/write operation for compressed Btrfs Single

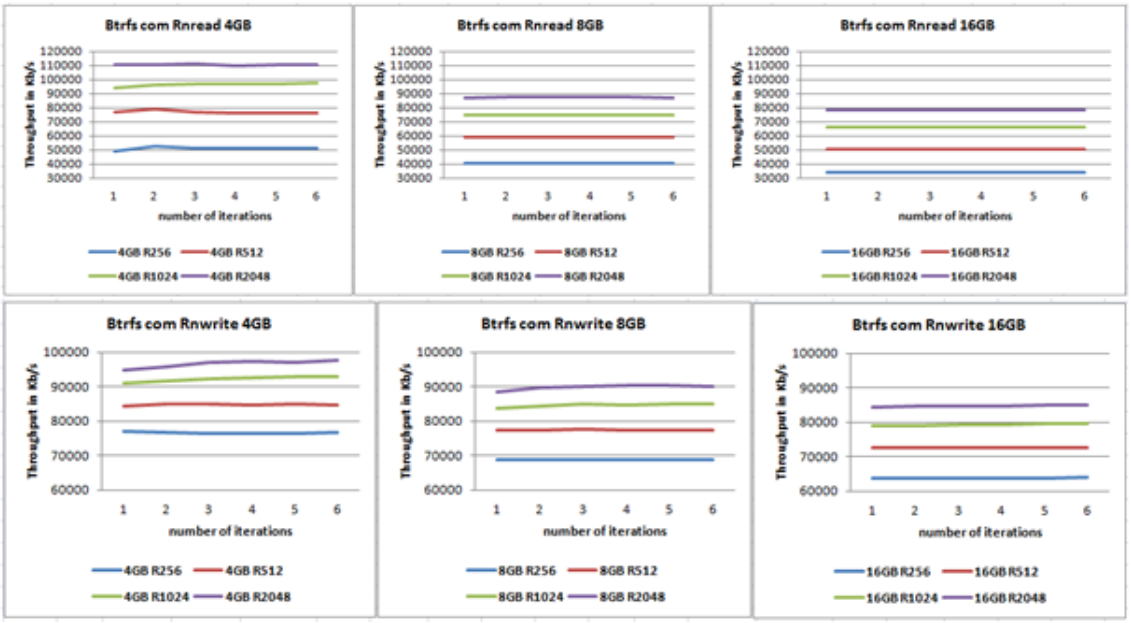


Figure 4.20: Random read/write operation for compressed Btrfs Single per file size

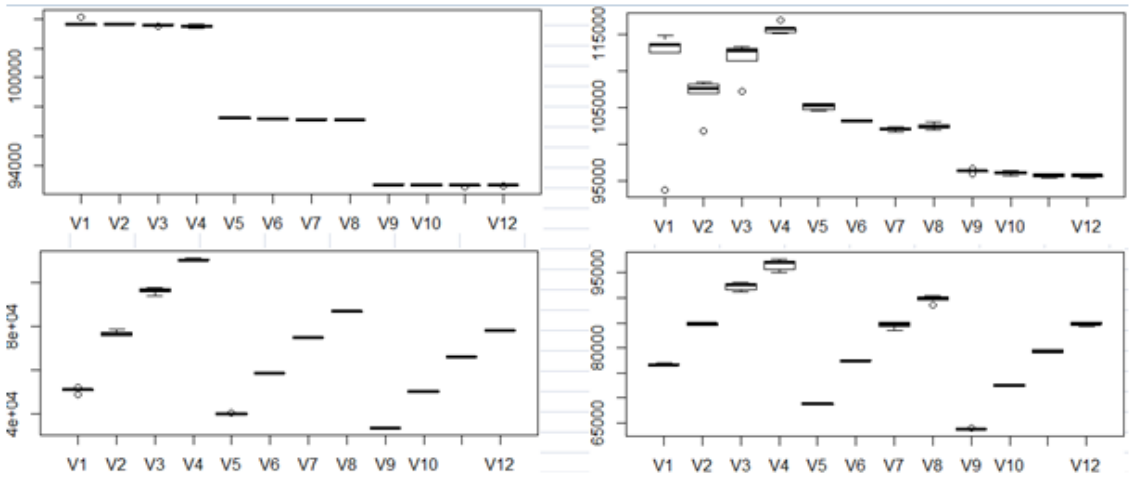


Figure 4.21: Read/Write operation for compressed Btrfs in boxplot

4.3 Performance benchmarking test Results for Zfs Raidz1 uncompressed

This section presents the results obtained from sequential read/write and random read/write tests of Iozone. The results are the output of 6 runs and are plotted below.
For the results presented in this section, results are limited to four record sizes: 256KB, 512KB, 1024KB and 128KB.
The graphs below show both sequential and random operations performance of a Raidz1 zfs disk for uncompressed features.

Here the Y axis represents the amount of kilobytes read or written to disk in thousands. The X axis represents the number of iterations. Excel graphs (line and bar graphs) are used to show the pattern of the raw data values.

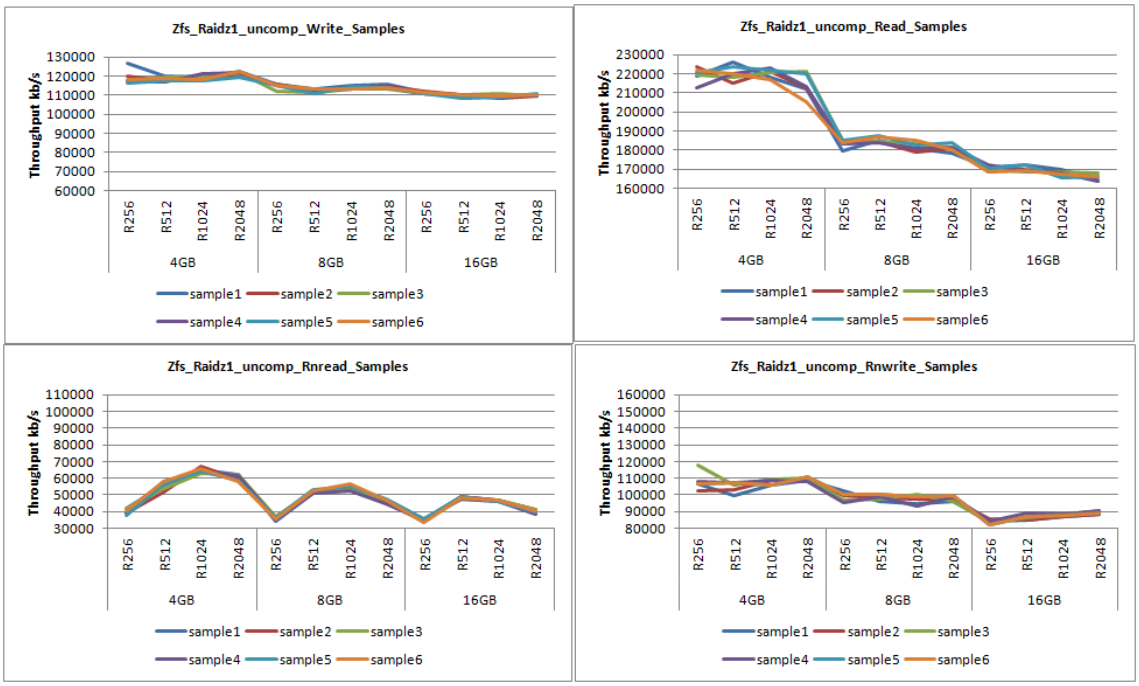


Figure 4.22: Write/Read operation for uncompressed Zfs Raidz1

4.4 Performance benchmarking test Results for Zfs Raidz1 compressed

This section presents the results obtained from sequential read/write and random read/write tests of Iozone. The results are the output of 6 runs and are plotted below.
For the results presented in this section, results are limited to four record sizes: 256KB, 512KB, 1024KB and 128KB.
The graphs below show both sequential and random operations performance of a Raidz1 zfs disk for compressed features.

Here the Y axis represents the amount of kilobytes read or written to disk in thousands. The X axis represents the number of iterations. Excel graphs (line and bar graphs) are used to show the pattern of the raw data values.

Figure 4.23: Write/Read operation for compressed Zfs Raidz1

4.5 Performance benchmarking test Results for Btrfs Raid5 uncompressed

This section presents the results obtained from sequential read/write and random read/write tests of Iozone. The results are the output of 6 runs and are plotted below.
For the results presented in this section, results are limited to four record sizes: 256KB, 512KB, 1024KB and 128KB.
The graphs below show both sequential and random operations performance of a Btrfs Raid5 disk for uncompressed features.

Here the Y axis represents the amount of kilobytes read or written to disk in thousands. The X axis represents the number of iterations. Excel graphs (line and bar graphs) are used to show the pattern of the raw data values.

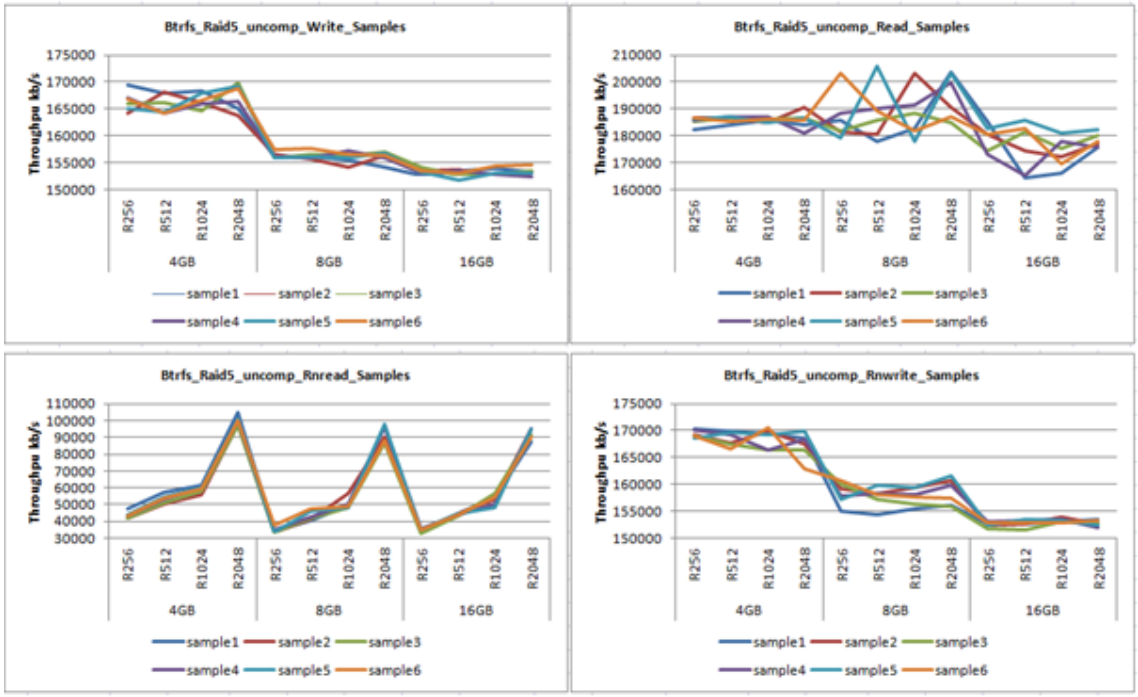


Figure 4.24: Write/Read operation for uncompressed Btrfs Raid5

4.6 Performance benchmarking test Results for Btrfs Raid5 compressed

This section presents the results obtained from sequential read/write and random read/write tests of Iozone. The results are the output of 6 runs and are plotted below.
For the results presented in this section, results are limited to four record sizes: 256KB, 512KB, 1024KB and 128KB.
The graphs below show both sequential and random operations performance of a Btrfs Raid5 disk for compressed features.

Here the Y axis represents the amount of kilobytes read or written to disk in thousands. The X axis represents the number of iterations. Excel graphs (line and bar graphs) are used to show the pattern of the raw data values.

Figure 4.25: Write/Read operation for compressed Btrfs Raid5

4.7 Performance benchmarking test Results for Ext4 Single uncompressed

This section presents the results obtained from sequential read/write and random read/write tests of Iozone. The results are the output of 6 runs and are plotted below.
For the results presented in this section, results are limited to four record sizes: 256KB, 512KB, 1024KB and 128KB.
The graphs below show both sequential and random operations performance of a Ext4 single disk for uncompressed features.

Here the Y axis represents the amount of kilobytes read or written to disk in thousands. The X axis represents the number of iterations. Excel graphs (line and bar graphs) are used to show the pattern of the raw data values.

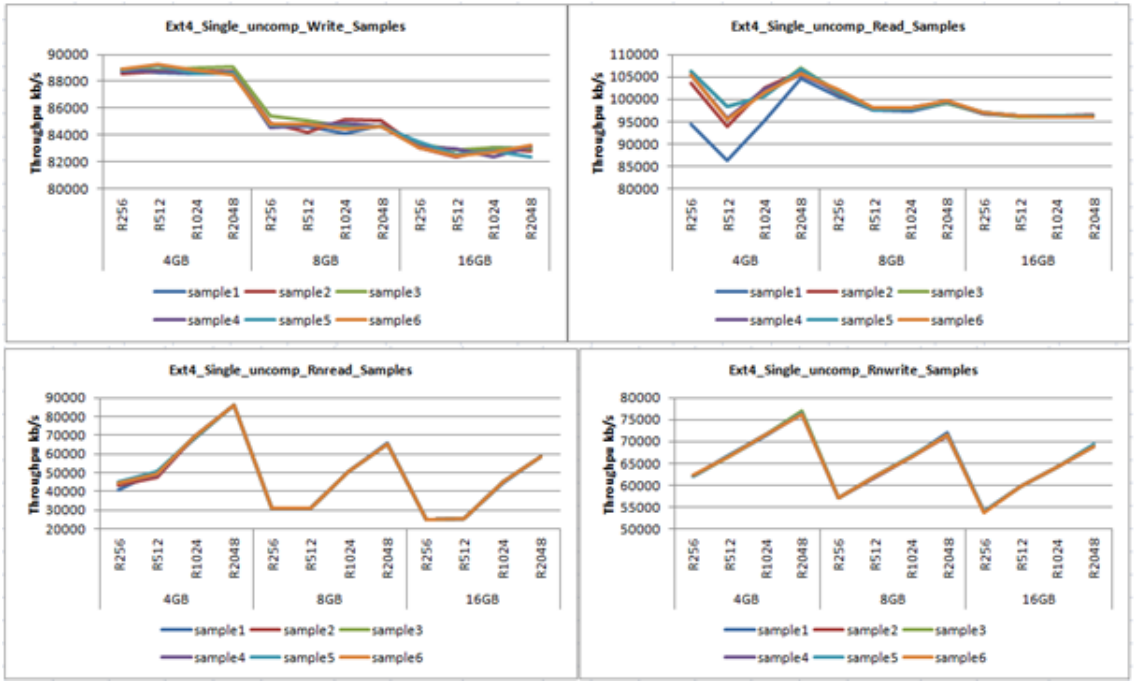


Figure 4.26: Write/Read operation for uncompressed Ext4 Single

4.8 Performance benchmarking test Results for Ext4 Raid5 uncompressed

This section presents the results obtained from sequential read/write and random read/write tests of Iozone. The results are the output of 6 runs and are plotted below.
For the results presented in this section, results are limited to four record sizes: 256KB, 512KB, 1024KB and 128KB.
The graphs below show both sequential and random operations performance of a Ext4 Raid5 disk for uncompressed features.

Here the Y axis represents the amount of kilobytes read or written to disk in thousands. The X axis represents the number of iterations. Excel graphs (line and bar graphs) are used to show the pattern of the raw data values.

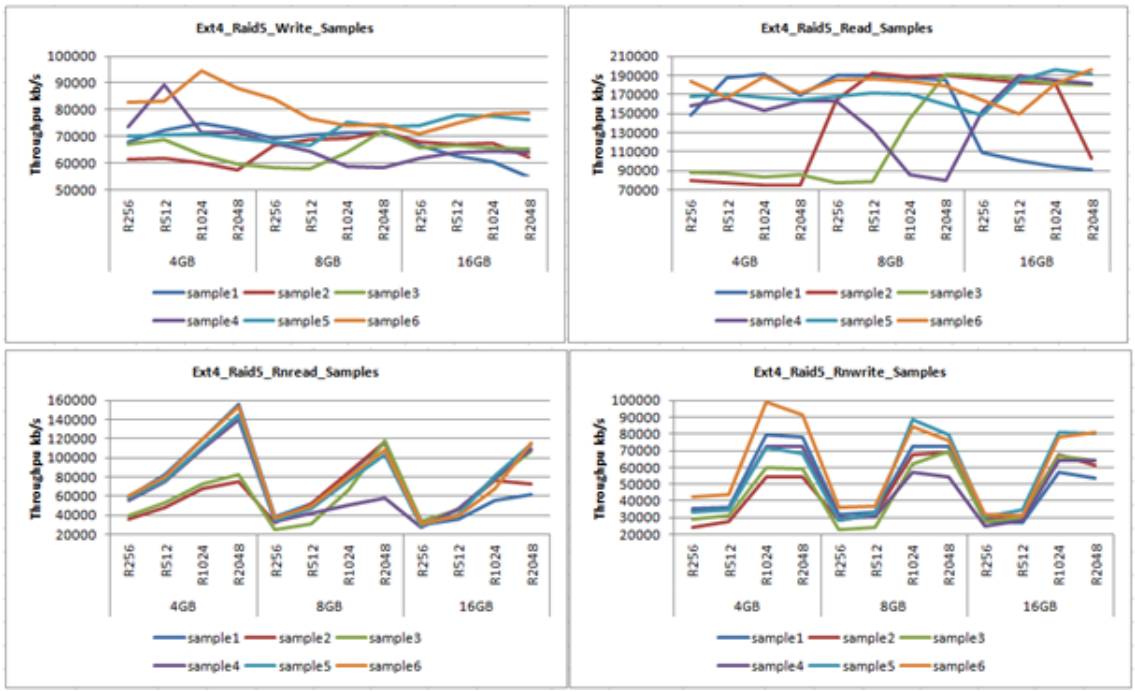


Figure 4.27: Write/Read operation for uncompressed Ext4 Raid5

Chapter 5

Analysis

This chapter presents the analysis of the results of the various tests described in the previous chapter.
The main purpose of this research is to investigate the performance impact of compression feature on zfs and btrfs file systems. The benchmarking and result collection are already done in the previous chapter.
Boxplots and MS Excel line and bar graphs are used to show the results. The three types of graphs used to express results are all important as they all give different kind of information about the samples collected.
These are line graph to show the how the data look like at every single point and its trend, boxplot to show general summary of the whole data (like min, max, median and the 25, 50, and 75 percent quartiles of the raw data plus the outliers if any), and bargraphs to compare single values of different data (like to compare average, min, max, STDEV, MEM, CONF, etc. . .).

Terms	Meaning
Min	Sample's Minimum value
Max	Sample's Maximum value
Mean	Sample's average value
Median	Sample's middle value after listing in ascending or descending order
STDEV	Sample's Standard Deviation value $=\frac{1}{N-1}\sqrt{\sum(S-M)^2}$, Where, S = sample value, N is sample's size, and M is Sample's Mean value. This value shows in how extent the samples are scattered or it just tells us how much each sample value is far from the Sample's Mean. So; obviously, lesser STDEV is better, as it shows the samples are less scattered.
SEM	Standard Error of the Mean $=\frac{STDEV}{\sqrt{N}}$. It tells us the precision of the sample's Mean (how far the sample's Mean is from the true Mean). It is always less than STDEV. As the sample size (N) increases, SEM will be much smaller than STDEV.
CONF	95% CONFIDENCE INTERVAL of the given samples using excel [CONFIDENCE (0.05, **STDEV**, sample size)]. It can be interpreted as with 95% confidence the true mean lies in between **(M – CONF) & (M + CONF)**, or **[(M – CONF) < TRUE MEAN < (M + CONF)]**. So; lesser CONF value is better, as it shows the sample's Mean value is closer to the true mean value.

Figure 5.1: Explanation of of statistical terms used in this thesis

5.1 Zfs Compression vs default Comparison for Single Disk

For the results presented in this section, results are limited to four record sizes: 256KB, 512KB, 1024KB and 128KB.

The two graphs below show both sequential and random operations performance of a single zfs disk for both compressed and uncompressed features. The performance of both features are very much different, and there is significant difference on the achieved average throughput.
The tables summarize the performance differences between the default and compression feature for the Zfs single disk file systems tests
The noted differences in performance run across all record sizes and file sizes unless explicitly stated otherwise. The calculation is taken as the percentage of increase of the mean Performance of Compression feature against default feature for table.

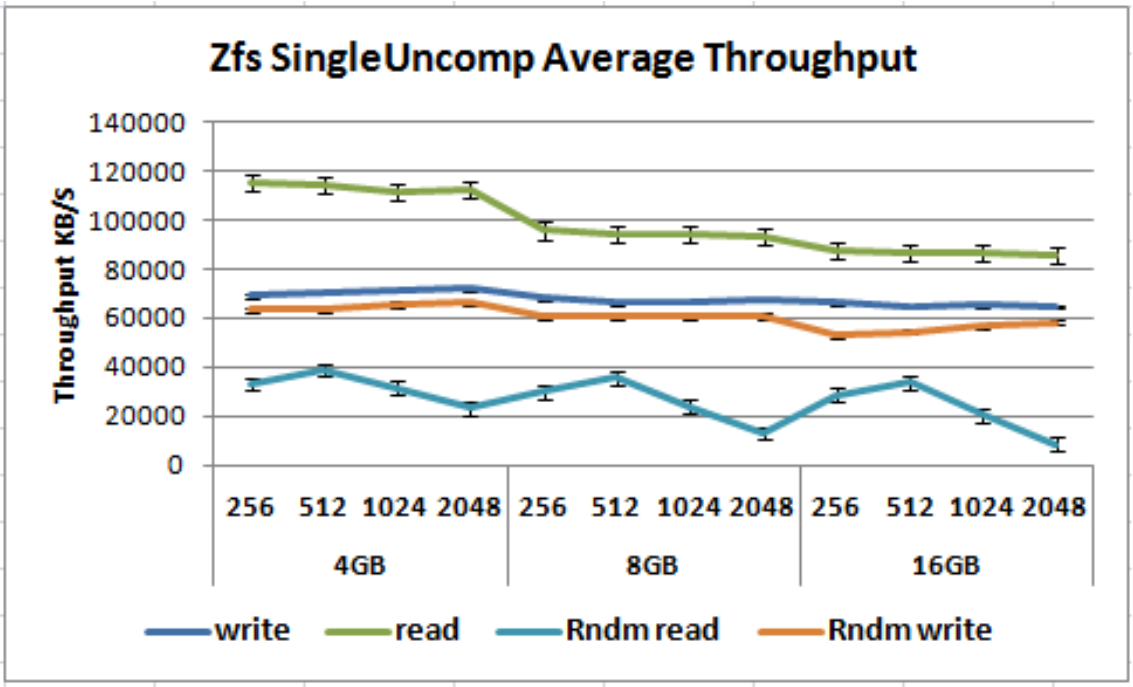


Figure 5.2: Zfs single average throughput uncompressed

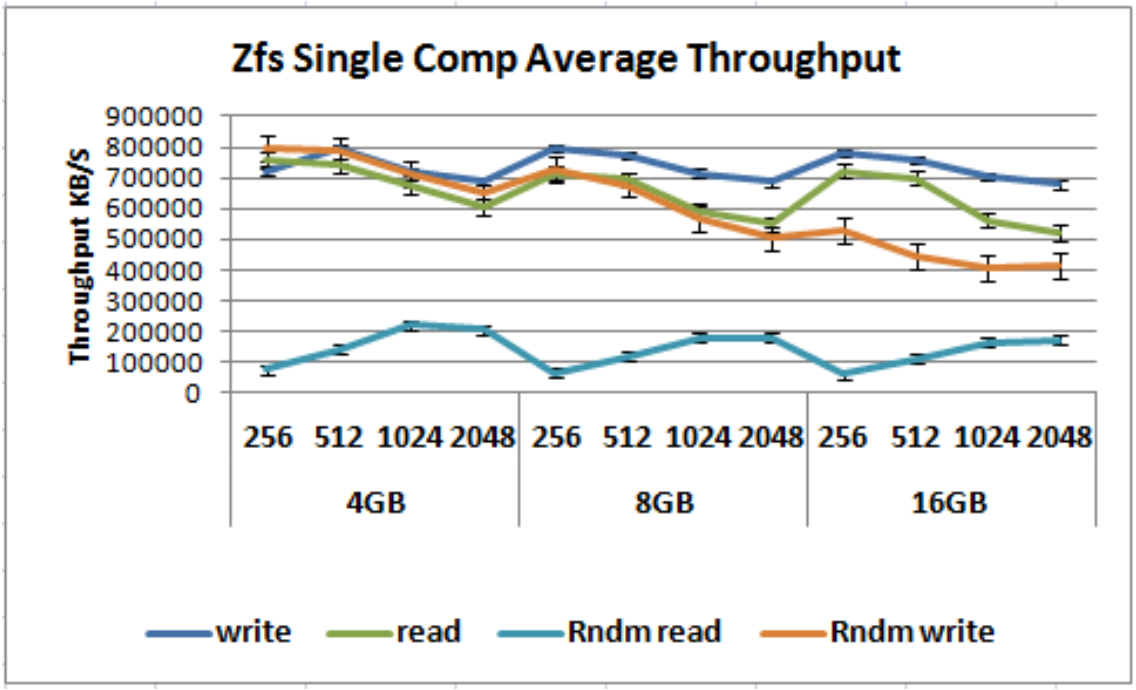


Figure 5.3: Zfs single average throughput compressed

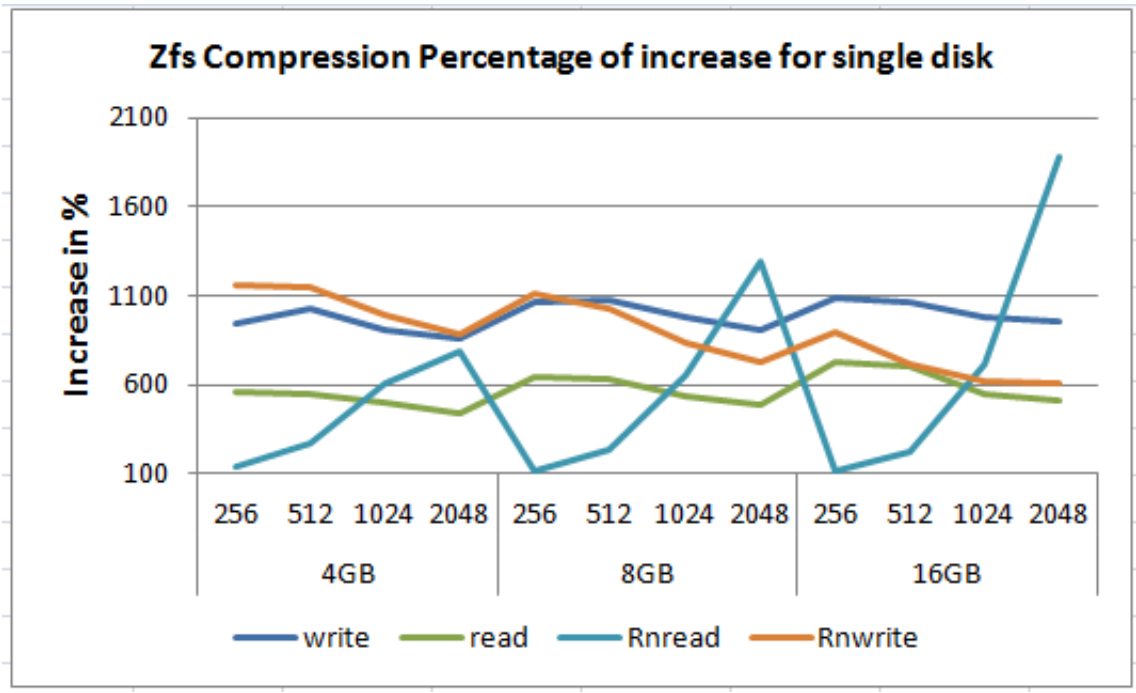


Figure 5.4: Zfs single average throughput percentage of increase due to compression

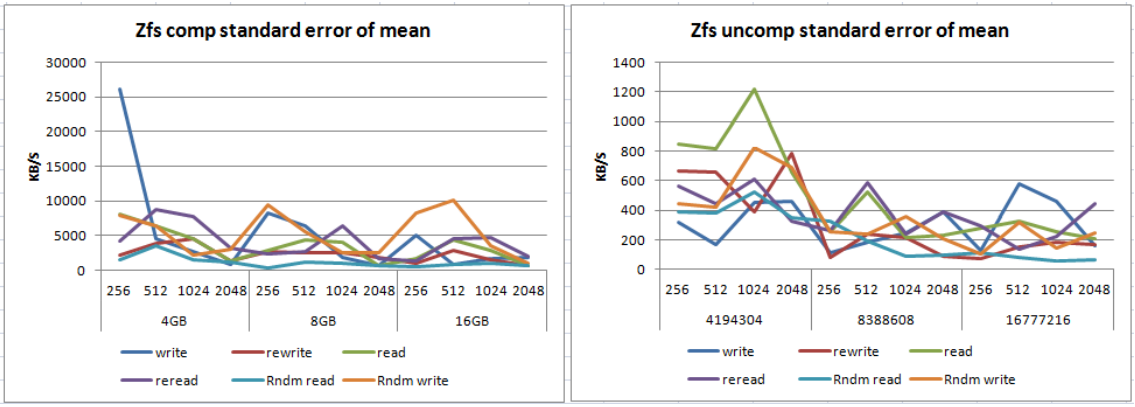


Figure 5.5: Zfs single standard error of mean comparison

Test	**Avg Throughput with SEM Increase**	**Disk usage Increase by IOPS and**	**Cpu usage Increase by SYS, IOWAIT, IDLE**
Sequential Write	900%, high SEM for lower record 2300%	-91%, -30%	940%, -66%, -52%
Sequential Read	600%, 700%	-29%, -43%	1600%, -74%, -37%
RandomRead	100%, 200%, 600%, 1200% per Record and 400%, 500%, 700% per File and 500% SEM	-87%, -5%	124%, -5%, 0%
RandomWrite	900%, 1400%	-82%, -22%	1160%, -63%, -62%

Table 5.1: Significant Performance Differences for Zfs Single compression feature

5.1.1 Zfs Single Sequential Write Operations comparison

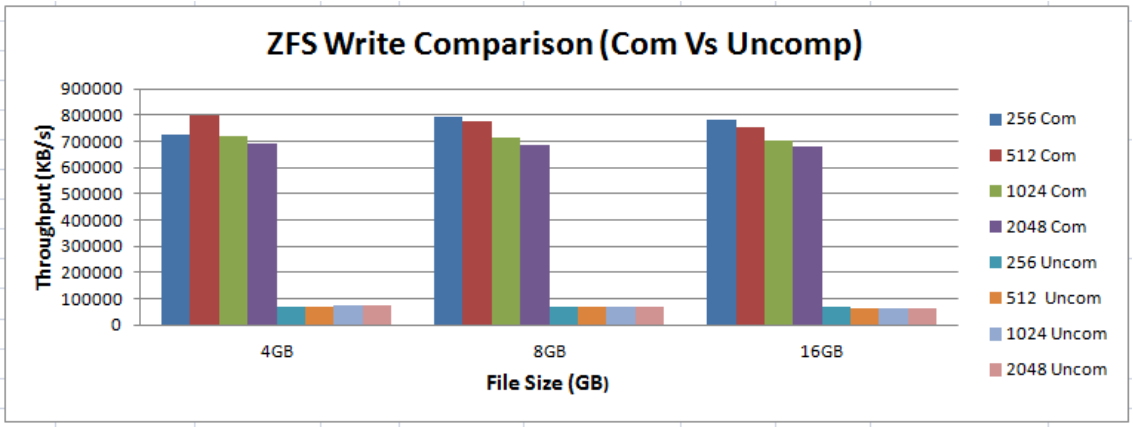


Figure 5.6: Zfs Single Write Avg Throughput Comparison

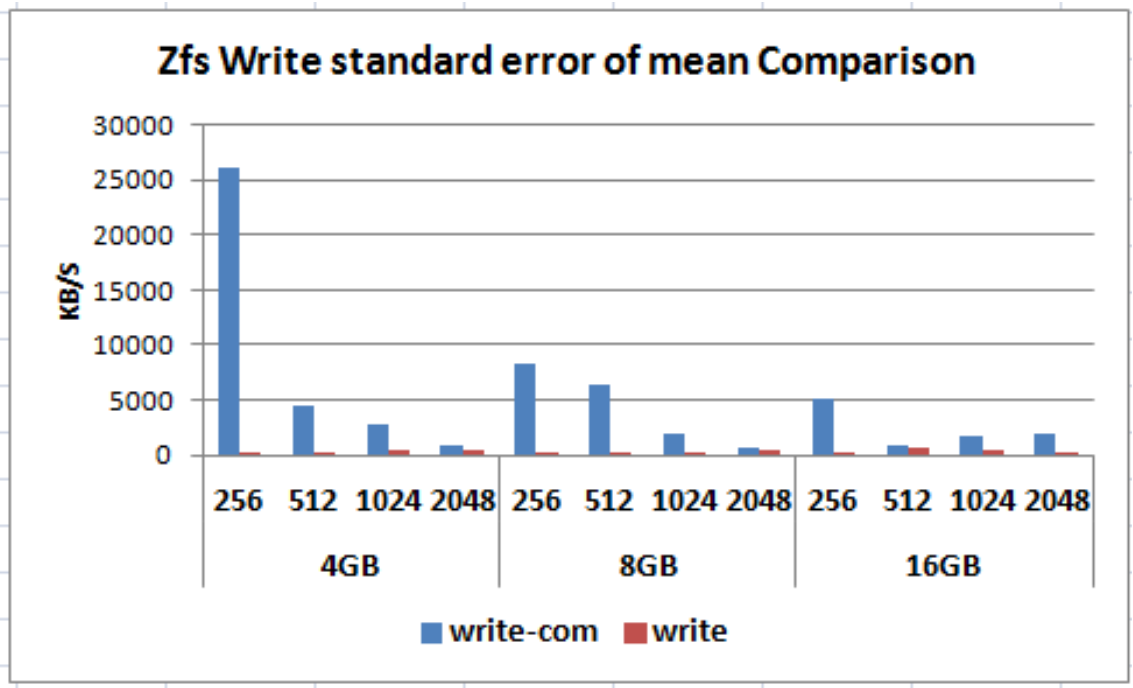


Figure 5.7: Zfs Single write throughput standard error of mean comparison

As can be seen from figures 5.2, 5.3, 5.4, 5.5, 5.6 and finally from table 5.1 in the analysis section the impact of compression on the write operation is clearly seen to be very high.
Moreover, the attained throughput difference is similar with respective record size and file size.
This is because the throughput in both uncompressed and compressed mode decreases as the record size and file size increases.

When the percentage of increase of mean throughput of compression to default feature calculated, on average 900% or ten times higher performance is obtained.
In addition the compression feature exhibits higher standard error of mean specially for lower record sizes.
The other noticeable difference observed is disk and cpu usage out put characteristics.
As mentioned numerically in table 5.1 above, compression affected to have lower write IOPS which has decreased almost by 90% times and hence the disc bandwidth utilization ratio has too decreased by 93% times. The only negative impact observed is that the cpu time usage has increased almost 900% or ten times.

5.1.2 Zfs Single Sequential Read Operations comparison

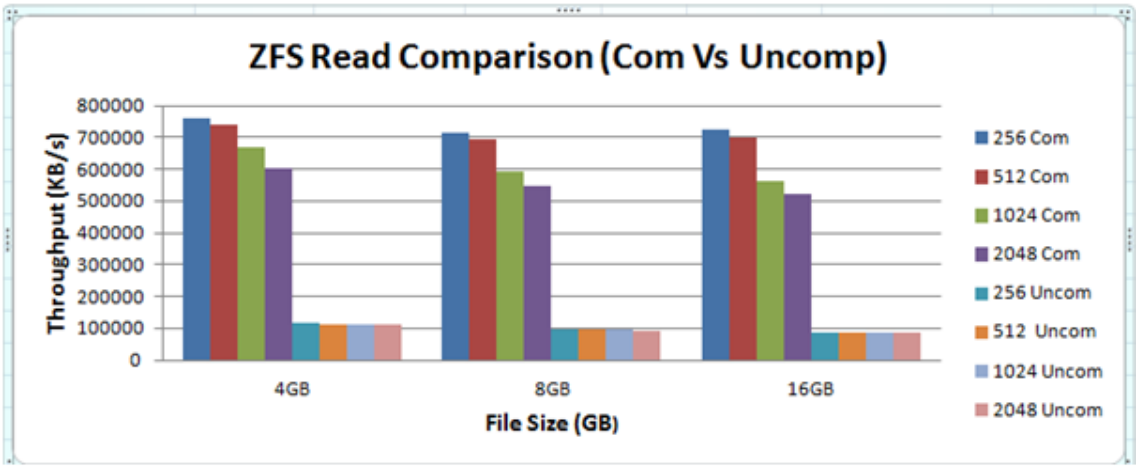


Figure 5.8: Zfs Single Read Average Throughput Comparison

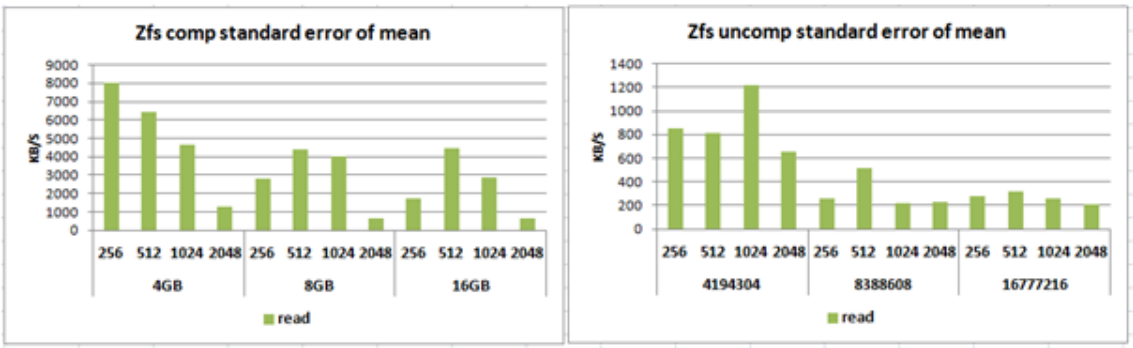


Figure 5.9: Zfs Read throughput standard error of mean Comparison

As can be seen from figure 5.2, 5.3, 5.4, 5.8, 5.9 and finally from table 5.1 in the analysis section the impact of compression on the read operation is clearly

seen to be very high.

The performance of Zfs with its compression feature outshine the performance of Zfs with the default options for all record size and file size as it has been similarly shown with sequentially reading from a file.
The read operation decreases as both record size and file sizes increase. More over the throughput difference remain the same for all file size and record size. The difference is calculated and it is seven times or 600% better than the default option with standard error of mean increase around eight times or 700% higher for compression.
The other noticeable difference observed is disk and cpu usge out put characterstics. As mentioned numerically in table 5.1 above, compression affected to have lower read IOPS which has decreased almost by 90% and hence the disk bandwidth utilization ratio has too decreased by 40%.
The only negative impact observed is that the cpu time usage has increased almost seventeen (17x) times or 1600%.

5.1.3 Zfs Random Read Operations comparison

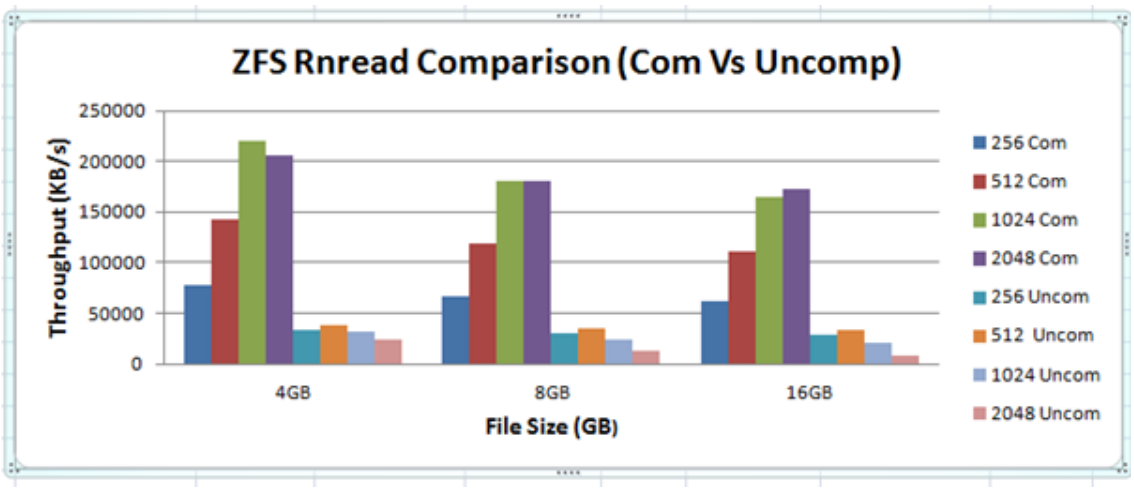


Figure 5.10: Zfs Random Read Avg throughput Comparison

As can be seen from figure 5.2, 5.3, 5.4, 5.10, 5.11 and finally from table 5.1 in the analysis section the impact of compression on the random read operation is clearly seen to be very high.
As it can be seen from the figures, the throughput achieved in randomly reading a file after enabling the compression feature exhibits better performance.
More over one interesting characteristics of random reading as clearly seen from the graph and table is that the performance increases as record size increases and decreases as file size increases for compressed feature where as the performance decreases with file size increase in both compression and default case. throughput comparison between compression and default case varies with both record and file sizes.

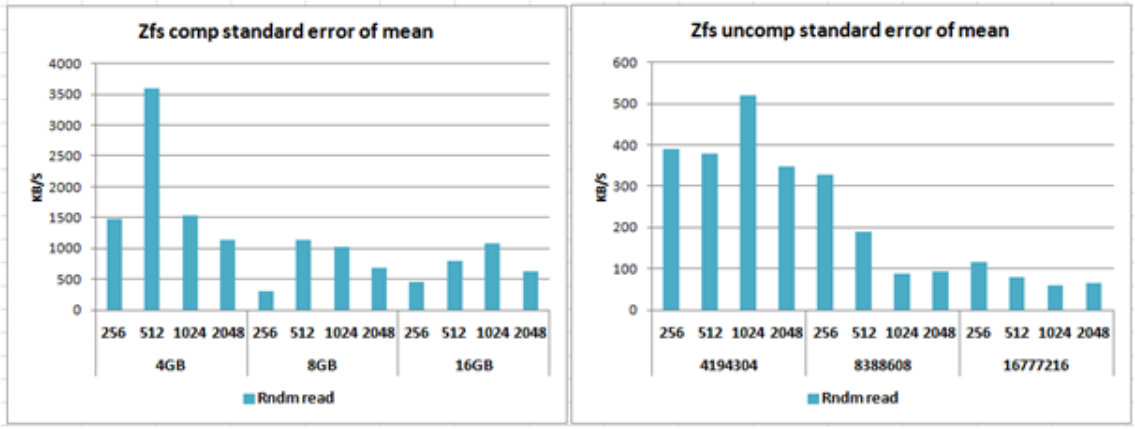


Figure 5.11: Zfs Random Read throughput standard error of mean Comparison

The difference is calculated and it is 585% better than the default option with standard error of mean ratio around six times or 500% higher for compression. The other noticeable difference observed is disk and cpu usge out put characterstics. As mentioned numerically in table 5.1 above, compression affected to have lower random read IOPS which has decreased almost by 99% and hence the disk bandwidth utilization has too decreased by 5%.

The only negative impact observed is that the cpu time usage has increased almost 124%.

5.1.4 Zfs Random Write operations comparison

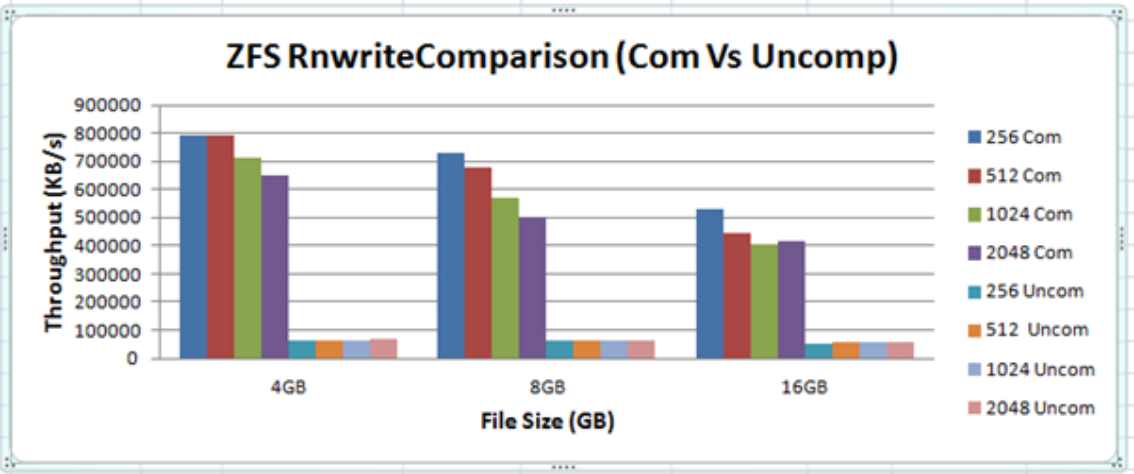


Figure 5.12: zfs Random Write Avg throughput Comparison

As can be seen from figure 5.2, 5.3, 5.4, 5.12, 5.13 and finally from table 5.1 in the analysis section the impact of compression on the random write operation is clearly seen to be very high.

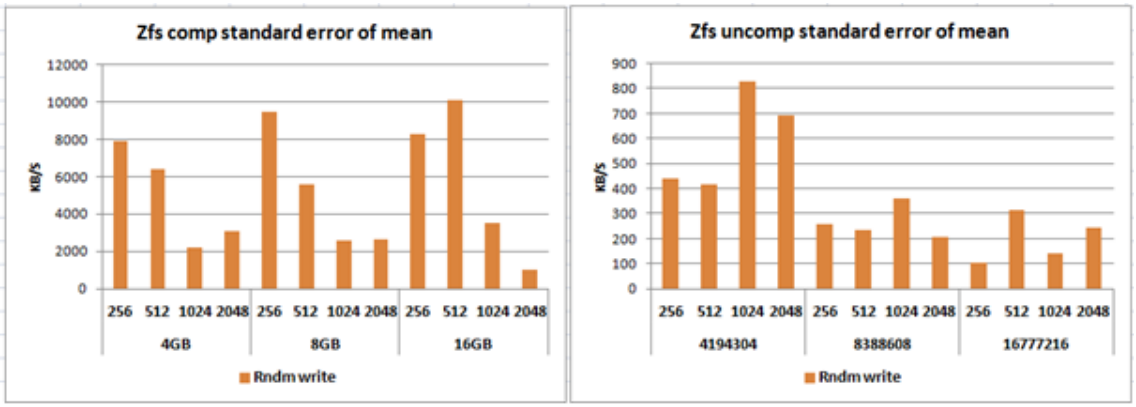


Figure 5.13: zfs Random Write throughput standard error of mean Comparison

As it can be seen in the graphs, Zfs with compression enabled shows better throughput for the single disk random write operation. Randomly writing to a file after enabling compression feature on Zfs shows much a larger performance improvement for all record and file size combinations.
While for compressed feature the performance degrades as file size and record size increases where as the performance with file size and increases with record size instead.
In addition the comparison is more or less the same for the different scenarios given. The difference is calculated and it is ten times or 900% better than the default option with standard error of mean ratio around fifteentimes or 1400% higher for compression.
The other noticeable difference observed is disc and cpu usge out put characterstics.
As mentioned numerically in table 5.1 above, compression affected to have lower random read IOPS which has decreased almost by 82% and hence the disk bandwidth utilization has too decreased by 22%.
The only negative impact observed is that the cpu time usage has increased almost 1160%.

5.2 Btrfs Single Compression vs default Comparison

For the results presented in this section, results are limited to four record sizes: 256KB, 512KB, 1024KB and 128KB.

The two graphs below show both sequential and random average throughputof a single Btrfs disk for both compressed and uncompressed features.
The tables summarize the performance differences between the default and compression feature for the Btrfs single disk file systems tests
The noted differences in performance run across all record sizes and file sizes unless explicitly stated otherwise. The calculation is taken as the percentage of increase of the mean Performance of Compression feature against default feature for table.

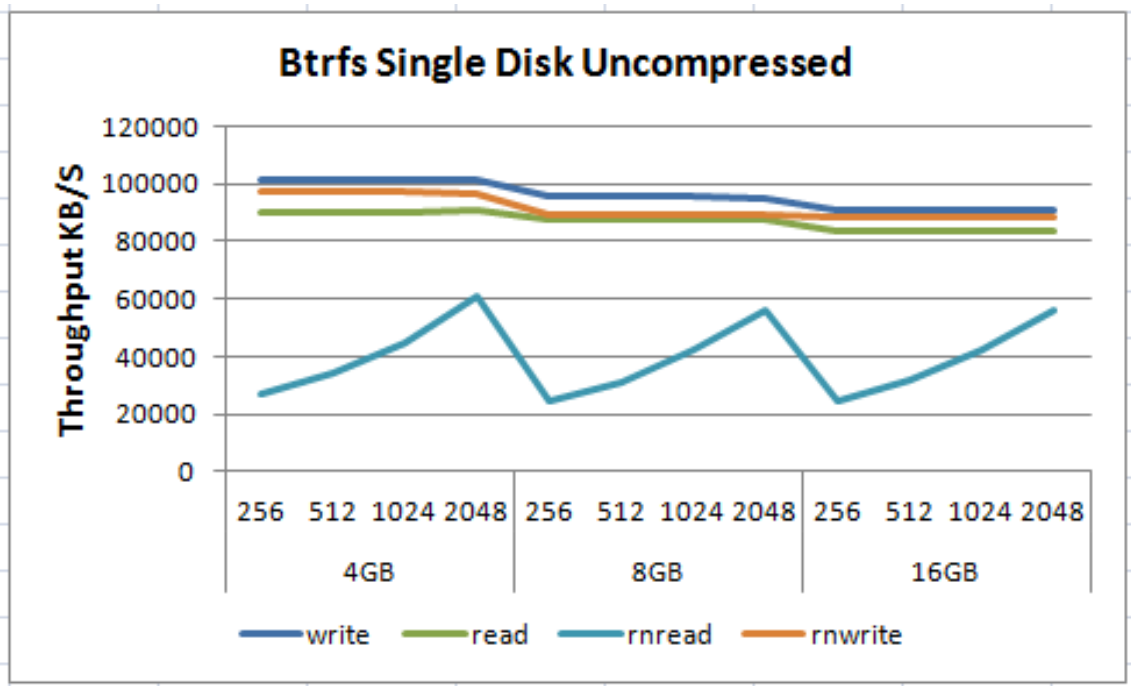


Figure 5.14: Btrfs single disk uncompressed average Throughput

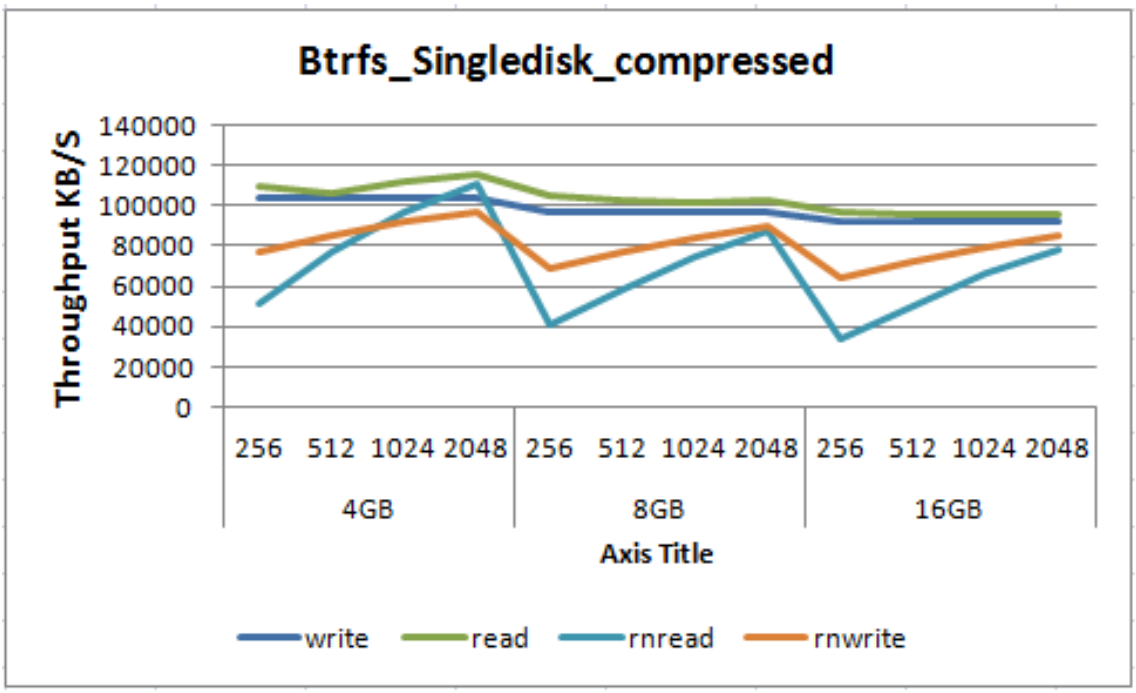


Figure 5.15: Btrfs single disk compressed average Throughput

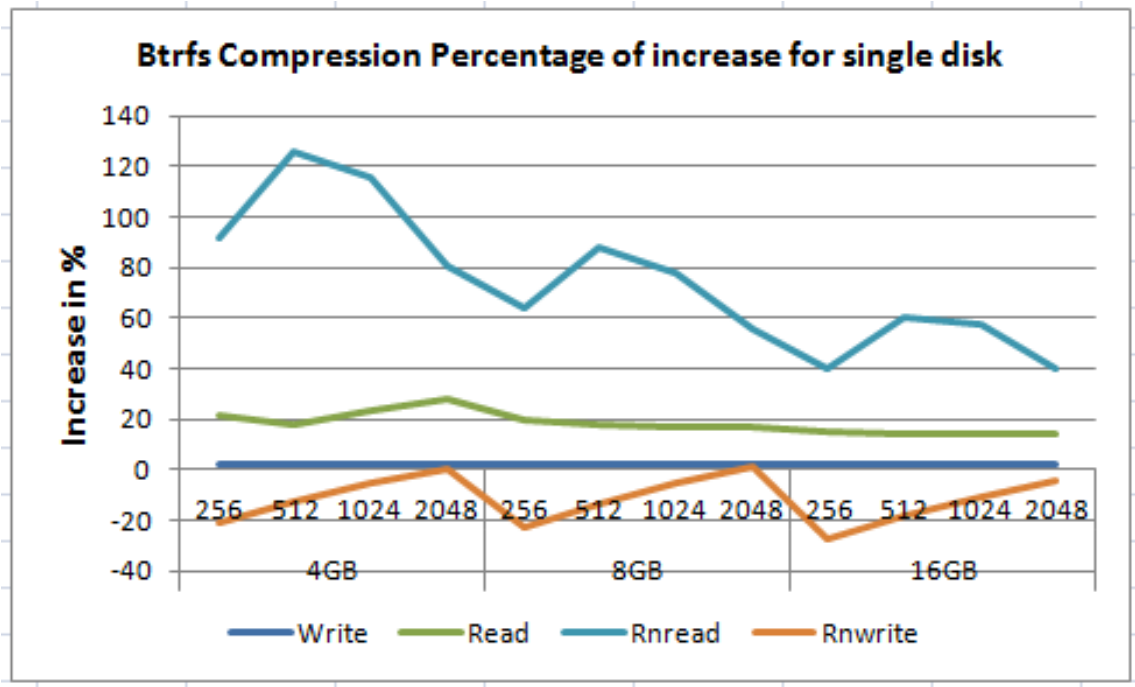


Figure 5.16: Btrfs single average throughput Percentage of increase due to compression

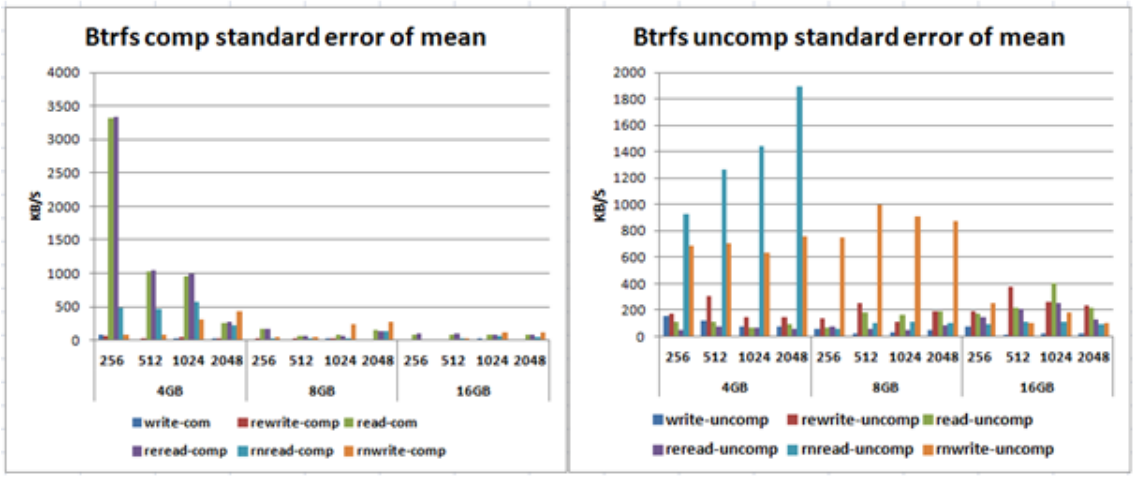


Figure 5.17: Btrfs single disk comparison standard error of mean

Test	**Avg Throughput with SEM Increase**	**Disk usage Increase by IOPS and %utili**	**Cpu usage Increase by SYS, IOWAIT, IDLE**
Sequential Write	2%, -50%	-81%, -36%	980%, -91%, -75%
Sequential Read	20%, -25% except for 4GB filesize	-85%, -54%	484%, -98%, -67%
RandomRead	74%, -69%	-92%, -12%	265%, -19%, -9%
RandomWrite	-10%, -67%	88%, -9%	576%, -52%, -58%

Table 5.2: Significant Performance Differences for Btrfs Single compression feature

5.2.1 Btrfs Sequential Write Operations Comparison

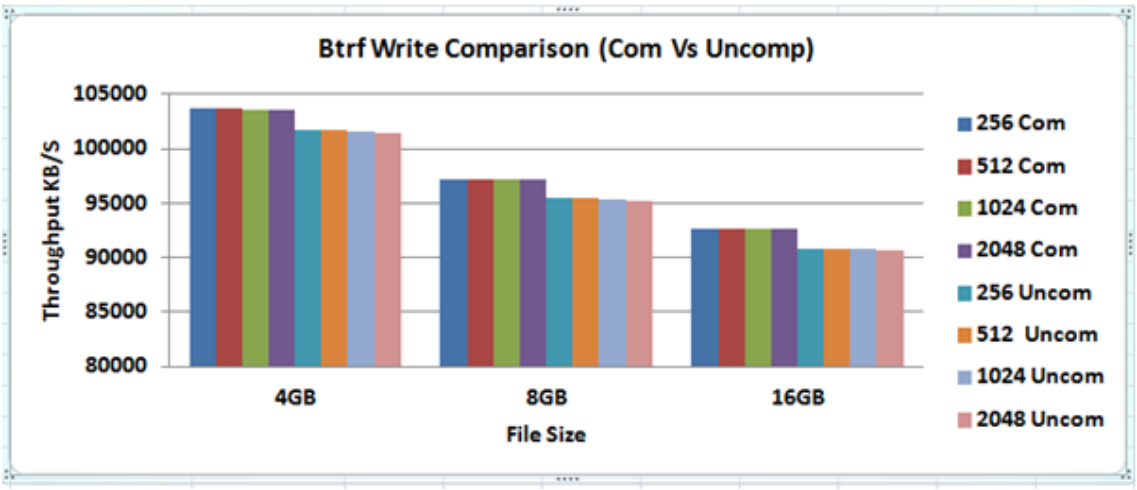


Figure 5.18: Btrfs single disk Write Avg Throughput comparison

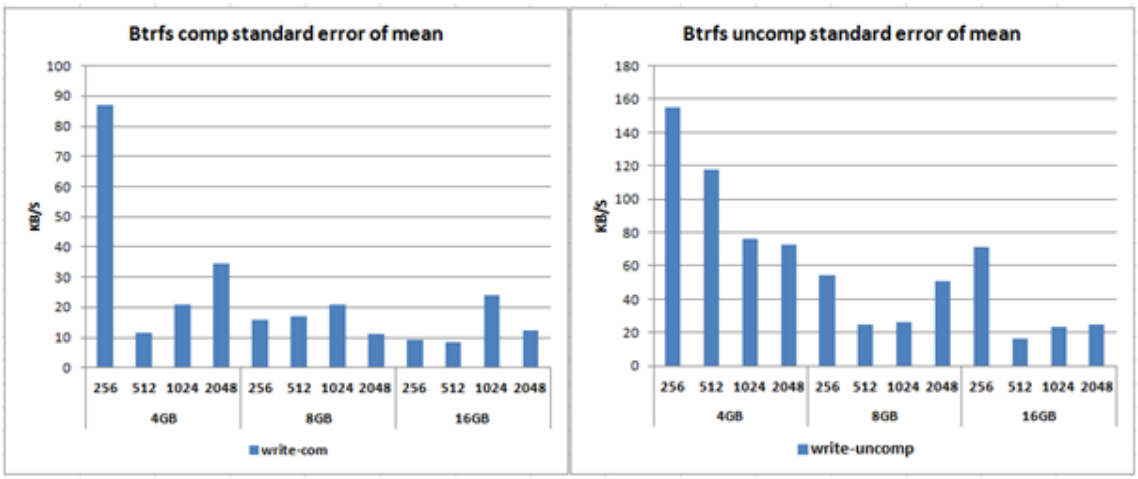


Figure 5.19: Btrfs single disk Write Throughput standard error of mean comparison

As can be seen the figures 5.14, 5.15, 5.16, 5.18, 5.19 and finally the table 5.2 in the analysis section show that sequentially writing to a file in a single disk with compression enabled provides a minimal performance improvement for the mentioned files sizes and record sizes.

For both compression and default the attained throughput decreases with the increment of file sizes and record sizes. When compared compression provides minimal improvement and it is constant for all records and files.

The difference is calculated and it is 2% better performance than the default option with standard error of mean ratio around half times or 50% lower

for compression.

The other noticeable difference observed is disk and CPU usage out put characterstics. As mentioned numerically in table 5.2 above, compression affected to have lower write IOPS which has decreased almost by 85% and hence the disk bandwidth utilization has too decreased by 54%. The only negative impact observed is that the CPU time usage has increased almost 980%.

5.2.2 Btrfs Sequential Read Operations Comparison

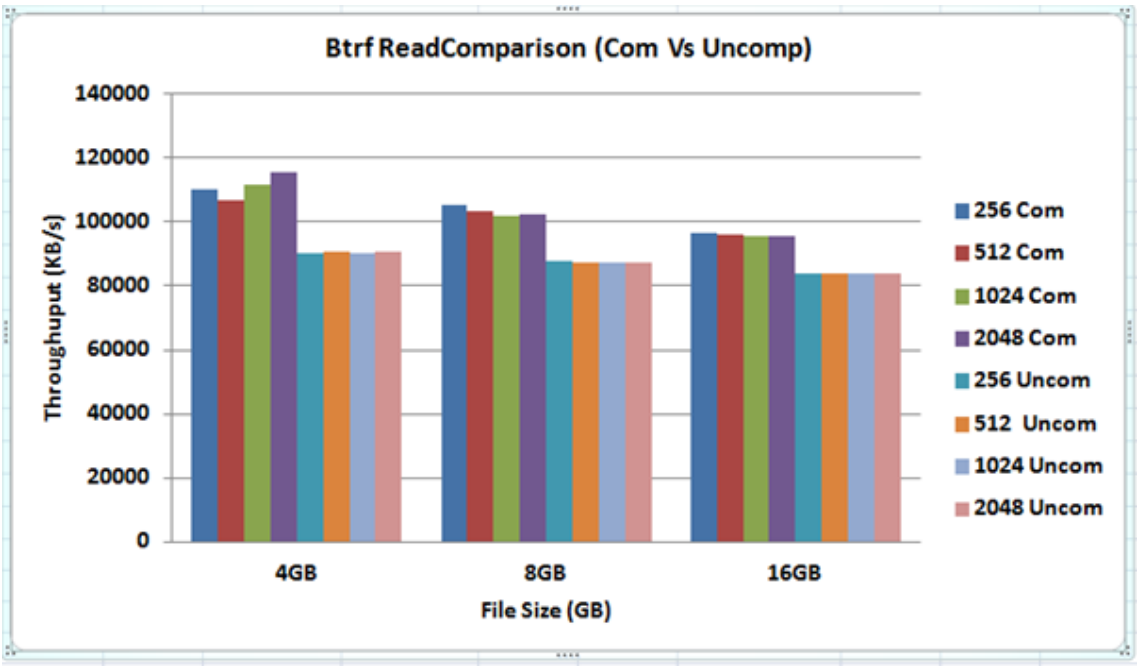


Figure 5.20: Btrfs single disk Read Avg Throughput comparison

As can be seen from the The graphs in figures 5.14, 5.15, 5.16, 5.20, 5.21 and the table 5.2, the performance of Btrfs with its compression feature starts to outshine the performance of Btrfs with the default options.

The behavior of the difference could be further investigated by file and record level. Hence for compression feature the throughput decreases with the increase of file and record except for file size of 4gb where the throughput increases with increase of file and record. Where as for default the through put relatively remain the same for record sizes and decreases significantly as file size increases.

Therefore the compression provides performance improvement that decreases with file size in effect. The difference is calculated and it is 20% better performance than the default option with standard error of mean ratio around

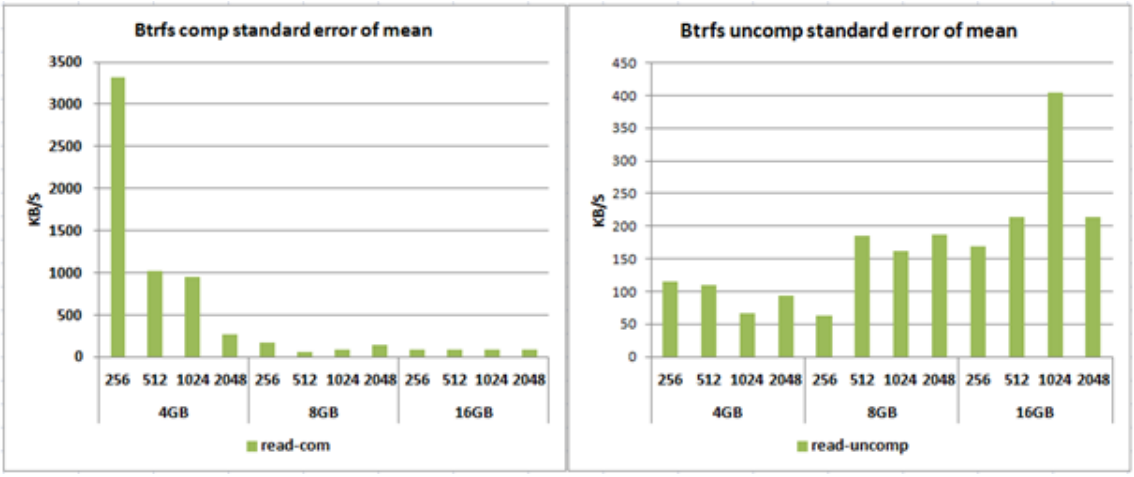


Figure 5.21: Btrfs Read Throughput standard error of mean comparison

25% lower for compression except for 4GB file which is high.

The other noticeable difference observed is disc and CPU usage out put characteristics. As mentioned numerically in table 5.2 above, compression affected to have lower write IOPS which has decreased almost by 81% and hence the disk bandwidth utilization has too decreased by 36%. The only negative impact observed is that the CPU time usage has increased almost 484%.

5.2.3 Btrfs Random Read Operations comparison

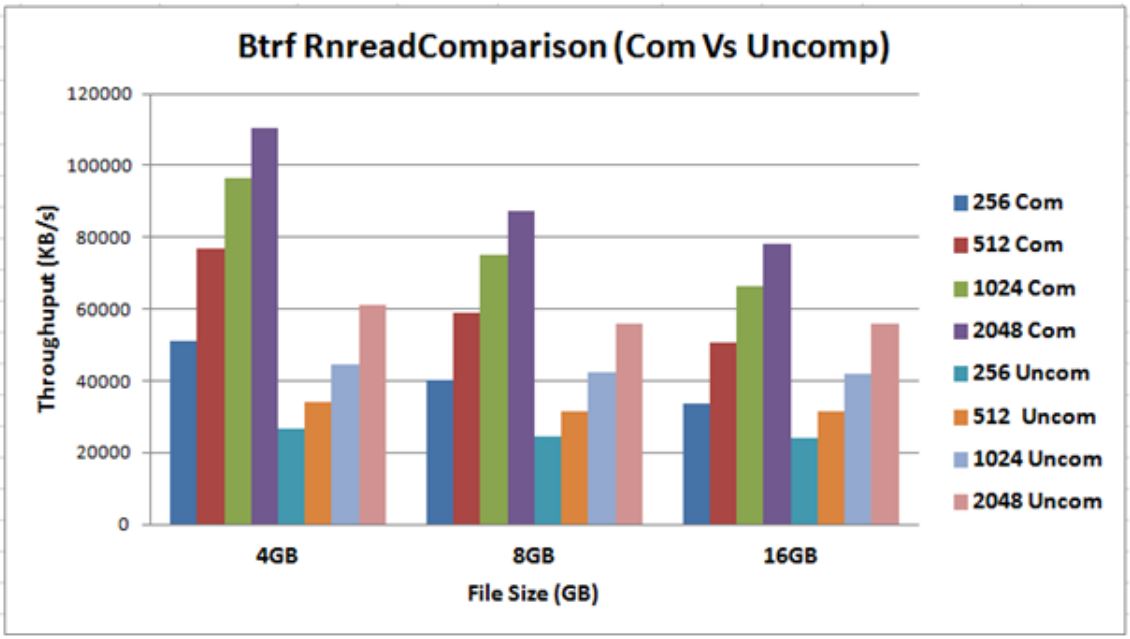


Figure 5.22: Btrfs Random Read Avg Throughput comparison

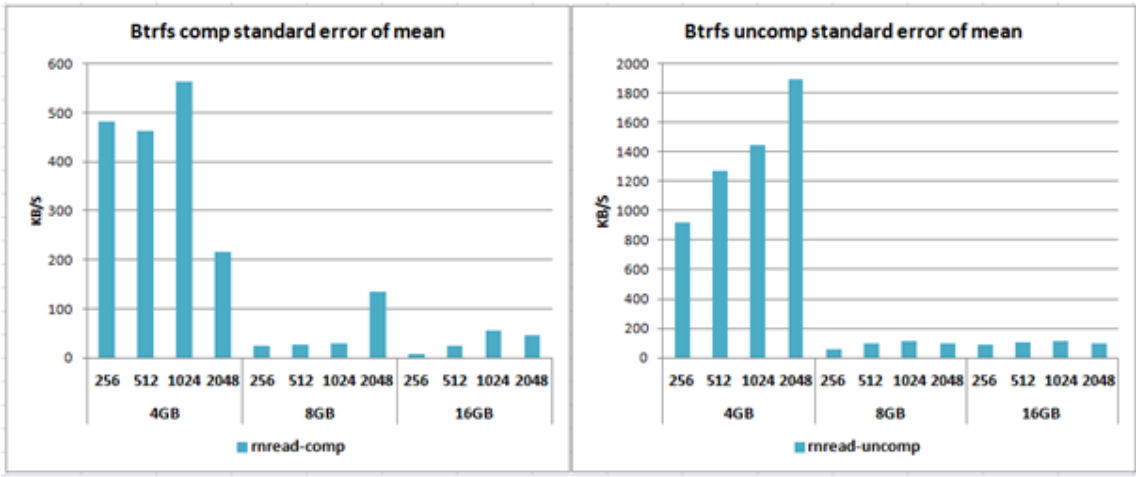


Figure 5.23: Btrfs Random Read Throughput standard error of mean comparison

As can be observed from the graphs in figures 5.14, 5.15, 5.16, 5.22, 5.23 and the table 5.2, the throughput achieved in randomly reading a file after enabling the compression feature does show better performance as compared to the default option.

The graph obviously shows throughput increase as record size increases where as decreases as file size increases.

Actually it is in this random reading case where the highest significant difference is obtained for Btrfs single set up. But compression provides performance improvement as compared to default where the comparison remains the same for records and decreases as file size increases.

The difference is calculated and it is 74% better performance than the default option with standard error of mean 59% lower for compression.

The other noticeable difference observed is disk and CPU usage out put characteristics. As mentioned numerically in table 5.2 above, compression affected to have lower write IOPS which has decreased almost by 92% and hence the disk bandwidth utilization has too decreased by 12%.

The only negative impact observed is that the CPU time usage has increased almost 265%.

5.2.4 Btrfs Random Write Operations Comparison

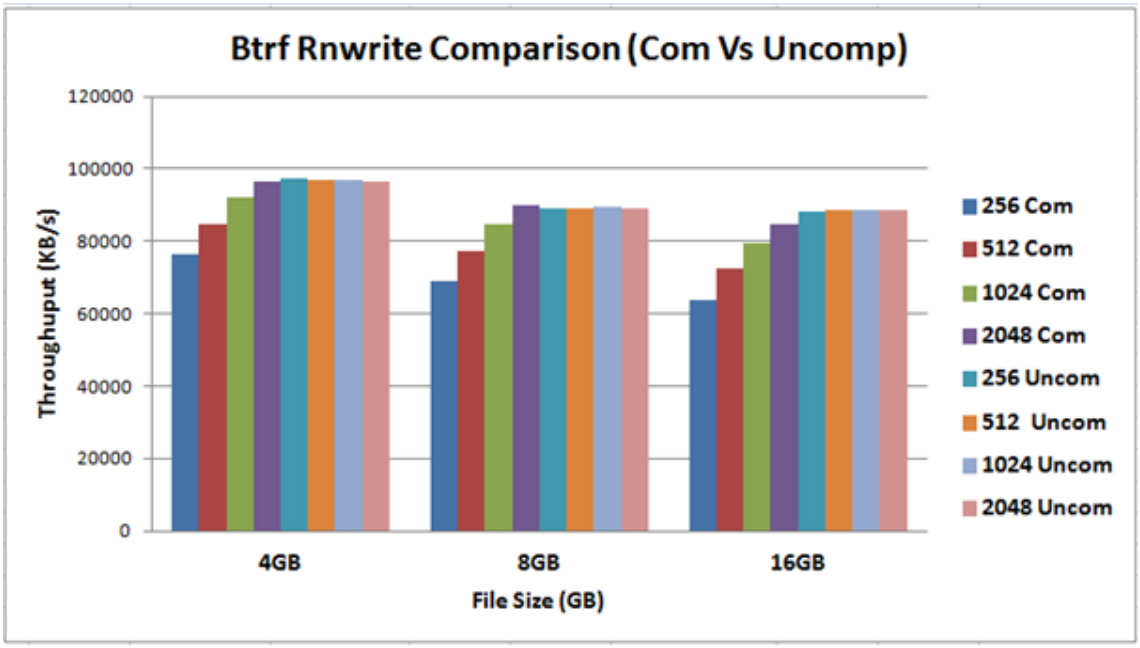


Figure 5.24: Btrfs Random Write Avg Throughput comparison

As it can be seen from the graphs in figures 5.14, 5.15, 5.16, 5.24, 5.25 and the table 5.2 for the first time in this benchmark testing process Btrfs without the compression feature enabled shows better throughput for all record sizes and file sizes.

The performance without compression feature is significant for small record sizes. For larger record sizes the difference is getting lower as is proved by the

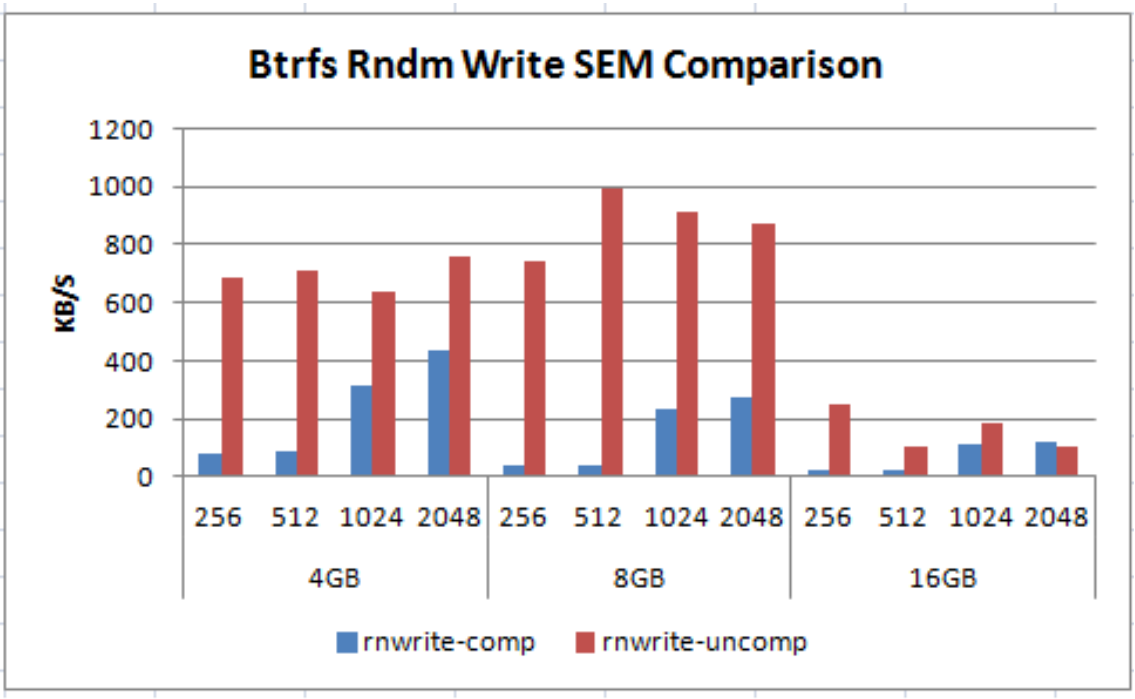


Figure 5.25: Btrfs Random Write Throughput standard error of mean comparison

graph in figure 5.24.

The average throughput difference is calculated and it is 10% lower performance than the default option with standard error of mean difference around 67% lower for compression.

The other noticeable difference observed is disk and CPU usage out put characterstics. As mentioned numerically in table 5.2 above, compression affected to have lower write IOPS which has decreased almost by 88% and hence the disk bandwidth utilization has too decreased by 9%. At the same time CPU time usage has increased almost 576%.

5.3 Multi-thread VS Zfs Single compression resource utilization

The experiment is done for one file size and one record size changing the number of threads running from one to five.
For the results presented in this section, results are limited to one record sizes: 2048KB, one file size: 8GB file size and one to five processes used.

5.3.1 CPU time Comparison

CPU time in contrast to wall time measures only the time during which the processor is actively working on a certain task.
As it can be seen in the graph in Figure 5.26 below the I/O operations for compressed feature has taken more CPU time to finish than with default feature except for random read and write.
The CPU time significantly increases with the increases of the process number. The CPU time difference b/n compressed and uncompressed feature is significant for read operaton. The rest difference is not significant.

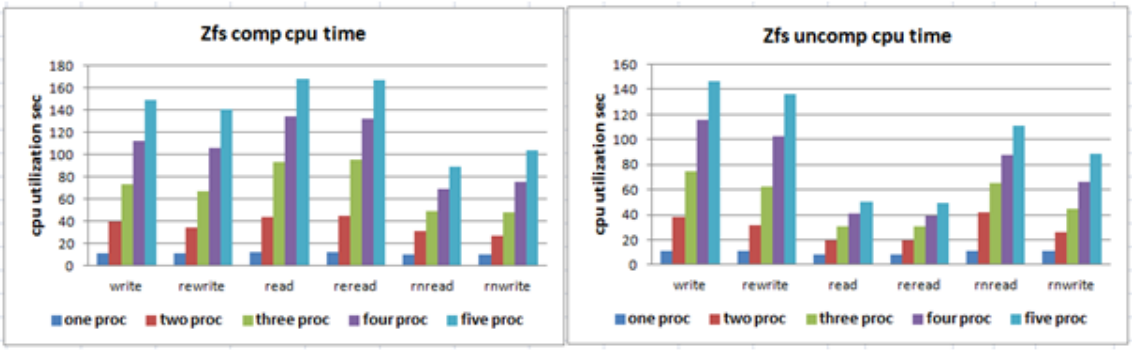


Figure 5.26: CPU time comparison

5.3.2 CPU utilization Comparison

CPU utilization being the percent of CPU cycles spent on a process, figure 5.27 below show clearly that compression feature has actually taken more CPU cycles to compress and decompress I/O channel data than the default feature. Hence compression features takes advantage of more CPU resources. One can experience latency due high CPU utilization in this case.

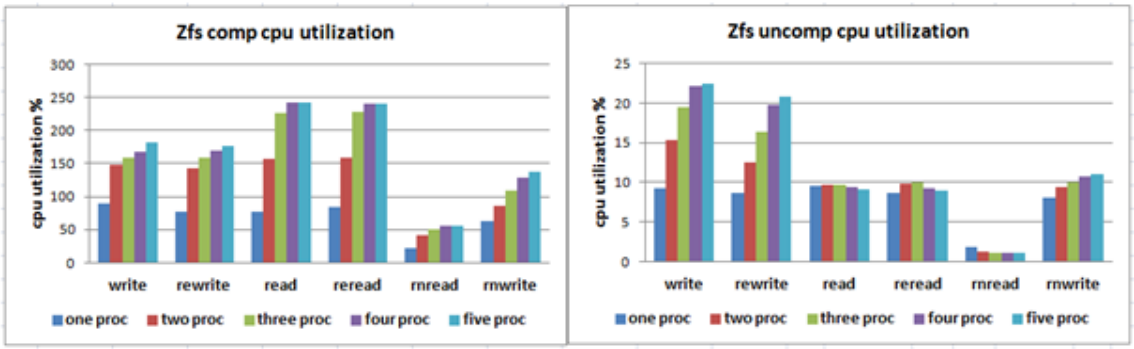


Figure 5.27: CPU utilization comparison

5.4 Monitoring disk and CPU usage by Zfs Single Compression feature

The following sections present the iostat output graphs that are obtained from the block layer and CPU utilization done when performance test is taken for 6GB file size and 128KB record size. The iostat output were taken simultaneously with the Iozone tests.

5.4.1 IOPS comparison

The results shown in Figure 5.28 displays the read and write I/O per second taken during the whole run time of performance benchmark for 6GB file and 128KB recordsize.
There is significant throughput difference in the two cases. The effect of compressing the data in to smaller size has affected the size of write and read operation in the range of ten times reduction for compression feature.

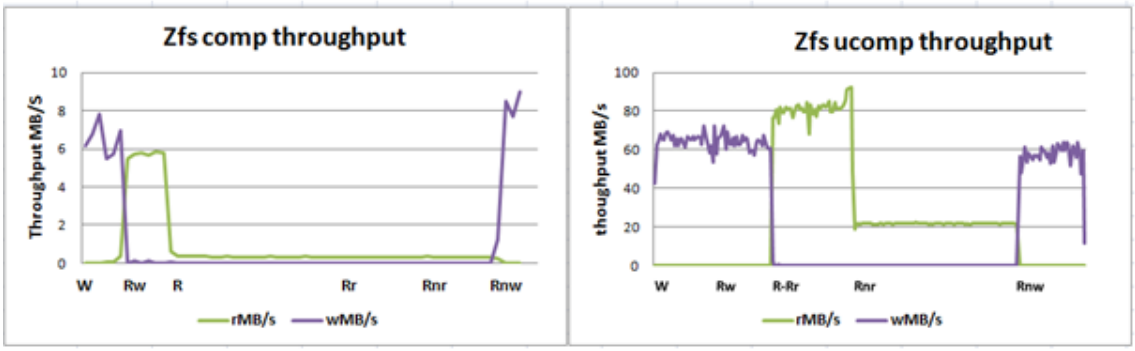


Figure 5.28: Zfs IOPS comparison

5.4.2 Bandwidth utilization comparison

Figure 5.29 illustrates that Zfs exhibits a much lower bandwidth utilization when applied compression feature and this is how compression relieves disk bandwidth saturation by sending smaller size i/o channel data by compressing data.

The graph shows Percentage of CPU time during which I/O requests were issued to the device (bandwidth utilization for the device). Device saturation occurs when this value is close to 100

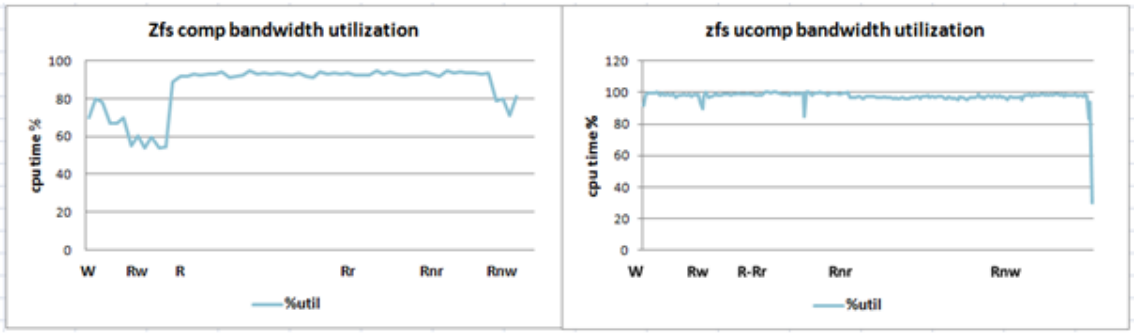


Figure 5.29: Zfs bandwidth utilization comparison

5.4.3 CPU usage comparison

The graph in figure 5.30 depicts the behavior of the cpu usage pattern during the performance benchmarking test taken.
%system show the percentage of CPU utilization that occurred while executing at the system level (kernel).
%iowait show the percentage of time that the CPU or CPUs were idle during which the system had an outstanding disk I/O request.
%idle show the percentage of time that the CPU or CPUs were idle and the system did not have an outstanding disk I/O request.
The graph reveals that the CPU was busy of compressing file during the writing mode(sequential and random) and it was less idle or there was less iowait cycles.
Where as in the sequential reading the compression feature is relatively uses lesser CPU than default. But random reading was different from the other that the compression feature uses higher CPU cycle.

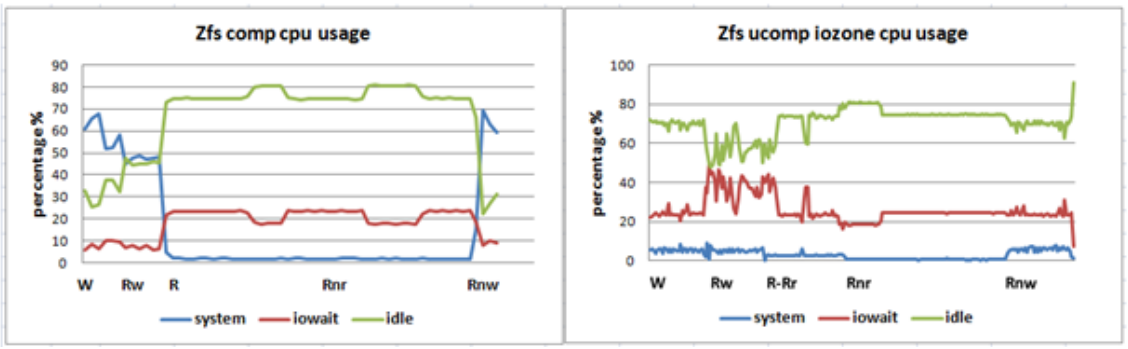


Figure 5.30: CPU Utilization Comparison

5.5 Multi-threading Vs Btrfs Single compression feature

5.5.1 Cpu utilization Comparison

CPU utilization being the percent of CPU cycles spent on a process, figure 5.31 below shows clearly that compression feature has actually taken more CPU cycles to compress and decompress I/O channel data than the default feature. Hence compression features takes advantage of more CPU resources. One can experience latency due high CPU utilization in this case.

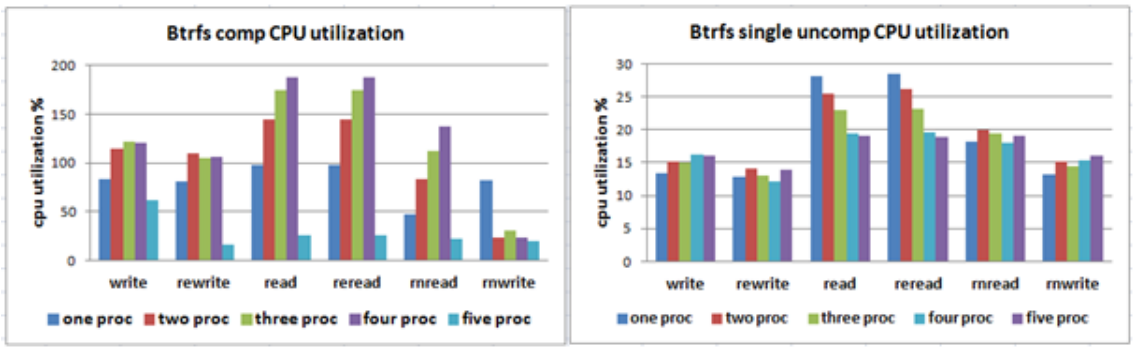


Figure 5.31: cpu utilization Comparison(Comp Vs Ucomp)

5.6 Monitoring disk and cpu usage by Btrfs Single Compression feature

The following sections present the iostat output graphs that are obtained from the block layer and cpu utilization done when performance test is taken for 6GB file size and 128KB record size.
The iostat output were taken simultaneously with the Iozone tests.

5.6.1 Write and Read IOPS comparison

The results shown in Figure 5.32 displays the read and write throughput per second taken during the whole run time of performance benchmark for 6GB file and 128KB recordsize.

There is significant throughput difference in the two cases. The effect of compressing the data in to smaller size has affected the size of write and read operation in the range of ten times reduction for compression feature.
In both cases the random read has lower value operation.

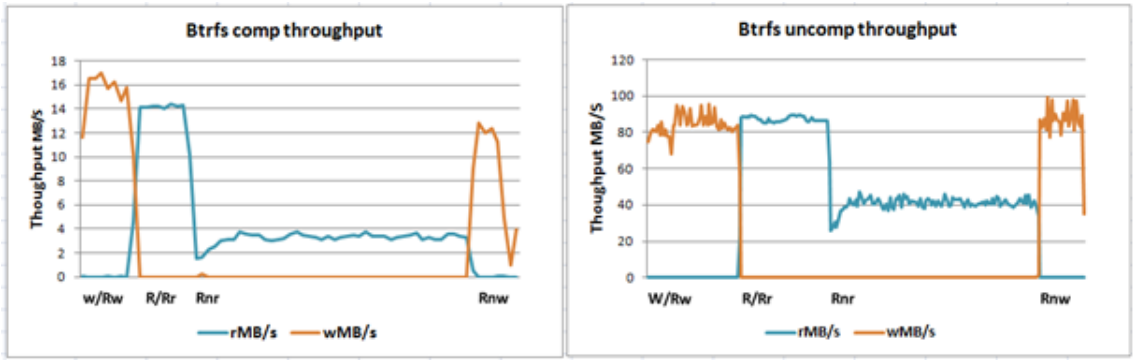


Figure 5.32: Instant throughput Comparison(Comp Vs Ucomp)

5.6.2 Bandwidth utilization comparison

Figure 5.33 illustrates that Btrfs exhibits a much lower bandwidth utilization when applied compression feature and this is how compression relieves disk bandwidth saturation by sending smaller size i/o channel data by compressing data.

The graph shows Percentage of CPU time during which I/O requests were issued to the device (bandwidth utilization for the device).

Device saturation occurs when this value is close to 100%. In addition the graph shows that while reading and rereading bandwidth utilization is significantly reduced due compression feature and random writing has not changed much due to the impact of compression.

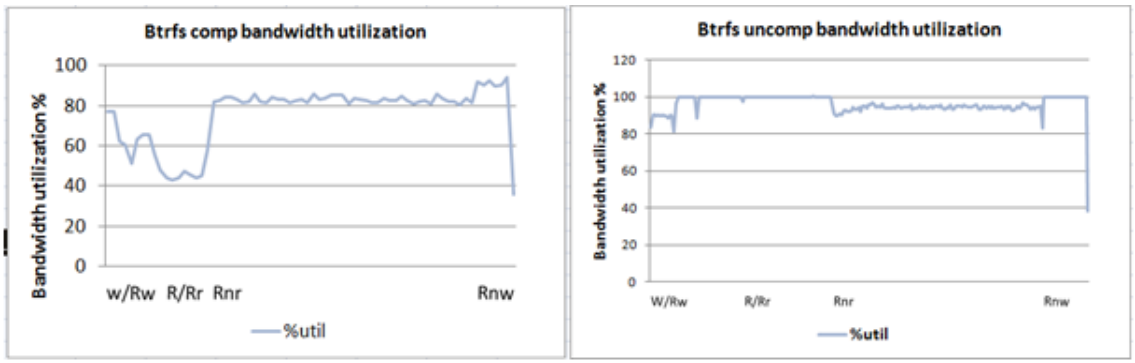


Figure 5.33: Bandwidth utilization Comparison(Comp Vs Ucomp)

5.6.3 CPU usage comparison

The graph in figure 5.34 depicts the behavior of the CPU usage pattern during the performance benchmarking test taken.
The graph reveals that the CPU was busy of compressing file during the writing mode (both sequential and random) and sequential reading so it was less idle or there was less iowait cycles.
But random reading was different from the other that both compression feature and default feature shows the same pattern.

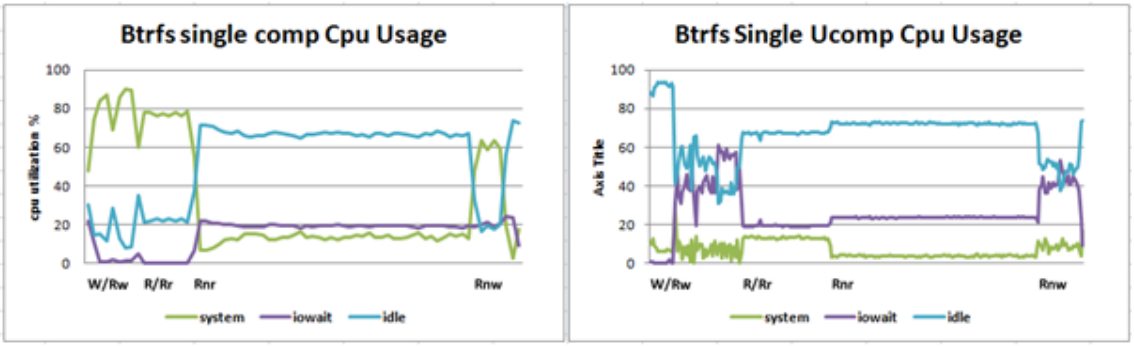


Figure 5.34: Cpu Utilization Comparison (Comp Vs Ucomp)

5.7 Performance Analysis for Zfs raidz1

This section presents the analysis obtained from sequential read/write and random read/write tests of Iozone. The results are the average throughputs and and their standard error mean.
For the results presented in this section, results are limited to four record sizes: 256KB, 512KB, 1024KB and 128KB.
The graphs below show both sequential and random operations performance of a Raidz1 zfs disk for uncompressed and compressed features.
The findings are further summarised in the table

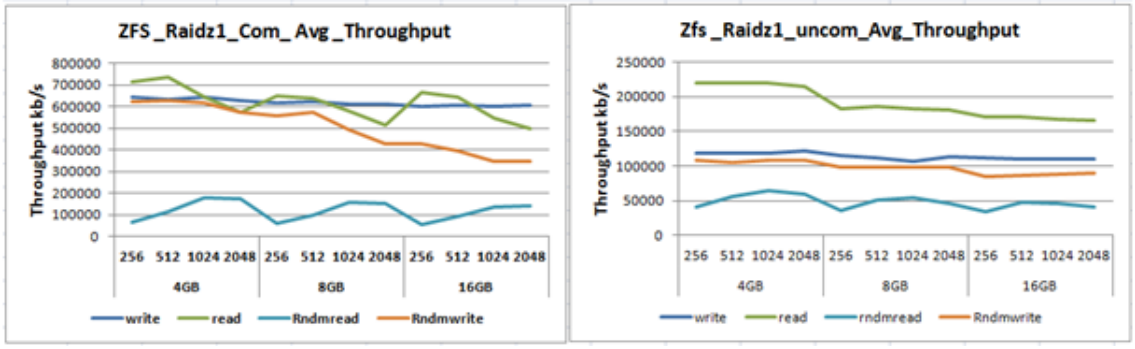


Figure 5.35: Average Throughput for Zfs Raidz1

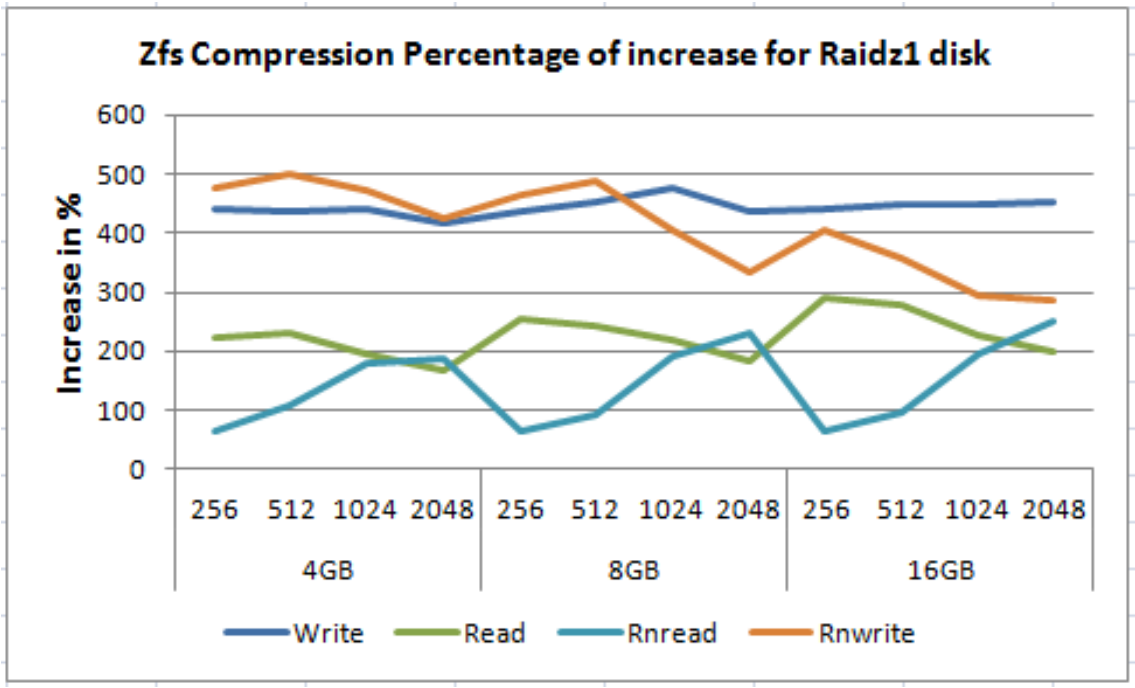


Figure 5.36: Zfs Raidz1 average throughput percentage of increase due to compression

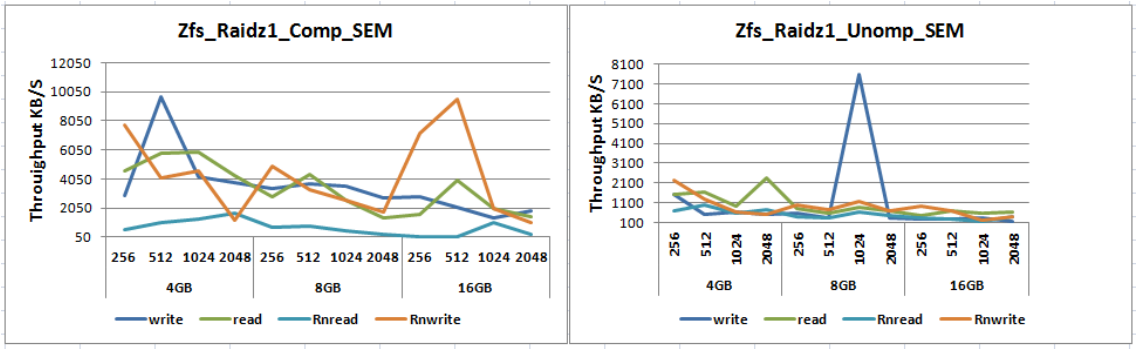


Figure 5.37: Standard error of mean comparison for Zfs Raidz1

Test	**Avg Throughput Increase**
Sequential Write	450%
Sequential Read	200%, decrease with record and file
RandomRead	144%, increase with file and record
RandomWrite	400%, Decrease with record and file

Table 5.3: Summary of Zfs Compression Comparison for Raid disks

5.8 Performance test Analysis for Btrfs Raid5

This section presents the analysis obtained from sequential read/write and random read/write tests of Iozone. The results are the average throughputs and and their standard error mean.
For the results presented in this section, results are limited to four record sizes: 256KB, 512KB, 1024KB and 128KB.
The graphs below show both sequential and random operations performance of a Raid5 Btrfs disk for uncompressed and compressed features.
Further the findings of the comparison are summarised in the table.

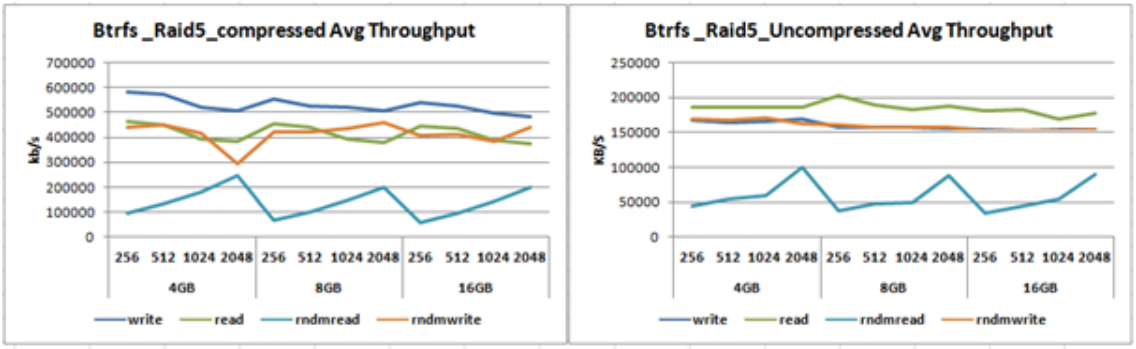


Figure 5.38: Average Throughput for Btrfs Raid5

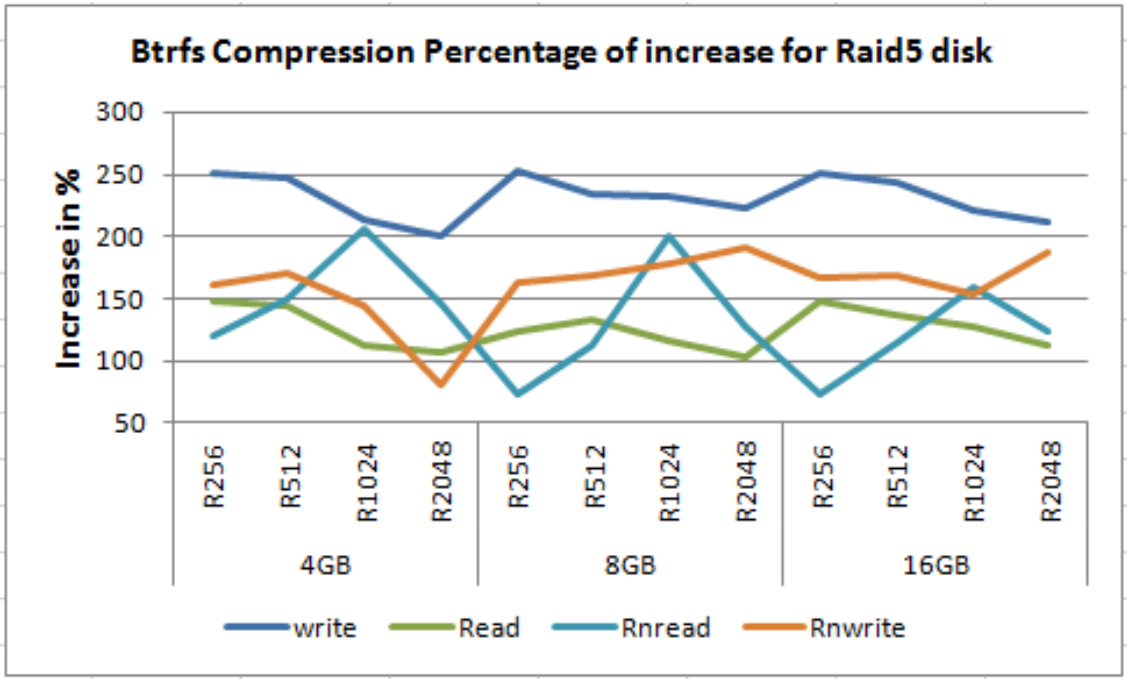


Figure 5.39: Average Throughput percentage of increase for Btrfs Raid5 due to compression

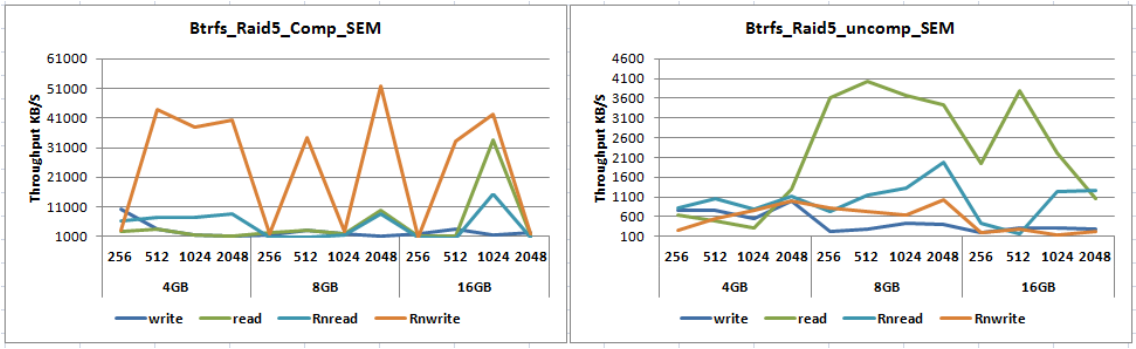


Figure 5.40: Standard error of mean Comparison for Btrfs Raid5

Test	**Avg Throughput Increase**
Sequential Write	250% - 200%, decrease with record, increase with file
Sequential Read	100% - 150%, decrease with record
RandomRead	70% - 200%, decrease with file pattern for record
RandomWrite	150% - 200%, constant with file

Table 5.4: Summary of Btrfs Compression Comparison for Raid disks

5.9 Zfs Compression Against Btrfs Compression Comparison

Both the figures and the tables show the analysis of the comparison of the impact of compression on the average throughput of Zfs when compared with the impact on Btrfs.

5.9.1 Zfs Compression Against Btrfs Compression Comparison for Single Disk

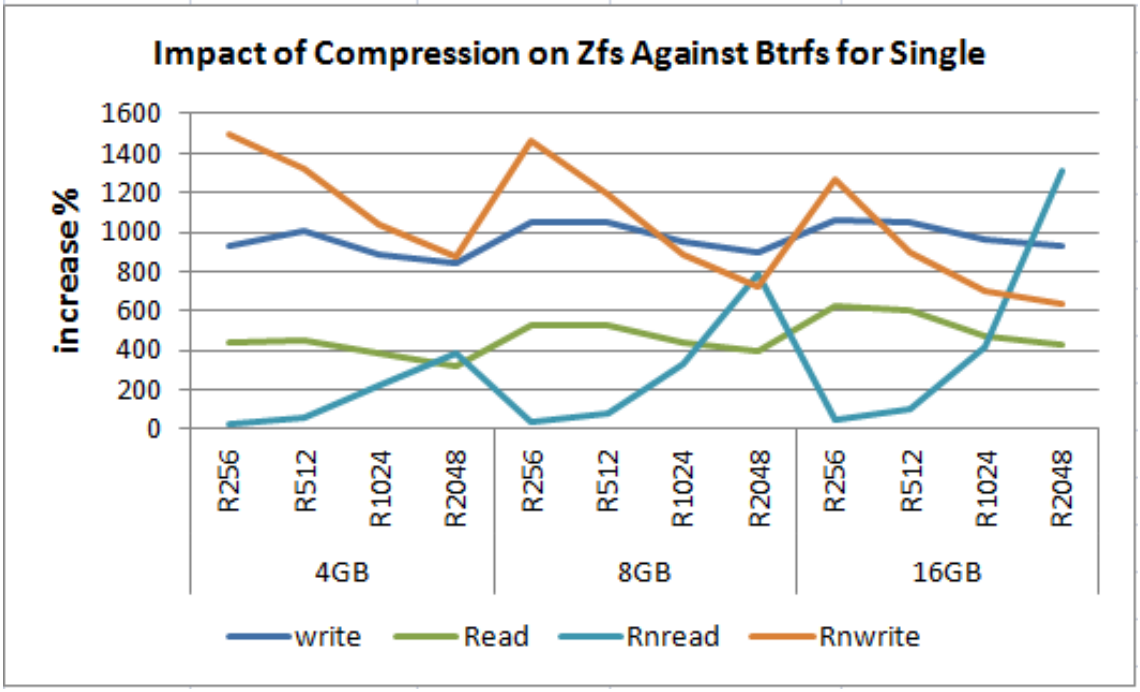


Figure 5.41: Impact of compression on Zfs against Btrfs for Single Disk

Test	**AVG THROUGHPUT Increase**	**Disk usage Increase by IOPS and %UTILITI**	**Cpu usage Increase by SYS, IOWAIT, IDLE**
Sequential Write	900%	-55%, 10%	-4%, 284%, 90%
Sequential Read	450%	-55%, 23%	193%, 1370%, 93%
RandomRead	170%, 290%, 460% per File and 34%, 79%, 312%, 790% per Record	-84%, 8%	-39%, 17%, 9%
RandomWrite	1000% per File and 1000%, 1100%, 880%, 750% per Record	47%, -14%	86%, -24%, -11%

Table 5.5: Summary of impact of compression on Zfs against Btrfs for Single Disk

5.9.2 Impact of Compression on Zfs against Btrfs for Raid Disk

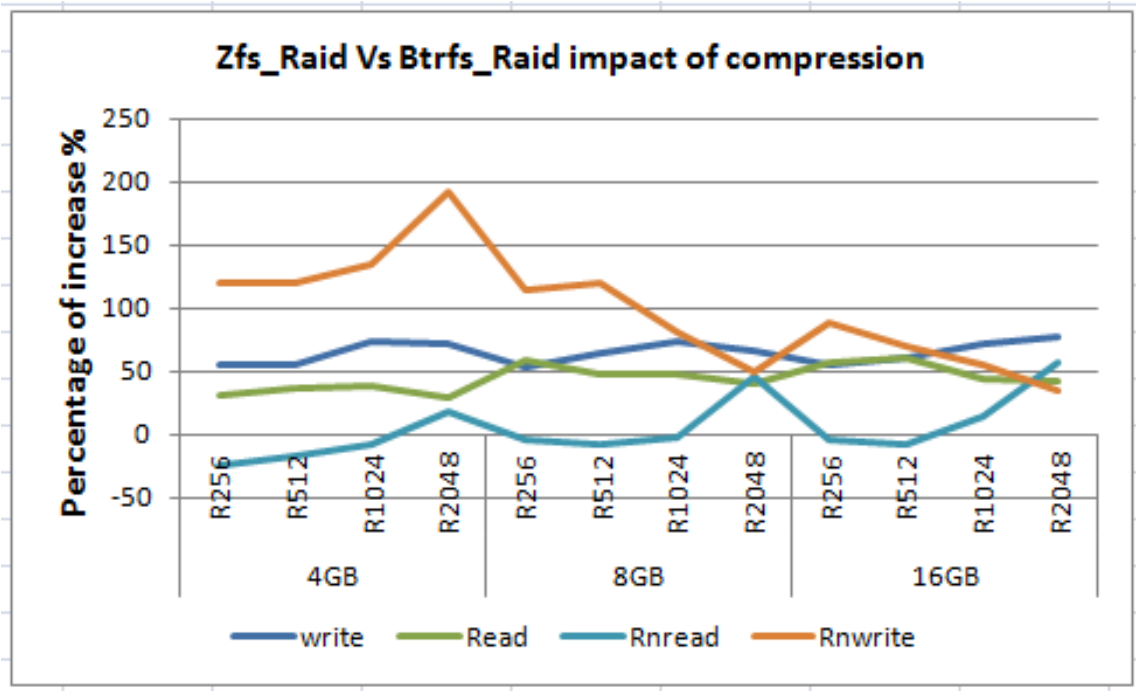


Figure 5.42: Impact of compression on Zfs against Btrfs for Raid Disk

Test	Avg Throughput Increase
Sequential Write	55%
Sequential Read	50%
RandomRead	-30% - 50%, Increase with file and record
RandomWrite	30% - 190%, Decrease with file and record

Table 5.6: Summary of impact of compression on Zfs against Btrfs for Raid Disk

5.10 Performance benchmarking test Analysis for Ext4

This section presents the analysis obtained from sequential read/write and random read/write tests of Iozone. The analysis includes both the single and raid disk set up offcourse for the uncompression feaure only as EXT4 doesnot include transparent compression so far.
For the results presented in this section, results are limited to four record sizes: 256KB, 512KB, 1024KB and 128KB.
The graphs below show the average for both sequential and random operations performance of a Ext4 single and Raid5 disk.

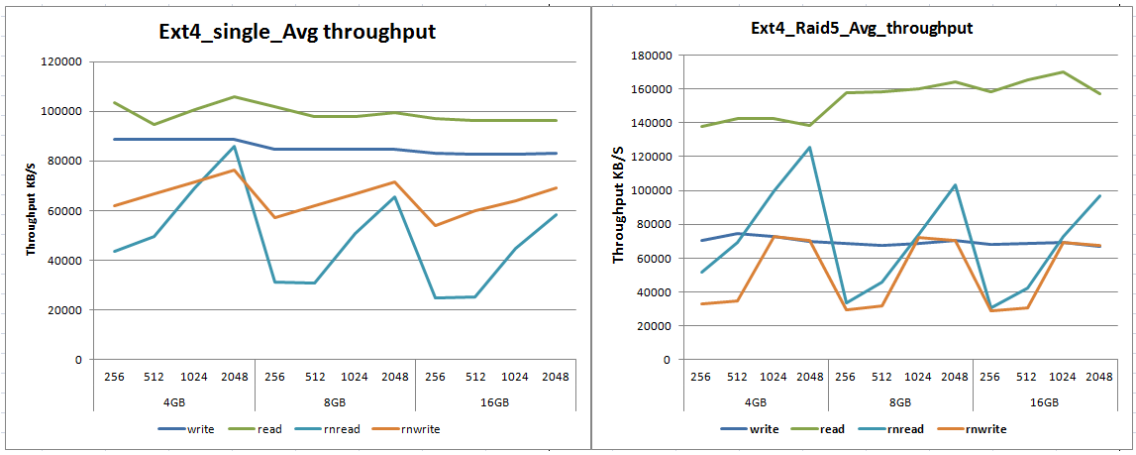


Figure 5.43: Average Throughput Comparison for Btrfs Ext4

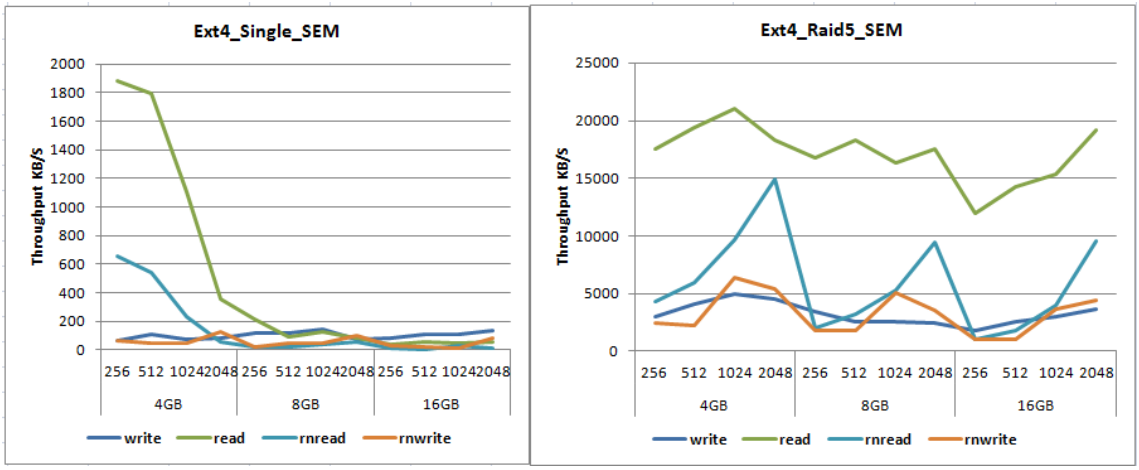


Figure 5.44: Standard error of mean Comparison for Btrfs Ext4

5.11 DD Command File Write and Read Test Results

The test was performed writing to and reading from all filesystems.
The Linux DD command was used to benchmark 6GB file size and 128KB record size.
Then the report of the elapsed time, system time and user time were taken to perform the read and write operations were taken.
dd if=/dev/zero of=speetest2 bs=128k count=46875 conv=fdatasync
dd if=speetest2 of=/dev/null bs=128k

As can be seen in the graph in figure 4.45 below, compressed Zfs performs better for writing and reading a file.

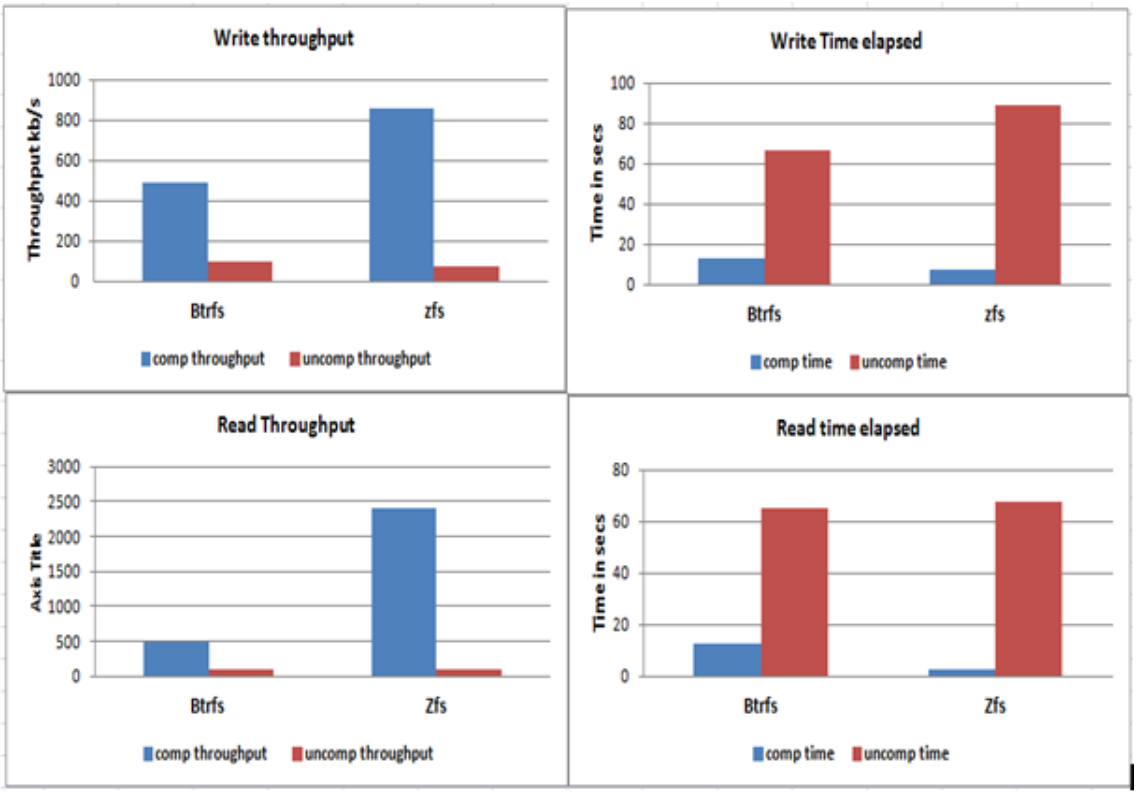


Figure 5.45: Average Throughput and Time elapsed Comparison for Btrfs Zfs

5.12 Linux kernel compile Test Analysis

The following figure 4.46 shows the result of analysis obtained from Linux kernel 3.14 compile test comparison for the two file systems.
Even though there is no significant time difference in minutes measured when taken in to account the total half hour time it took to compile the kernel the opposite is recoded for Btrfs and Zfs.

While Btrfs compressed has taken longer time, Zfs Compressed has taken lesser time compared with the respective default feature.

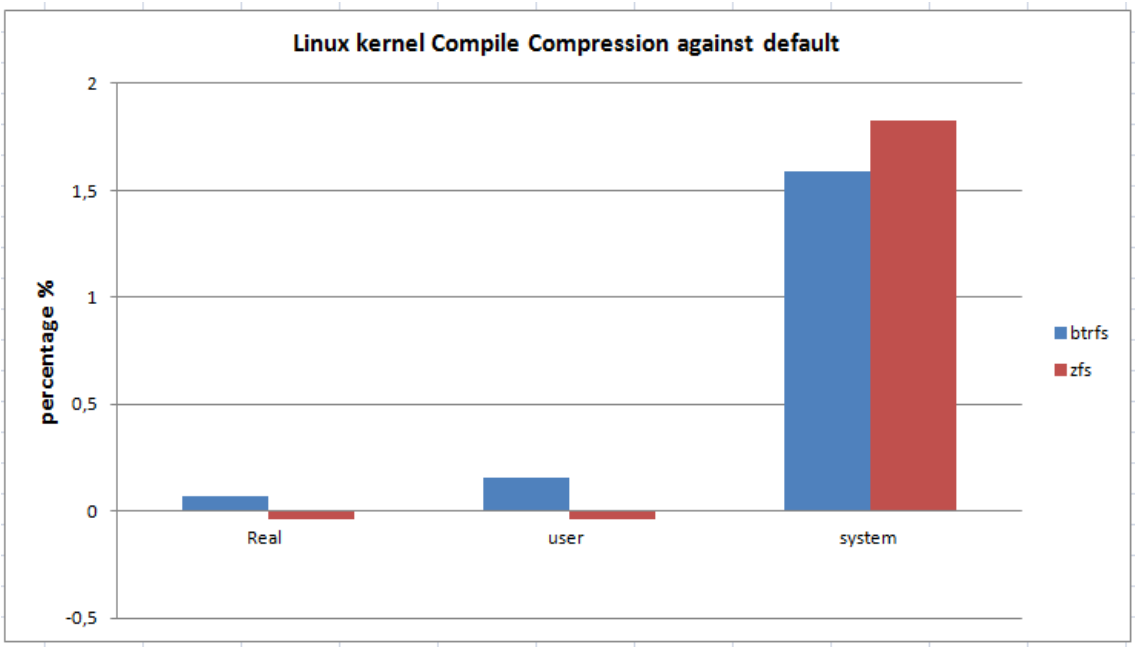


Figure 5.46: percentage increase in time to compile for Btrfs and Zfs

Chapter 6

Discussion

Transparent Compression in Zfs and Btrfs have impact on performance. It is relatively new feature in file system that would give performance boost by utilizing the abundantly available cpu and memory power of todays servers.

More over Zfs and Btrfs are also modern file system which are designed to be object-based than the old fashioned block-based file systems. Due to their design and architecture they are very flexible for management and to add additional new features which goes with the demand of the day. Zfsonlinux is relatively new file system included to linux that will highly boost the need for linux in the market as Zfs has so many attractive benefits. So studying this filesystem will help to investigate those advantages that will be used by linux users in the future. Both file systems are copy on write and transactional file systems that fulfill security and safety in data handling.

The compression feature specially will have a great potential that will be utilized in todays data intensive society as the study could pin point out which work loads in the form of file size and record size as well as for which data operations in the form of either sequential or random operations are affected by this new feature.

In this chapter we will discuss the overall project process from the beginning till the end in a summarized manner. In general we will go through the following key points:
. Challenges and limitations
. Thesis summary
. Retrospective study of approach
. Unexpected results
. Reproducibility, Repeatability, and predictability

i. Challenges and limitations

Limiting the scope of the comparative study was the main challenge. In the beginning the intention was to compare three file systems with and with out

compression for a number of record and file sizes and for single disk and raid configuration. This make the study too much for analysis and even to compare each other will be difficult. Hence formulating the problem statement and designing the approach was affected by the process. Comparative or investigative study of performance of file system in general is a complex task. Detailed study of the variable involved will affect the quality of the study. It is possible to compare too many variables with out deep analysis but obscured and important points will be missed.

ii Thesis summary

The Introduction chapter of this thesis provides the following important information:
.The motivation of the author to do this research and why it is important.
. File system overview.
. The type of problem to be tackled and why it is a problem.
. How the problem is going to be solved and approached.
. The goal and contribution of this research work.

In the background and related work chapter (chapter 2), it is discussed what file systems are all about, their evolution, architecture and feature. Benchmarking a file system and IOzone benchmarking tool are also discussed in chapter2.

The approach and methodology chapter explains first how the problem is going to be solved and approached and then the methodology and the best experimental set up for the suggested approach. The resources used and the lab topology are also explained. Perl scripts used are discussed. Even though Iozone has many options, write/rewrite, read/reread, random read and random write speeds are measured in all benchmarking tests. Four record sizes (256, 512, 1024, 2048) and three file sizes (4GB, 8GB, 16GB) are also used in all cases. The experiment setup was divided into the following main types. First the setup is divided according to the three filesystem. Then the setup has single and raid disk setup which will give us six setup. Next is test of performance with compression and and with out compression which will make the number of setup ten. Then it was intended to study performance by scaling up file size and record size which will make the total number of variables to be studied at this point to be 20 different variables. Finally in total for the above ten setup 30 file sizes and 40 record sizes were studied. For these 30 files and and 40 records 4 write and read operations were taken and becomes 160 record operations and 120 file operations. Totally from the above 10 setup 280 write and read operations were investigated individually. Which makes the result collection and analysis to make and difficult. This is only for iozone file and record operation test. But studying the impact of compression on record and file size is not enough. The impact of compression on throughput as the number of thread increases were necessary. So for one file and one record size when the number of thread scales from one to five were studied. Which means 10 through put commands were run and data was collected. Then it

was necessary to test and see the characterstics of disk IOPs and CPU usage for one file and one record size for four Btrfs and Zfs single compressed and default set up. That is additional four commands where run to collect data with iostat for 4 iozone commands that makes the 8 commands. Additional 5 dd commands and 5 linux compile commands were run to see additional impact of compression other than iozone one file copy and compiling time. So additional 10 commands where run. Finally in total 38 scripts were used to get result for the necessary tests mentioned above.

Let us talk about data collection procedure next. Mainly MS excel were used to collect the sample data for the two iozone commands, one for the record and file combination and the other for throughput. So MS excel was used for the data from the 20 scripts. But for the 4 iostat scripts grep command were used extensively to take cpu and disk data separetly. Next to manuplate them individually Ms excel were used. Still for the file copy dd command and linux compile command MS excel was used.

Last but not least of the approach and setup chapter is how the disk and file system configuration looks likes. First I have 5 benchmarking disks of 73GB size. The first step was to prepare the disks for single disk setup. Then all the three filesystem were to be configured first with out compression and next to umount the file system and do it with compression and all the tests had to be taken. Then the disks had to be configured for raid set up and the above tests had to be repeated again.

In chapter 4 we get the results of the iozone(record and file), iozone throughput, iostat, dd and linux compile experiment types that showed scalablity test of compression for file size, record size and process numbers as well as the impact of compressing file on the patten of disk and cpu usage. Ms Excel line and bar graphs and R boxplots are used to show the results. All three types of graphs used are important when we see the type of information we get from each of them. They are all explained in the same chapter. Except there were few outliers, the overall result clearly shows that enough samples were taken to keep the normality of raw data distribution. It was not possible to take more than six samples as the size of file size is big enough to have smooth data samples. The out put of each iteration by it self was average out puts that the raw datas were smples of means. Each iterations take anumber of hours that will prove the normality of the raw data distribution.

The analysis chapter deals with the best way of analyzing the raw data so as to get the best possible meaning for the main question of the theis experiment. There for taking the average and the standard error of mean of the average was the best meaning full way of presenting the solution. In this way the average data will be compared as well as the distribution characterstics will be obtained from the standard error of mean which is the result of the standard deviation divided by the square root of the number of samples. This will best describe the distribution specially when the sample is smaller. So that as ex-

plained above the focus of the analysis section was to get comparative average values and their distribution so as to compare the performance boost or degradation for each experiment set up. For the the file and record scalablity set up alone we have four different set up to compare each other. Namely :
. zfs single disk compressed and uncompressed
. zfs raid disk compressed and uncompressed
. Btrfs single disk compressed and uncompressed
. Btrfs raid disk compressed and uncompressed
Then next comparison was the impact of compression on
. Zfs single against Btrfs single
. Zfs raid against Btrfs raid
For the single disk set up the disk and cpu usage characterstics were important. There for from the disk usge the following variables were choosen
. Disk IOPS in MB/s
. Disk bandwidth utilization
. usage (System, iowait and idle time)
For process scalablity and impact of compression test the following variables were taken
. CPU time processor time for the task
. CPU utilization - percent of cpu cycle spent on the task
Large File Copy and linux Compile test were done in addition to using iozone as additional tests as alternative test approach and the following variables were studied
. Large file writing throughput
. Large File reading throughput
. Large file writing time
. Large file reading time
. Compiling Time and Compression

The main findings from the analysis of the data look likes the following few points summarized. The details of the findings are summarized in different tables in the analysis section. It is so detailed that it is difficult to raise every result here. But the main findings will be worth mentioning and discussing.

The first very high impact of compression is displayed for Zfs single disk setup. The second high impact is exhibited for Zfs raid setup. The third high impact is for Btrfs raid set up but the last impact is for Btrfs single disk set up.

To discuss roughly through each points, it was obtained that the single zfs set up has got 900% sequential write performance increase while the sequential read has got 600% high. Compression has provided on average 500% for random read and 900% for random write operations for the single Zfs set up. While mentioning the second high impact on zfs raid set up it was achieved 400% increase for sequential write and 200% increase for sequential read higher throughput where as the improvement for the random operation is 140% increase for random read and 400% times for random write.

The next significant difference is obtained from raid disk set for Btrfs. The sequential write has displayed from 250% - 200%, which decreases with record, and increases with file. The sequential read has displayed from 100% - 150%, that decreases with record. Random read has exhibited 70% - 200%, that decreases with file pattern for record. Lastly the random write improved by 150% - 200%, constant with file.

The least impact of compression is found for single disk Btrfs set up. Operation wise the result look likes the following when ranked by improvement percentage. Random read 74%, sequential read 20%, sequential write 2% and random write has -10% which is the only negative impact.

The other point to discuss is the comparison of compression impact according to the following variables.
. Read/write operations
. Record size
. File size
. Single/raid disk

Accordingly it was the sequential write that scored the highest improvement. The least and the negative impact of compression is on random write Btrfs single set up which is 10% lesser than the uncompressed throughput. The 512 record and 4GB file size has displayed the best throughput improvement so far.

In the above discussion the focus was on the average throughput advantage obtained by utilizing compression feature on different Zfs and Btrfs setups. The other finding in this thesis is the distribustion of the data which affects directly the average throughput behavior discussed above. This is directly related with SEM that is standard error of mean. Compression has negative impact on all set up except Btrfs single with regard to standard error of mean that is SEM is high. On average the SEM of Compressed data is 300% times higher than uncompressed data.

The other negative impact of compression is on the use of CPU time and cycle. Compression uses high CPU power especially when the number of processes increase, where as compression has positive impact on disk bandwidth utilization in all cases.

Ext4 file system was taken in this experiment as base line in the benchmarking of the modern file systems Zfs and Btrfs. The revolutionary design difference that is seen in Zfs and Btrfs which does not exist in ext4 has actually affected many other aspect of the filesystems features except one thing, that is Ext4 still has exhibited better performance in all read write operations in this experiment setup where the default setup is followed with out further tuning to get optimized performance in the single disk set up.

In case of raid set up Ext4 has shown lesser performance than the two modern

file system where Btrfs is seen to better in this case.

iii Retrospective study of approach

Benchmarking compression feature of Zfs and Btrfs by the suggested approach using iozone and the other tools has given results that support the theory that compression has more or less positive impact on performance at least for some work loads. But using synthetic and micro benchmarking tools could not reveal the actual real life experience of daily life. It could have been done by the alternative real benchmarking approach where data could have been collected from real working environments like big companies who have different kind of filesystem work loads. The problem with this approach is that it is difficult to get this kind of permission and the nature of the duration of the thesis approach. In addition, iozone script data has demonstrated that it might be highly compressible data that the results have been a little exaggerated for zfs setup.

iv Unexpected results

Transparent compression on the fly in Zfs and Btrfs compresses at high speed and the compression ratio depends on the compressibility of the data under the test. In this thesis it is not the compression ratio that is directly studied but indirectly the impact of compressing data on performance. The data type supplied by iozone benchmarking tool is found to be very compressible and might have influenced the result of the experiment. But from the theory Lzo compresses better than Lz4 where as Lz4 compresses faster than Lzo. Here the impact of compression speed is proved to be better than compression ratio. Speed matters in performance that is why Lz4 has greater impact on Zfs than Lzo has on Btrfs.

v Reproducibility, Repeatability, and predictability

Concerining the setup, installation and configuration of this thesis didnt create very significant challenges rather it was straightforward. The results also show clear trend and meaningful graphs that the whole project could be reproduced. Since the trends of the results found are consistent and show clear line of graph except one or two few outliers (which are excluded from the average calculations), its predictability is high. The projects repeatability is also high. If many more repeated tests had been taken under same condition, similar result would have been found as margins of standard errors and confidence interval values are very less. Hence, the results were repeatable.

Chapter 7

Conclussion

7.1 Summary of main findings

The overall performance analysis made for transparent compression feature in Zfs and Btrfs shows positive differences in the performance results obtained with the synthetic benchmarking tool and real world application tests. The results obtained from the Iozone benchmarking tool show that the Zfs compression provides better performance for all read/write operations in general. Similarly, Btrfs compression provides higher performance with most read/write operations with random write operations being an exception for the single disk setup only.

The results of iostat disk and cpu usage monitoring show that Compression provides Better performance by utilizing the cpu power available 1000% and hence decreasing disk bandwith utilization by average 30% for the single disk setup. This will reduce for raid set up as long as raid is another solution for io performance improvement.

7.2 Evaluation and FutureWork

The study in this thesis carried out as many possible tests of filesystem compression feature functionality as feasible given the timeframe of the work. The following observations can be made about ways that this study could be strengthened even more.

- The Iozone benchmaking tool is good in providing an overall assessment of filesystem performance. However, the data type provided by iozone is proved to be highly compressible and the result is exaggerated at least for Zfs and LZ4. The result will be more comprehensive if different file types are tested, since this would help to identify the interactions of file types with compression algorithms
- Implementing tests with both macro and micro benchmarking tools would be helpful in understanding the differences between the filesystems.

- The set up is done relatively for large files but if the set up can be done in the future for small file sizes the result can show different scenarios.

Bibliography

[1] Lars Wirzenius, Joanna Oja, Stephen Stafford, and Alex Weeks. The linux system administrators guide (1993-2004).

[2] Online. http://en.wikipedia.org/wiki/Moore%27s_law. [Accessed 20-January-2014].

[3] Online. http://wiki.illumos.org/display/illumos/ZFS. [Accessed January 22,2014].

[4] Valerie Aurora. https://lwn.net/Articles/342892/. [Accessed January 23,2014].

[5] Nathan Willis. Weekend project: Get started with btrfs, october 15,2010. http://www.linux.com/learn/tutorials/371623-weekend-project-get-started-with-btrfs. [AccessedJanuary 23,2014].

[6] Suresh M. Ext4 file system- feature and setups. http://www.bobcares.com/blog/ext4-file-system-features-and-setup/. [AccessedJanuary 23,2014].

[7] National Institute of Standards and Technology. B-tree. http://www.xlinux.nist.gov/dads/HTML/btree.html, 2007. [AccessedFebruary1,2014].

[8] M. Tim Jones. Anatomy of the linux file system. http://www.ibm.com/developerworks/linux/library/l-linux-filesystem/, October 30,2007. [Accessed February1,2014].

[9] Robert Love. Linux kernel development, third edition. http://www.ibm.com/developerworks/linux/library/l-linux-filesystem/, Jun 22, 2010.

[10] Paul Krzyzanowski. File system design case studies. http://www.cs.rutgers.edu/~pxk/416/notes/13-fs-studies.html, March 23, 2012.

[11] Greg Gange Abraham Silberschatz, Peter Baer Galvin. Operating systems concepts,seventh edition. http://it325blog.files.wordpress.com/2012/09/operating-system-concepts-7-th-edition.pdf, 2004.

[12] Ben Rockwell. Understanding zfs compression. http://www.cuddletech.com/blog/pivot/entry.php?id=983, Nov 06 2008.

[13] Zfsonlinux.org. Faq-zfs on linux. http://zfsonlinux.org/faq.html, 2013.

[14] Jeff Bonwick. The slab allocator: An object-caching kernel memory allocator. http://zfsonlinux.org/faq.html, 2004.

[15] Matt Ahrens Val Henson and Jeff Bonwicki. Automatic performance tuning in the zettabyte file system. http://3c2controller.net/project/truetrue/solaris10/henson-self-tune.pdf, 2003.

[16] Jeff Bonwick. Zfs: The last word in filesystems. https://blogs.oracle.com/bonwick/entry/zfs_the_last_word_infilesystems, october 31, 2005.

[17] Jeff Bonwick. Raid-z. https://blogs.oracle.com/bonwick/entry/raid_z, Nov 17, 2005.

[18] David A. Patterson, Garth Gibson, and Randy H. Katz. A case for redundant arrays of inexpensive disks (raid). http://doi.acm.org/10.1145/50202.50214, 1988.

[19] Sun. Zfs on-disk specification. , 2006.

[20] Jeff Bonwick. Zfs block allocation. https://blogs.oracle.com/bonwick/en/entry/zfs_block_allocation, Nov 04, 2006.

[21] Benr. Understanding zfs: Compression. http://www.cuddletech.com/blog/pivot/entry.php?id=983, Nov 06,2008.

[22] Amanda Amcpherson. A conversion with chris mason on btrfs. http://www.linuxfoundation.org/news-media/blogs/browse/2009/06/conversation-chris-mason-btrfs-next-generation-file-system-linux, June 22,2009.

[23] Josef Bacik. Btrfs swiss army knife of storage. https://c59951.ssl.cf2.rackcdn.com/4376-bacik_0.pdf, February, 2012.

[24] Chris Mason. Btrfs design. https://oss.oracle.com/projects/btrfs/dist/documentation/btrfs-ukuug.pdf, February, 2012.

[25] Btrfs Wiki. Compression-btrfs. https://www.google.no/?gfe_rd\=cr\&ei\=j3QsU_jKMaGK8QecuoCQBA\#q\=btrfs\+compression, Jun 25, 2013.

[26] Markus F.X.J. Oberhumer. Lzo. http://www.oberhumer.com/opensource/lzo/, August 12, 2011.

[27] Mark Adler. Zlib, April 28, 2013.

[28] Abraham Silberschatz Greg Gagne, Peter Baer Galvin and Jo Allan Beran. Operating system concepts. http://www.cs.uic.edu/~jbell/CourseNotes/OperatingSystems/13_IOSystems.html, 2008.

[29] Carl Henrik Holth Lunde. Improving disk i/o performance on linux. http://heim.ifi.uio.no/paalh/students/CarlHenrikLunde.pdf, 2009.

[30] Khalid Sayood. Introduction to data compression. http://www.google.no/books?hl=no&lr=&id=Lhrge2YVpBwC&oi=fnd&pg=PP2&dq=Data+compression,the+process+of+encoding+digital+information+using+fewer+bits&ots=iKXetFlPJk&sig=j_NVT50eBZDdxCkd1soU0D7CQnU&redir_esc=y#v=onepage&q&f=false, 2012.

[31] Frank Ebner and Volker Schneide. Analysis of web data compression and its impact on traffic and energy consumption. https://www.bibliothek.fhws.de/static/PDFs/fhws-science-journal/Analysis_of_Web_Data_Compression_and_its_Impact_on_Traffic_and_Energy_Consumption.pdf, 2013.

[32] Yann Collet. Real time data compression. http://fastcompression.blogspot.no/p/lz4.html, 2014.

[33] Yann Collet. Extremely fast compression algorithm. https://code.google.com/p/lz4/, 2014.

[34] Silesian University of Technology. The data compression resource on the internet. http://www.data-compression.info/Corpora/SilesiaCorpus/, 2014.

[35] Don Capps and Tom McNeal. Iozone filesystem benchmark. http://www.iozone.org/, 2014.

[36] l Jan Kra SUSE Labs. Ext4, btrfs, and the others. http://atrey.karlin.mff.cuni.cz/~jack/papers/lk2009-ext4-btrfs.pdf, 2009.

[37] Dominique A. Heger. Workload dependent performance evaluation of the btrfs and zfs filesystems. http://www.dhtusa.com/media/IOPerfCMG09.pdf, 2009.

[38] Meaza Tayer Kebede. Performance comparison of btrfs and ext4 filesystems. https://www.duo.uio.no/handle/10852/34150, 2012.

[39] Sakis Kasampalis. Copy on write based file systems performance analysis and implementation. http://faif.objectis.net/download-copy-on-write-based-file-systems, 2010.

[40] Avishay Traeger, Erez Zadok, Nikolai Joukov, and Charles P Wright. A nine year study of file system and storage benchmarking. *ACM Transactions on Storage (TOS)*, 4(2):5, 2008.

[41] Wasim Ahmad Bhat and SMK Quadri. Benchmarking criteria for file system benchmarks. *International Journal of Engineering Science and Technology (IJEST)*, 3(1), 2011.

Appendix A

Scripts full overview

Listing A.1: BenchMarking Scripts

```
Iozone script
Btrfs
iozone -a -i 0 -i 1 -i 2 -r 256 -r 512 -r 1024 -r 2048
-s 4g -s 8g -s 16g -f /btrfstest/btrfs_single_compressed$i
iozone -a -i 0 -i 1 -i 2 -r 256 -r 512 -r 1024 -r 2048
-s 4g -s 8g -s 16g -f /btrfstest/btrfs_single_uncompressed$i
iozone -i 0 -i 1 -i 2 -+u -l 1 -u 5 -r 2048 -s 8g -F
/btrfs_single_uncomp/thrpt1 /btrfs_single_uncomp/thrpt2
/btrfs_single_uncomp/thrpt3 /btrfs_single_uncomp/thrpt4
/btrfs_single_uncomp/thrpt5

iozone -a -i 0 -i 1 -i 2 -r 256 -r 512 -r 1024 -r 2048
-r 4096 -s 4g -s 8g -s 16g -f /btrfsraiduncomp/uncompressed
iozone -a -i 0 -i 1 -i 2 -r 256 -r 512 -r 1024 -r 2048
-r 4096 -s 4g -s 8g -s 16g -f /btrfsraid5comp/compressed
iozone -i 0 -i 1 -i 2 -+u -l 1 -u 5 -r 2048 -s 8g -F
/btrfsraid5comp/thrpt1 /btrfsraid5comp/thrpt2
/btrfsraid5comp/thrpt3 /btrfsraid5comp/thrpt4
/btrfsraid5comp/thrpt5
iozone -i 0 -i 1 -i 2 -+u -l 1 -u 5 -r 2048 -s 8g -F
/btrfsraiduncomp/thrpt1 /btrfsraiduncomp/thrpt2
/btrfsraiduncomp/thrpt3 /btrfsraiduncomp/thrpt4
/btrfsraiduncomp/thrpt5
Zfs
iozone -a -i 0 -i 1 -i 2 -r 256 -r 512 -r 1024 -r 2048
-s 4g -s 8g -s 16g -f /tank/dataset1/soltest1/test3
iozone -a -i 0 -i 1 -i 2 -r 256 -r 512 -r 1024 -r 2048
-s 4g -s 8g -s 16g -f /tank/dataset2
iozone -i 0 -i 1 -i 2 -+u -l 1 -u 5 -r 2048 -s 8g -F
/tank/dataset1/thrpt1 /tank/dataset1/thrpt2
/tank/dataset1/thrpt3 /tank/dataset1/thrpt4
```

```
/tank/dataset1/thrpt5
iozone -i 0 -i 1 -i 2 -+u -l 1 -u 5 -r 2048 -s 8g -F
/tank/dataset2/thrpt1 /tank/dataset2/thrpt2
/tank/dataset2/thrpt3 /tank/dataset2/thrpt4
/tank/dataset2/thrpt5
iozone -a -i 0 -i 1 -i 2 -r 256 -r 512 -r 1024 -r 2048
-s 4g -s 8g -s 16g -f /tank1/datasetuncomp/uncompzaidz1
iozone -a -i 0 -i 1 -i 2 -r 128 -r 256 -r 512 -r 1024
-r 2048 -s 4g -s 8g -s 16g -f /tank1/datasetcompred/
zfs_raid_compfile
iozone -i 0 -i 1 -i 2 -+u -l 1 -u 5 -r 2048 -s 8g -F
/tank1/uncompred/thrpt1 /tank1/uncompred/thrpt2
/tank1/uncompred/thrpt3 /tank1/uncompred/thrpt4
/tank1/uncompred/thrpt5
iozone -i 0 -i 1 -i 2 -+u -l 1 -u 5 -r 2048 -s 8g -F
/tank1/datasetcompred/thrpt1 /tank1/datasetcompred/thrpt2
/tank1/datasetcompred/thrpt3 /tank1/datasetcompred/thrpt4
/tank1/datasetcompred/thrpt5
Ext4
iozone -a -i 0 -i 1 -i 2 -r 256 -r 512 -r 1024 -r 2048
-s 4g -s 8g -s 16g -f /dev/sdd
iozone -i 0 -i 1 -i 2 -+u -l 1 -u 5 -r 2048 -s 8g -F
/Ex4_singledisk_uncomp/thrpt1
/Ex4_singledisk_uncomp/thrpt2
/Ex4_singledisk_uncomp/thrpt3
/Ex4_singledisk_uncomp/thrpt4
/Ex4_singledisk_uncomp/thrpt5
iozone -a -i 0 -i 1 -i 2 -r 256 -r 512 -r 1024 -r 2048
-s 4g -s 8g -s 16g -f /Ext4_raid5/ext4raid
iozone -i 0 -i 1 -i 2 -+u -l 1 -u 5 -r 2048 -s 8g -F
/ext4_raid5/thrpt1 /ext4_raid5/thrpt2 /ext4_raid5/thrpt3
/ext4_raid5/thrpt4 /ext4_raid5/thrpt5

data collection commands

grep "Avg" Ext4_singledisk_ucomp_throughput
>>Ext4_singledisk_ucomp_throughputAvg
grep "CPU" Ext4_singledisk_ucomp_throughput
>>Ext4_singledisk_ucomp_throughputcpu
grep "Avg" Btrfs_raidz5_uncomp_throughput
>>Btrfs_raidz5_uncomp_throughputAvg
grep "CPU" Btrfs_raidz5_uncomp_throughput
>>Btrfs_raidz5_uncomp_throughputcpu
grep "Avg" Btrfs_raid5_comp_throughput
>>Btrfs_raid5_comp_throughputAvg
grep "CPU" Btrfs_raid5_comp_throughput
>>Btrfs_raid5_comp_throughputcpu
```

```
grep "Avg" single_zfs_disk_ucomp_throughput >>
single_zfs_disk_ucomp_throughputAvg
grep "CPU" single_zfs_disk_ucomp_throughput >>
single_zfs_disk_ucomp_throughputcpu
grep "Avg" single_zfs_disk_compr_throughput >>
single_zfs_disk_compr_throughputAvg
1599  grep "CPU" single_zfs_disk_compr_throughput >>
single_zfs_disk_compr_throughputcpu
grep "Avg" raidz1_zfs_ucomp_throughput >>
raidz1_zfs_ucomp_throughputAvg
grep "CPU" raidz1_zfs_ucomp_throughput >>
raidz1_zfs_ucomp_throughputcpu
grep "Avg" raidz1_zfs_comp_throughput >>
raidz1_zfs_comp_throughputAvg
grep "CPU" raidz1_zfs_comp_throughput >>
raidz1_zfs_comp_throughputcpu


File copy commands
wget  http://www.gutenberg.org/ebooks/2701
cp 2701-h.htm /tank/dataset1/mobby1
  cp 2701-h.htm /tank/dataset2/mobby2
  du -ah /tank/dataset1/mobby1
  du -ah /tank/dataset2/mobby2

time cp dickens.txt /tank/dataset1/dick.txt
 time cp dickens.txt /tank/dataset2/dick2.txt
 time cp dickens.txt dick3.txt
time cp dickens.txt /Btr_single_uncomp/dick4.txt
time cp dickens.txt /Btr_single_comp/dick5.txt


Linux kernel Compile Commands
wget https://www.kernel.org/pub/linux/kernel/v3.x/linux-
3.14.tar.xz
cd linux-3.11.0/
echo "Clean"; make clean >/dev/null 2>&1 ;
echo "Defconfig"; make defconfig >/dev/null 2>&1 ;
echo "Build";  time ( make  >/dev/null 2>&1 );


cat iostatBtr_single_compall.out | grep -A1 'avg-cpu' >>
iostatBtr_single_compall.out1
cat iostatBtr_single_compall.out1 | grep 0 >>
iostatBtr_single_compall.out-avg-cpu
less iostatdataset2all.out |grep sdc >> iostatdataset2IOPS
less iostatdataset1all.out |grep sdc >>iostatdataset1IOPS
```

```
cat iostatBtr_single_compall.out |grep sdg >>
Btrfs_single__comp_IOPS
cat iostatBtr_single_uncompall.out |grep sdg >>
Btrfs_single_ucomp_IOPS


DD commands
dd if=/dev/zero of=speetest3 bs=128k count=46875
conv=fdatasync
dd if=speetest3 of=/dev/null
```

Listing A.2: ConfigurationScripts

```
Fdisk  /dev/sdg
mkfs.btrfs /dev/sdg
mkdir btrfs_single_uncomp
mount /dev/sdg /btrfs_single_uncomp/
touch btrfs_single_uncompressed
./iozone_test.pl&

mkdir btrfs_single_comp
 mount -o compress=lzo /dev/sdg  /btrfs_single_comp/
mount-o compress=lzo /dev/sdg /btrfstestcompress
./iozone_test.pl&


 fdisk /dev/sdd
 fdisk /dev/sde
  fdisk /dev/sdf
  mkfs.btrfs -f -m raid5 -d raid5 /dev/sdd1 /dev/sde1 /dev/sdf1
  mkdir /btrfsraid5comp
  mount -o compress=lzo /dev/sdd1 /btrfsraid5comp/
mkdir /btrfs_raid5_compressed
touch btrfs_raid5_compfile
  ./iozone_test.pl&

fdisk /dev/sdd
mkdir /Ext4_singledisk_uncomp
mount /dev/sdd /Ext4_singledisk_uncomp
./iozone_test.pl&


mdadm --create --verbose /dev/md0 --level=5 --raid-devices=3
 /dev/sdd /dev/sde /dev/sdf
  mkfs.ext4 /dev/md0
  mkdir /ext4_raid5
   mount /dev/md0 /ext4_raid5/
./iozone_test.pl&
```

```
fdisk /dev/sdg
mkdir /Ex4_singledisk_uncomp
1533  mount /dev/sdg1 /Ex4_singledisk_uncomp/
 1534  mkfs.ext4 /dev/sdg1
 1535  mount /dev/sdg1 /Ex4_singledisk_uncomp/

Zpool create  f  tank /dev/sdc
zfs create tank1/datasetuncomp
 zfs create tank1/datasetcomp
zfs set compression=lz4 tank1/datasetcomp


zpool create tank1 raidz1 /dev/sdd1 /dev/sde1 /dev/sdf1
zfs create tank1/uncompred
zfs create -o compression=lz4 tank1/datasetcompred
./iozone_test.pl&
```

Appendix B

Supplementary graphs of benchmarking results

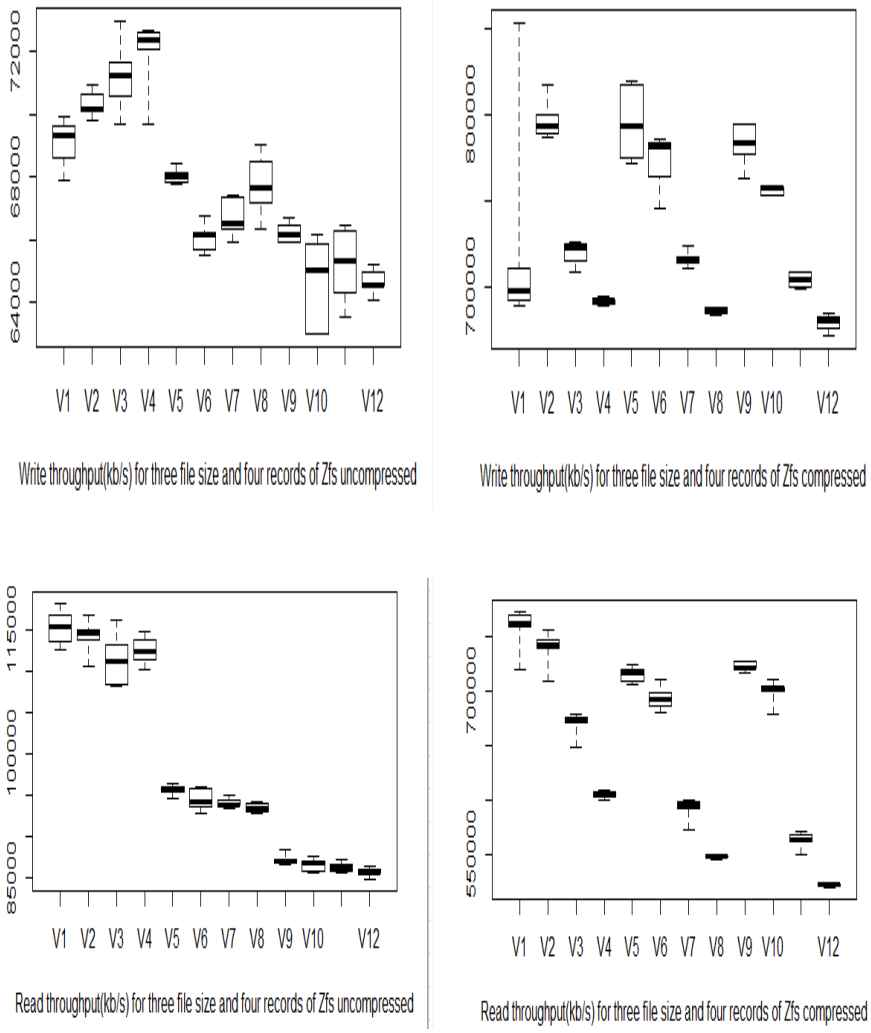
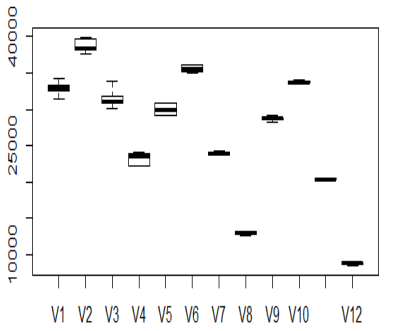
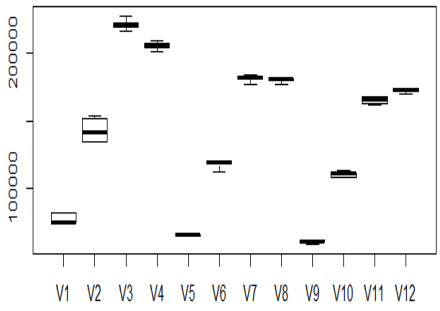


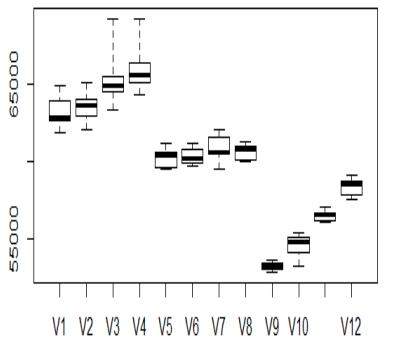
Figure B.1: Boxplot for Zfs single Write Read result



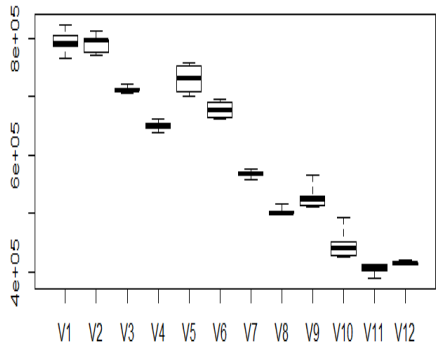
Rnread throughput(kb/s) for three file size and four records of Zfs uncompressed



Rnread throughput(kb/s) for three file size and four records of Zfs compressed



Rnwrite throughput(kb/s) for three file size and four records of Zfs uncompressed



RnWrite throughput(kb/s) for three file size and four records of Zfs compressed

Figure B.2: Boxplot for Zfs single Rnwrite Rnread result

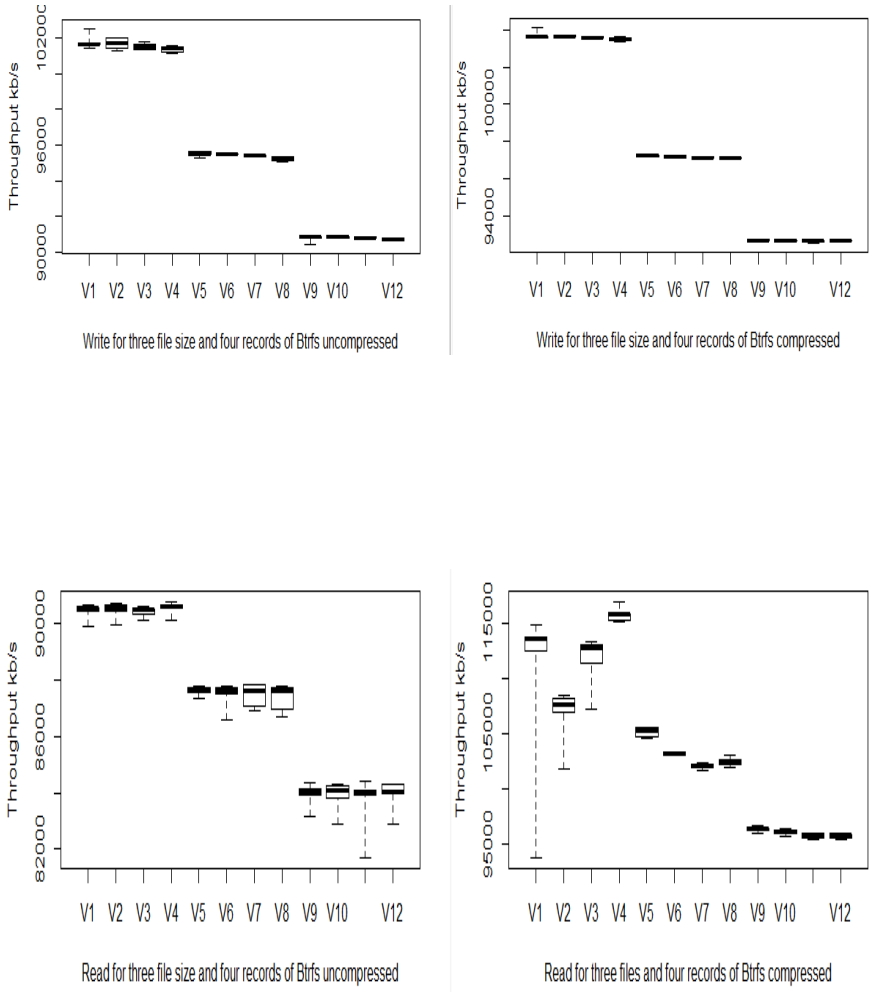


Figure B.3: Boxplot for Btrfs single Write Read result

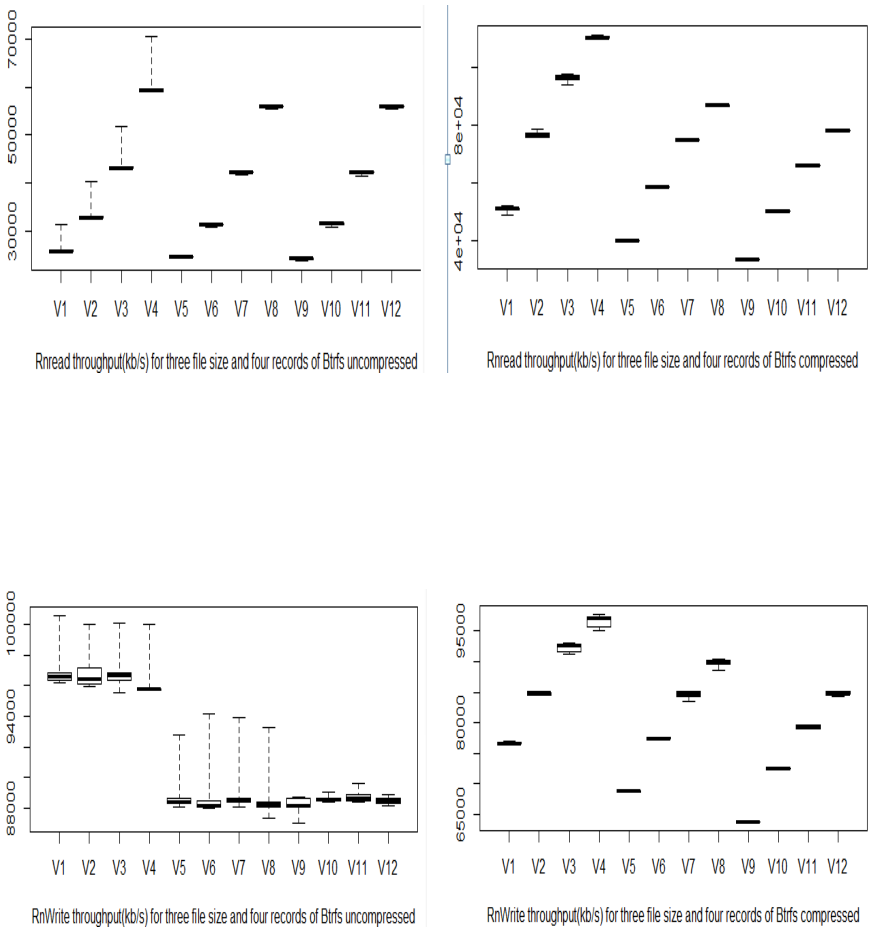


Figure B.4: Boxplot for Btrfs single Rnwrite Rnread result

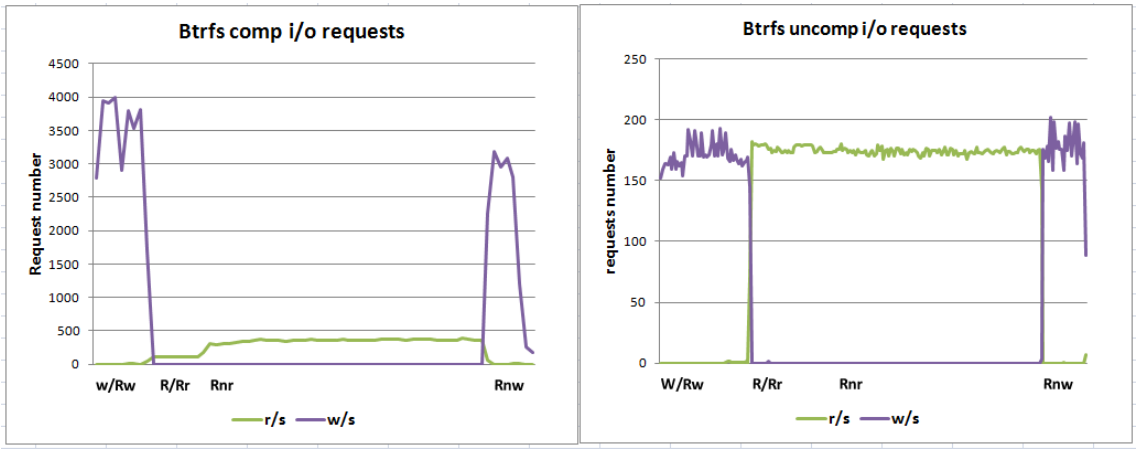


Figure B.5: Btrfs Comparison for IOPS Requests

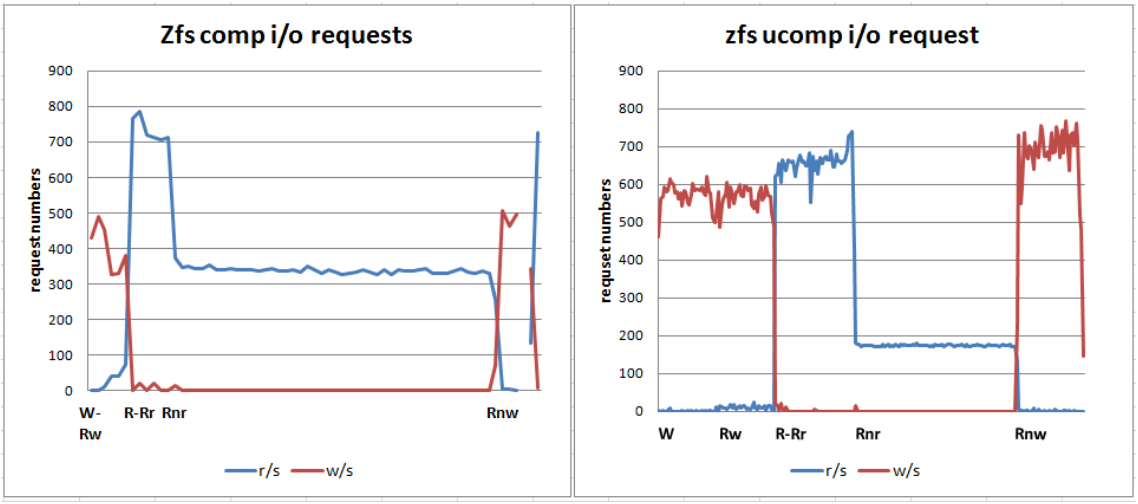


Figure B.6: Zfs Comparison for IOPS Requests

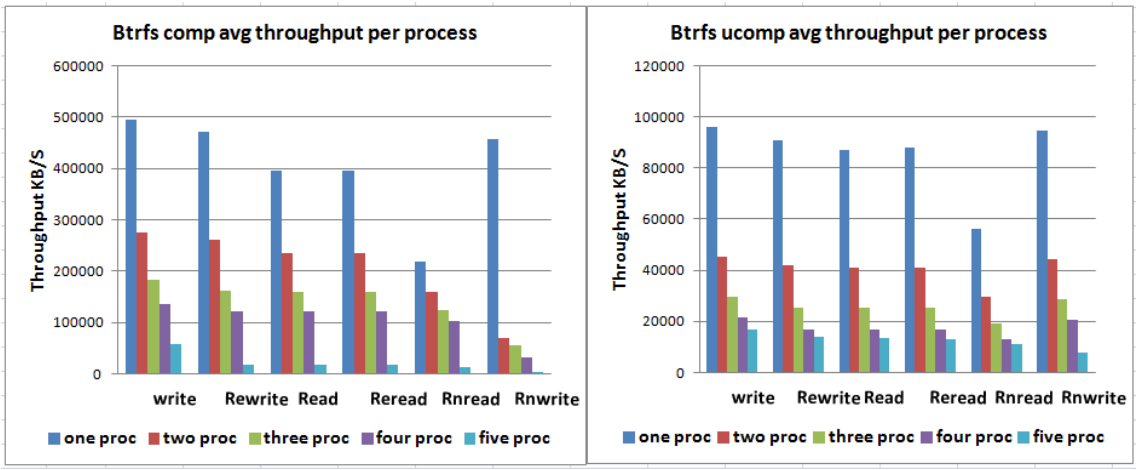


Figure B.7: Btrfs Compression Percentage of increase for single disk against multi processe

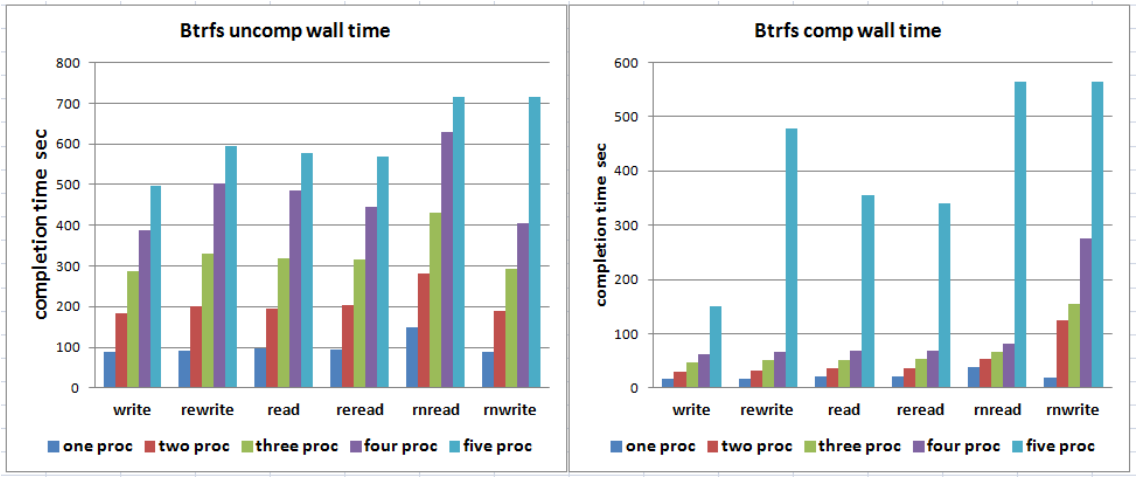


Figure B.8: Btrfs Single WallTime per processes

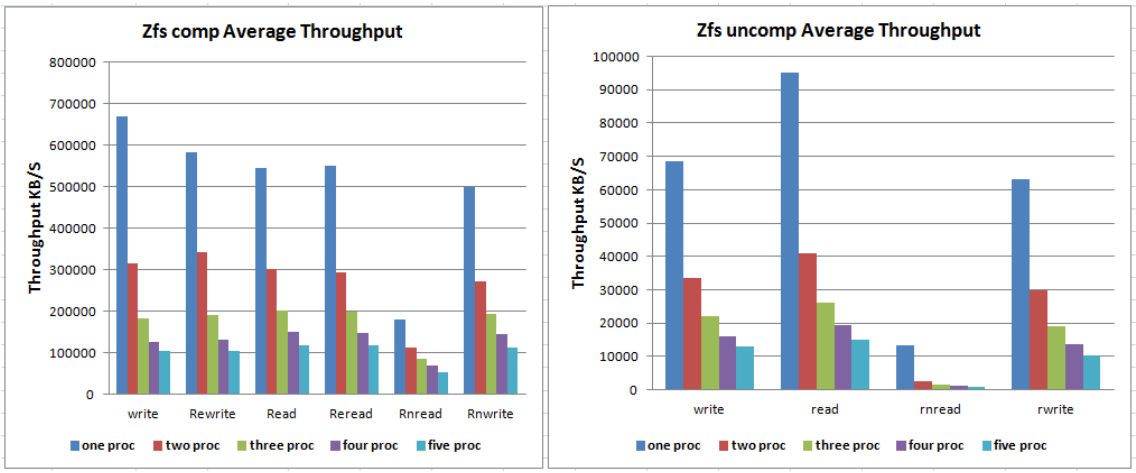


Figure B.9: Zfs Compression Percentage of increase for single disk against multi processe

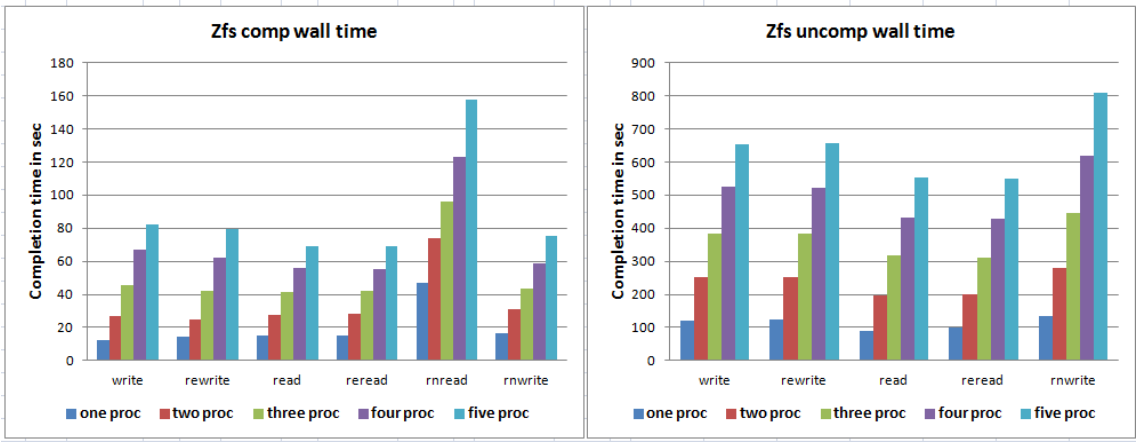


Figure B.10: Zfs Single WallTime per processes

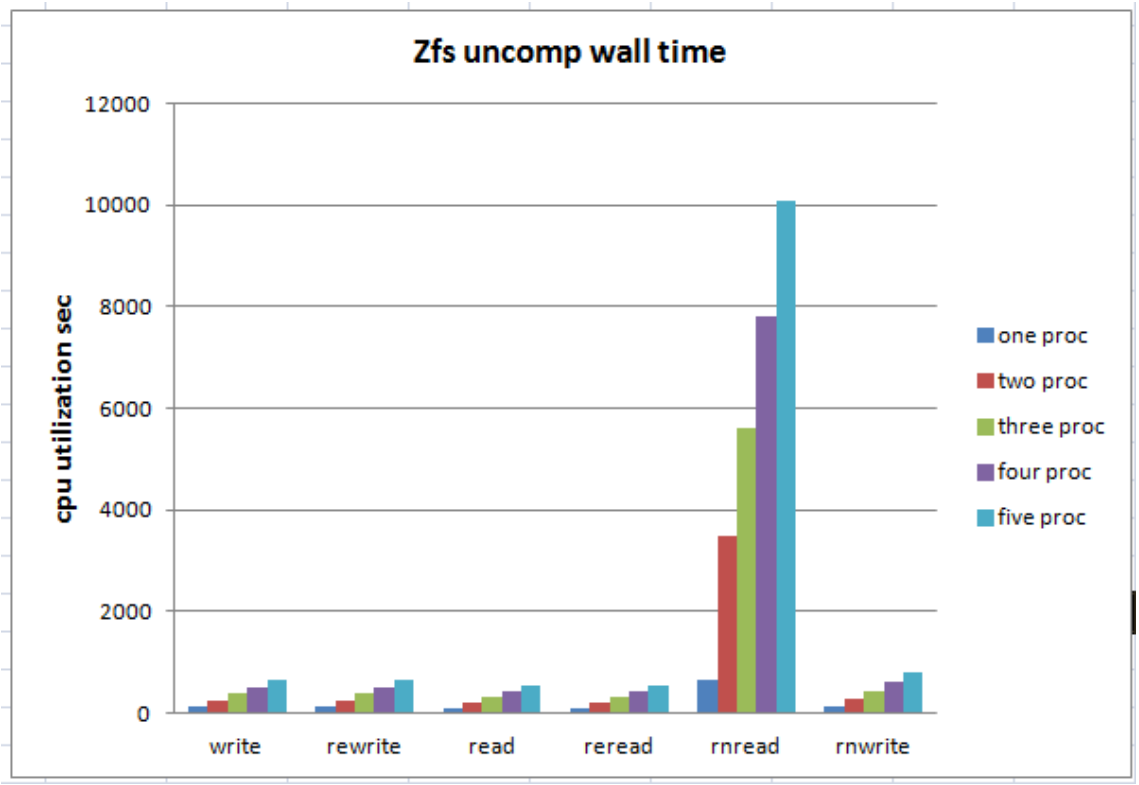


Figure B.11: Zfs Single WallTime per processes with Rnread

	user	system	iowait	idle	rrqm/s	wrqm/s	r/s	w/s	rMB/s	wMB/s	avgrq-sz	avgqu-sz	await	r_await	w_await	svctm	% util
W	222	986	-91	-75	-100	30083	601	2039	2833	-81	-99	-91	-100	-98	-100	-97	-36
R	290	484	-98	-67	338579	-96	-30	131	-84	-100	-77	-96	-94	-94	-75	-33	-54
Rnr	277	265	-19	-8	46746	-100	105		-92		-96	-60	-81	-81		-57	-12
Rnw	232	576	-51	-58	-100	8613	15231	1382		-88	-99	-96	-100	-95	-100	-93	-9

Figure B.12: Btrfs Single Impact of Compression Comparison, CPU and Disk Usage

Zfs new	system	iowait	idle	r/s	w/s	rMB/s	wMB/s	avgrq-sz	avgqu-sz	await	r_await	w_await	svctm	%util
Agwrati	944	-66	-51	11175	-40	261629	-91	-87	-67	-58	-87	-59	-10	-29
Agrratic	1617	-74	-36	11	359	-93	-42	-94	-72	-75	-75	-93	-49	-43
Agrnrra	125	-5	0	62	948	-99	3431	-99	-38	-54	-55	183	-48	-5
Agrnwr	1162	-63	-62	-91	-13	-100	-82	-81	-47	-30	-88	-24	-28	-21

Figure B.13: Zfs Single Impact of Compression Comparison, CPU and Disk Usage

Appendix C

Acronyms

Zfs: Zeta file system
Btrfs: Btree file system or Butter or Better file system
Ext4: Extended Family version 4 file system
LZ4: Real time Data Compression Protocol
LZO: Real time Data Compression Protocol
COMP: Compression
UNCOMP: Uncompressed
W: Write
R: Read
Rw: Rewrite
Rr: Reread
Rn: Random
Rnw: Random write
Rnr: Random read
Avg: Average
SEM: Standard error of Mean
proc: process
IO: Input output
r/s: Read Request per second in MB/S
w/s: write request per second in MB/s
Single: Single Disk Set up
Raid: Raid5 or Raidz1 set up