5-2011

Implementing Transparent Compression and Leveraging Solid State Disks in a High Performance Parallel File System

David Bonnie
Clemson University, dbonnie@clemson.edu

Implementing Transparent Compression and Leveraging Solid State Disks in a High Performance Parallel File System

A Thesis
Presented to
the Graduate School of
Clemson University

In Partial Fulfillment
of the Requirements for the Degree
Master of Science
Computer Engineering

by
David Bonnie
May 2011

Accepted by:
Dr. Walter B. Ligon III, Committee Chair
Dr. Stan Birchfield
Dr. Adam Hoover

Abstract

In recent years computers have been increasing in compute density and speed at a dramatic pace. This increase allows for massively parallel programs to run faster than ever before. Unfortunately, many such programs are being held back by the relatively slow I/O subsystems that they are forced to work with. Storage technology simply has not followed the same curve of progression in the computing world. Because the storage systems are so slow in comparison the processors are forced to idle while waiting for data; a potentially performance crippling condition.

This performance disparity is lessened by the advent of parallel file systems. Such file systems allow data to be spread across multiple servers and disks. High speed networking allows for large amounts of bandwidth to and from the file system with relatively low latency. This arrangement allows for very large increases in sustained read and write speeds on large files although performance of the file system can be hampered if an application spends most of its time working on small data sets and files.

In recent years there has also been an unprecedented forward shift in high performance I/O systems through the widespread development and deployment of NAND Flash-based *solid state disks* (SSDs). SSDs offer many advantages over traditional platter-based *hard disk drives* (HDDs) but also suffer from very specific disadvantages due to their use of *Flash memory* as a storage medium as well as use of a hardware *flash translation layer* (FTL).

The advantages of SSDs are numerous: faster random and sequential access times, higher *I/O operations per second* (IOPS), and much lower power consumption in both idle and load scenarios. SSDs also tend to have a much longer *mean time between failure* (MTBF); an advantage that can be attributed to their complete lack of moving parts.

Two key things prevent SSDs from widespread mass storage deployment: storage capacity and cost per gigabyte. Enterprise level SSDs that utilize *single-level cell* (SLC) Flash are orders of magnitude more expensive per gigabyte than their enterprise class HDD counterparts (which are also higher capacity per drive).

Because of this disparity we propose utilizing relatively small SSDs in conjunction with high capacity HDD arrays in parallel file systems like *OrangeFS* (previously known as the Parallel Virtual File System, or PVFS). The access latencies and bandwidth of SSDs make them an ideal medium for storing file metadata in a parallel file system. These same characteristics also make them ideal for integration as a persistent server-side cache.

We also introduce a method of transparently compressing file data in striped parallel file systems for high-performance streaming reads and writes with increased storage capacity to combat rising checkpoint sizes and bandwidth requirements.

Dedication

This paper is dedicated to all of my family and friends who have pushed me to succeed over the years.

Acknowledgments

I would like to thank my advisor Dr. Walt Ligon for both his support and direction. I would also like to thank Dr. Bradley Settlemyer for all the help he has offered for this and past works.

Table of Contents

List of Tables

List of Figures

Chapter 1

Introduction

1.1 High Performance Computing

From the first electronic computers to the fastest machines available today there has always been a drive for higher performance. Regardless of the speed of the newest machines being brought online there continues to be a push for them to be ever faster to run larger and more complicated calculations and simulations [11]. In the past this push contributed to the development of very large (at the time) computers deemed *supercomputers* that contained very powerful processors, large amounts of memory, and machine-specific high-speed communication networks. These supercomputers required programmers to know the details of the system to write programs that utilized the available resources efficiently because each machine varied from others and required different tool-sets.

In the last two decades there has been a nearly complete shift in the *high performance computing* (HPC) community away from large monolithic supercomputers to even larger distributed supercomputers made up of commodity parts [13]. These newer supercomputers contain anywhere from tens of cores in smaller systems to hundreds of thousands of cores [2]. Each *node* in these distributed supercomputers contains its own CPU or CPUs, memory, and communications interface. In general these systems are much easier to program for when compared to the monolithic supercomputers of the past. Each node tends to be identical and the tools in place to program on one system also tend to be very similar to the framework on any other so it is relatively simple to port code to a newer or different system.

These large distributed supercomputers are essential to the continuing progress of technology and research in many different fields of study. Without these large systems many advances in engineering, science, and technology would either take much longer to come to fruition or they would simply be impossible to complete in a reasonable timeframe. At the time of this writing many problems can not be performed on the fastest and largest supercomputers to the degree of accuracy that scientists and researchers desire for accurate results. Because of this, even for algorithms and simulations on the fastest systems in the world, there are trade offs that must be made to ensure that the results of a given experiment can be obtained in a timely and accurate manner [11].

1.2 I/O and HPC

Even though processors, memories, and communication interfaces have continued to progress at a very high rate there is one area that has been somewhat stagnant in comparison: storage [10]. As computation becomes more complex it also tends to require more data. Main memory sizes have been increasing steadily but the demand for data in most scientific applications exceeds the available memory on any given machine [11]. Because of this phenomenon the demands on the storage subsystems have increased steadily as well. Unfortunately these storage subsystems are performance-bound by a device that has remained relatively unchanged over the years: the mechanical spinning platter *hard disk drive* (HDD). The moving parts in a HDD limit it in a way that most other computer components are not. While other components can become faster through advances in silicon fabrication techniques the HDD is inherently limited by the speed of its rotating platters used to store data. Increasing the rotating speed of the platters does give some performance benefits but the physical problems of high rotation speeds (heat, vibration, and power consumption) have prevented speeds from further increasing past the levels found in the year 2000. Even today the fastest enterprise-level mechanical HDDs still rotate at 15,000 RPM and suffer from seek times similar to drives 10 years ago as well [28].

Because storage components are slow compared to processors many HPC programs can spend much of their time sitting idle while waiting for data to be either read from or written to disk [21]. Many programmers in the HPC community go to great lengths to reduce the amount of time spent doing I/O to avoid starving the processors of data and complete the job more quickly [24]. This disparity in performance becomes more significant as systems continue to grow in size [29].

Checkpointing is the process of taking a snapshot of the currently running program and saving it to disk. This process allows a program to be restarted from a specific checkpoint without recomputing all previous data. This is becoming more essential as supercomputers grow in size because as they grow the aggregate *mean time to failure* (MTTF) drops proportionally. While a single component in a node (like a power supply, HDD, or memory) may have a MTTF of 1,000,000 hours the combination of 100,000 or more of them in a single larger system can bring the MTTF of the whole system to as little as a few hours [27]. Instead of many years of runtime without error these large systems can sometimes only be run for hours before a failure occurs that requires service (and the subsequent loss of progress in any currently running programs).

Checkpointing alleviates this concern to a certain degree as long as it can be done in a timely manner without drastically increasing the program runtime. Checkpointing too often can make programs take much longer to complete than they would without checkpointing which increases the chance that the system will suffer a failure before computation is completed. On the other hand, checkpointing too infrequently likewise increases the chance that the system will suffer a failure before the current stage of computation is complete.

1.3 Solid State Disks

Recently an entirely new storage paradigm has developed: the *solid state disk* (SSD). SSDs have no moving parts and because of this they do not suffer from the rotation and seek delays that traditional mechanical drives do. Current SSDs utilize NAND Flash memory for persistent storage and employ a hardware *flash translation layer* (FTL) in the SSD drive controller to present the Flash memory modules as a standard disk drive to the host drive controller. This strategy eases deployment in current and future systems by removing the need for special hardware or software. This strategy does however present unique problems that must be overcome by the drive controller to perform adequately and reliably. Current SSDs do handle these problems relatively well and are further detailed in Section 2.1.1.

SSDs offer faster sequential and random access times, much higher sustained read/write bandwidth, and lower power use than traditional mechanical HDDs. The performance in random read/write workloads can be many orders of magnitude faster than HDDs. Typical enterprise HDDs can sustain approximately 350 *input/output operations per second* (IOPs) while the typical enterprise SSD can sustain well over

50,000 IOPs [28][25]. Sustained bandwidth is also much faster with typical SSDs delivering 250+ MB/s of sustained read/write bandwidth where standard HDDs average around 100 MB/s in fully sequential workloads [25]. The main benefit of SSDs lies in their high IOPs which allow them to sustain very high read/write bandwidth even with fully random workloads where traditional HDDs slow to roughly 1 MB/s or worse. Clearly this jump in performance can help close the growing gap between processing power and storage speed.

There are however some drawbacks that require careful consideration before SSDs can be deployed in a system. SSDs currently cost more per gigabyte and have smaller capacity than traditional HDDs. Their performance offsets this but does not make them suitable for widespread system deployment as a complete replacement for HDDs [22]. Because SSDs are scarce resources they must be intelligently integrated into systems to maximize their effectiveness without driving costs up exorbitantly or reducing capacity.

This presents a problem for programmers as well. Ideally a programmer should not have to know much about the underlying system to write programs to take advantage of large systems. Avoiding situations where programs must be tailored for specific systems for high performance is agreeable for many reasons including program portability, ease of programming, and cost of development.

1.4 Transparent Compression

As processors have become faster it has become easier to integrate *transparent compression* technology into various electronics. Transparent compression is intended to provide seamless on-the-fly data compression without any knowledge or intervention by the user. It can allow the use of smaller disks without a cost increase or the storage of more data on the same disks. Some SSDs even have transparent compression built in which allows them to read and write compressible data faster than the physical memory inside can support. This is done entirely without the knowledge of the user and for most workloads provides higher read and write bandwidths than would otherwise be possible [25]. Care must be taken to avoid slowing the system and wasting storage when incompressible data is presented to the compression algorithm. Doing such compression in software before any data hits disk comes at the price of higher CPU utilization for any given operation. However, with processors becoming faster and disks lagging behind, it is agreeable to trade CPU cycles for enhanced I/O performance [17].

1.5 File Systems

Writing programs for supercomputers would be extremely difficult without a system to easily access the underlying disk subsystems. *File systems* present a common interface to the users that is, ideally, consistent, portable, and high performance. These software constructs allow for integration of different storage technologies without requiring the user of the file system to know anything about the underlying hardware. Any two systems with the same underlying file system can be accessed the same way from the view of the user. The *Portable Operating System Interface* (POSIX) standard defines a standard operating system interface and the associated behaviors that must be upheld to stay consistent [5].

A file system is essentially a database that stores and retrieves data though a consistent interface. The file system itself also must store *metadata* which is data that describes the data being written to disk. This metadata includes file attributes, permissions, access history, file system structures, indexes, and other data about the data on disk. Keeping the metadata itself consistent on disk requires careful consideration of the order of writes and how data is written to disk. Because the metadata must stay consistent with the data on disk it also can be a performance bottleneck in many implementations. Increasing metadata performance can increase performance in the general case and can drastically increase performance with metadata-intensive workloads (workloads that create, access, or otherwise modify many files).

Another key component to file system performance is *caching*. Writing and reading directly to disk is extremely slow when multiple processes are accessing the disk. File systems generally have some level of caching integrated into them that coalesces multiple smaller writes into single larger writes to improve the efficiency of writing to disk. Along the same vein file system caches generally make use of system memory to store recently read and recently written data for fast read access. This mechanism allows for many disk operations to proceed at the speed of the cache instead of the speed of the disk [15].

1.6 Parallel File Systems

Fortunately, even though HDDs have not maintained the same level of progress as components like processors, the advent of the *parallel file system* has somewhat alleviated this performance disparity. These parallel file systems can achieve comparatively high levels of performance through the same metric that allows for many individual HPC nodes to perform well: the utilization of many slower components in unison [6]. These parallel file systems combine many HDDs on specialized nodes called *I/O nodes* to bring up the aggregate performance of the file storage subsystem. These I/O nodes sometimes contain the same hardware as all other nodes in the system but generally contain hardware specific to their task. Large *redundant arrays of* *independent disks* (RAIDs) are composed of many disks working in parallel for speed, redundancy, or both speed and redundancy [7]. Parallel file systems can take many of these RAIDs and combine them into a single logical file system that is presented to the system. Speed is gained through the distribution of data across these large arrays of HDDs and the associated I/O nodes' network connections.

Even with the advantages in parallel file systems the gap between compute power and the underlying storage subsystem is increasing. As processors continue to progress and further outstrip the performance of HDDs the same problem that faces large distributed supercomputers starts to become a problem for parallel file systems. As it takes more and more HDDs to keep up with the growing need for data the I/O subsystem itself becomes more prone to failure. Although expensive and relatively small in capacity SSDs can mitigate these issues. If intelligently integrated into a parallel file system SSDs can alleviate this growing problem by increasing performance and by lowering the complexity of the system.

1.7 Goals

Supercomputers require high-performance I/O subsystems to avoid data starvation and to avoid losing already completed work in the midst of a system failure. Parallel file systems facilitate these goals by providing a scalable storage subsystem. These same parallel file systems are increasingly at risk of failure as they grow in size to cope with the increasing capacity and performance demands of HPC workloads. SSDs offer much higher performance than their traditional mechanical HDD brethren. In light of these details we contend that **implementing storage of file metadata on SSDs, transparent file system compression for checkpoint efficiency, and caching with SSDs are effective methods of both increasing**

performance and overall system reliability within a parallel file system on distributed computers.

1.8 Approach

In order to explore the potential performance gains possible through the use of SSDs and transparent compression in parallel file systems, we first determine the underlying characteristics of the problem, the potential performance of SSDs, and the feasibility of transparent compression in a parallel file system. We then analyze the results to best determine how to integrate these technologies into a parallel file system without drastically increasing cost, increasing system complexity for users, or regressing performance. As we show in Section 4.2, there is a potential performance impact with transparent compression that depends very heavily on the workload being performed. The workloads that benefit are detailed as well.

In order to measure the performance impacts of these technologies we have implemented metadata storage on SSDs as well as basic transparent compression capabilities and formulated tests to assess the potential performance of an SSD read and write cache within *OrangeFS*. OrangeFS is a branch of the *Parallel Virtual File System 2* (PVFS2) that focuses on small file operations, metadata optimization, and cross-server redundancy. OrangeFS is an open source parallel file system in continuing development at Clemson University that is designed to perform both as a testbed for research in parallel I/O as well as a fully functional high performance parallel file system. It is an effective platform for research because of its ease of modification as well as its consistently high performance. We perform quantitative analysis of the performance impacts under various workloads of the technologies outlined above.

In Chapter 2 we describe further the technologies involved in SSDs and parallel file systems. We also describe other research projects that address the performance of parallel file systems with SSDs. In Chapter 3 we describe the systems, methods, and reasoning behind the modifications performed on OrangeFS. In Chapter 4 we explore the performance characteristics of our parallel file system modifications. Finally, in Chapter 5, we summarize the results of our work and identify possible future research possibilities related to integrating SSDs and transparent compression into parallel file systems.

Chapter 2

Background and Related Work

2.1 Solid State Disks

2.1.1 Basics

Recently a new type of disk has been brought to market on a large scale: the solid state disk (SSD). Until recent years such disks have existed but did not see widespread use in the enterprise or consumer markets. SSDs are essentially large parallel arrays of NAND Flash memory with a drive controller that presents the Flash memory as a standard disk drive to a disk controller in a computer. With the exception of a few commands specific to SSDs these drives behave exactly like standard spinning platter hard disk drives (HDDs) as far as the host disk controller is concerned. The use of such a controller with a hardware *flash translation layer* (FTL) allows for the addition of high-speed SSDs to nearly any system without special hardware or software. The basic design of an SSD is shown in Figure 2.1.1.

These Flash drive controllers manage nearly every aspect of a drive's performance. Early SSDs used relatively primitive Flash controllers and as such did not

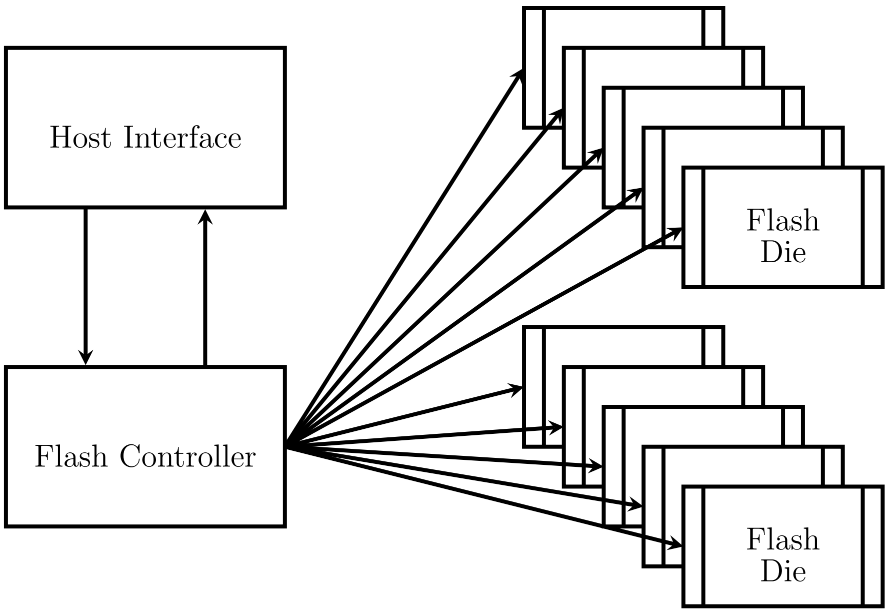


Figure 2.1: Basic hardware configuration of an SSD.

perform well enough to justify their increased cost per gigabyte and lack of capacity for most cases. These early controllers did not handle workloads with high frequency random-write requests very gracefully and this severely hampered their performance in common installations. These early controllers also tended to slow down considerably with mixed reads and writes. Modern SSDs like the Intel X-25E utilize an advanced controller that efficiently handles concurrent read and write requests whether they are sequential or random in nature.

A good Flash controller in an SSD would ideally do the following: minimize write amplification, maximize Flash parallelism, and minimize request/response time. Write amplification is described more in Section 2.1.3. Flash parallelism and minimization of request/response times are described further in Section 2.1.2.

There are two major types of NAND Flash devices in use today: *single-level cell* (SLC) and the more dense *multi-level cell* (MLC). SLC Flash stores a single bit per cell while MLC Flash stores at least 2 bits per cell but in many cases stores 3 or

4 bits per cell. MLC Flash is naturally more dense thus it is cheaper to produce an SSD with MLC Flash at the same storage capacity as an SSD with SLC Flash. There are performance implications when storing multiple bits per cell in MLC Flash that manifest themselves in slower read and write response times to the Flash memory as well as durability implications. More details on the differences between MLC and SLC Flash memories are listed in Section 2.1.3.

2.1.2 Advantages

The advantages of SSDs over HDDs are numerous. In general, they provide faster response times, higher bandwidth, and lower power usage. The first two advantages are beneficial from a performance standpoint and the lower power usage drives down the high cost of maintaining a large high-performance HDD array.

To achieve the fast response times desired from an SSD the drive controller must efficiently be able to handle concurrent reads and writes without large amounts of slowdown. Because a single Flash chip can only be read from or written to at a single time the drive controller must interleave and combine requests as much as possible. Reading from a Flash chip is a relatively fast procedure when compared with writing to a Flash chip. Another key component to fast response times is an adequate caching mechanism in the drive controller to coalesce multiple write requests into a single large internal write request to the Flash chips. Further explanation of complications when writing to Flash memory are explained in Section 2.1.3.

Achieving high data rates as well as low response times for both sequential and random read or write requests on an SSD requires the drive controller to maintain a high level of parallelism in its requests to the Flash memories beneath it. A single Flash memory die can sustain data rates in the area of 40 MB/s with current Flash technology. To obtain higher performance the SSD drive controller must interleave read and write requests to multiple Flash memories. The more channels an SSD drive controller can access at one time the higher potential read and write bandwidth.

In general, SSDs also use much less power under both idle and load conditions when compared to standard HDDs. This lower power is due to the lack of moving parts: SSDs require no motors, spindles, or read/write heads. This trait, when combined with their higher performance per drive, can drastically reduce the power usage of a high-performance disk array. The lower power usage also manifests itself in the far lower production of heat which further reduces the cost of maintaining the array through lower cooling costs.

2.1.3 Disadvantages

The disadvantages of SSDs vary in severity based upon the type of Flash storage used and what drive controller is used to present the Flash as a disk to the storage controller. Flash by nature requires erasure before it can be written. Standard HDDs can simply overwrite old data without any prior knowledge of the data for writes of block size or larger and sub-block writes require read-modify-write operations. Block sizes for HDDs tend to be very small, on the order of a few kilobytes, whereas erase blocks in SSDs tend to be hundreds of kilobytes. In order to write data to a previously written section of Flash memory the prior contents must be read, the new data merged with the old, and the entire set written back to the Flash. This process, depending on the Flash controller in the SSD, can incur a considerable performance penalty that sometimes slows writes to the point where a standard HDD would complete the write requests more quickly.

The block size of current-generation MLC NAND Flash compounds this problem: writes can be done at the page level (4 KB) but erasures can only be done at the Flash block level, typically 128 pages or 512 KB with MLC Flash. A single 4 KB write can require the reading of the whole block, pruning of the block for free pages, erasure of the block, and then up to a 512 KB write [19]. This behavior is not ideal but can be mitigated by intelligent Flash management in the drive controller and background garbage collection to consolidate free space. While most algorithms for such Flash management are proprietary they do tend to keep data organized to avoid requiring an erase cycle when writing data.

Another key issue with Flash memory as storage is that Flash cells can only be written a finite number of times before losing their ability to be written (whereas standard HDDs essentially have infinite write endurance). With continuing decreases in the minimum feature size of Flash the number of erase/write cycles per flash block has been decreasing. MLC NAND Flash memory produced at the 22-25 nanometer level typically endures 3,000 to 5,000 erase/write cycles before writes will fail. Previous MLC NAND Flash produced on the 35 nanometer process endured roughly 10,000 erase/write cycles. SLC Flash memory tends to endure approximately 100,000 erase/write cycles before write fail [19].

Wear leveling algorithms in high-performance SSDs mitigate the issues involved with Flash write endurance but do not eliminate it. Many modern drive controllers are intelligent enough to wear-level on the fly to reduce the performance penalties of consolidating free space during a write request. The advantages of this strategy are that no single Flash cell is written to more than necessary (because writes are spread out between Flash cells in the device) and that performance is generally better than naively writing sequentially to a Flash chip. Background garbage collection routines allow for higher performance at the cost of using more erase/write cycles to consolidate free space. Overly aggressive garbage collection can result in quicker than desired Flash exhaustion and overly passive garbage collection routines can result in slow performance after most of the Flash chips in the SSD have been previously written to (thus reducing the performance when reading and writing to the speed that the drive controller can run the garbage collection routine).

Many of these disadvantages are minimized but not eliminated with current high-performance drive controllers. Care is required to choose the correct SSD for a specific application or deployment. Choosing an unsuitable SSD for an application could result in very low performance, shorter lifetime than expected before Flash exhaustion, or both. Workloads with high levels of random writes are likely more suitable for SLC SSDs and workloads that are either primarily read-based or mostly sequential could make use of MLC SSDs without issue.

2.2 Parallel File Systems

As mentioned briefly in Section 1.6, parallel file systems can improve performance by aggregating the bandwidth of many individual systems under a common interface presented to the *computation nodes* in a computing cluster. Computation nodes access the parallel file system much like a traditional network file system by mounting the file system over whatever communication network is available as seen in Figure 2.2. This allows the computation nodes to access data contained on the parallel file system as if it were local to the node. The primary difference between traditional network file systems and parallel file systems is that parallel file systems are designed to stay consistent when being accessed by multiple clients. Traditional network file systems will allow multiple clients to read from a file at once but tend to break down in either performance, consistency, or both when multiple clients attempt to write to a single file [26].

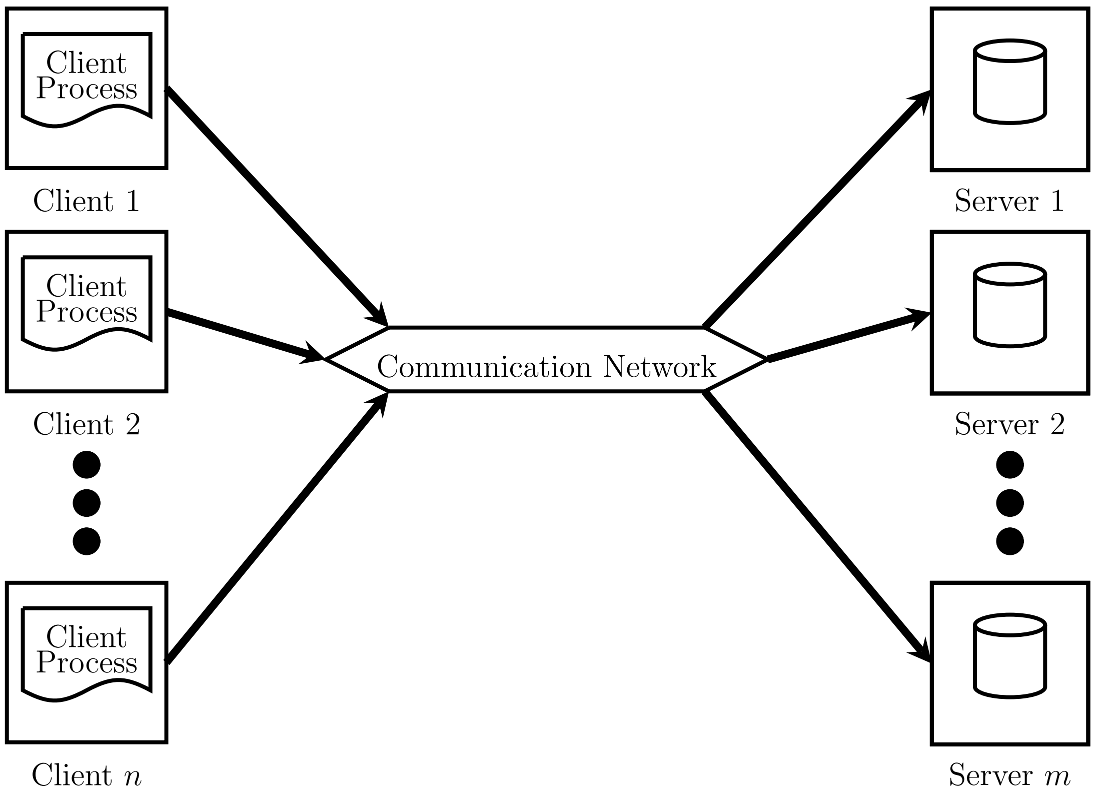


Figure 2.2: Parallel file system hardware configuration with n clients and m servers.

Parallel file systems mitigate the issues of traditional network file systems by striping user data across I/O nodes. Much like a RAID 0 array stripes data across disks, parallel file systems tend to stripe data across servers to provide higher aggregate bandwidth to client processes. Depending on the configuration of the particular parallel file system user data may be striped across all I/O nodes or just a subset of them. This design also provides an inherent type of link aggregation where the network connections from each server work together in parallel to provide data to the client. As more servers are added to a system the potential peak performance increases along with the storage capacity.

To access a parallel file system a client process must first contact a metadata server in order to gather information about the file or files it wants to open. Depending on the design of the parallel file system there can be anywhere from a single metadata server up to as many metadata servers as I/O nodes. Once a client process has received the relevant metadata it can contact one or more data servers to begin downloading the data. These data servers may be the same as the metadata servers or they may be completely different.

Actual access to the parallel file system over the network can be achieved in a number of ways. Many parallel file systems provide a kernel module that allows the file system to be accessed like a standard path in the local file system. Programs do not need to be altered to take advantage of the storage provided by the parallel file system. This method provides easy access to the parallel file system but does not allow for easy optimization of file system access though collective file routines. Another way to access many parallel file systems is through the *MPI-IO* layer built into the *Message Passing Interface* (MPI) specification. The MPI-IO layer allows programmers to natively perform collective file operations; a necessary optimization to achieve the best parallel performance on any given file system. Some parallel file systems also expose a native *application programming interface* (API) that allows programmers to directly call functions native to the file system for the highest possible performance.

2.2.1 OrangeFS

OrangeFS was chosen as the testbed parallel file system for modifications for many reasons. First, it is a fully open source project from Clemson University that is designed from the ground up to be modular in nature and thus is relatively easy to modify for research purposes. Unlike some other parallel file systems it is not restricted to a single metadata server [20][26]. This trait offers increased parallelism on metadata operations where file systems with a single metadata server tend to scale poorly under increased load. OrangeFS supports many different network architectures like Infiniband, Myrinet, Ethernet, and more. OrangeFS also performs consistently well on a wide range of hardware and software for a wide variety of workloads. All of these traits contributed to the choice of OrangeFS as the parallel file system to be modified for this research.

2.3 Related Work

Much research has been done in both the academic world as well as the commercial world to integrate SSDs into HPC and to examine their distruptive nature within storage technology. Sun Microsystems has studied integration of SSDs as part of the Lustre *Object Storage Server* (OSS) configurations where the SSDs are tasked with simple data storage [18]. Sun Microsystems has also implemented SSDs into its ZFS file system as part of a tiered storage design with SSDs acting as a larger, slower, main memory [14]. EMC also places SSDs into a tiered storage design with their Symmetrix DMX-4 storage system [9]. IBM offers the ability to stage data to and from SSD arrays within their storage products [8]. The Ceph distributed file system recommends use of SSDs on *object storage devices* (OSDs) to improve overall disk performance (no metadata is stored on the metadata servers; it is stored on the OSDs) [4][30]. PanFS by Panasas recently integrated SSDs into their metadata storage nodes to accelerate metadata operations and small file access [23].

Makatos describes methods of integrating transparent compression in an SSD caching system [17]. Lee studies the application of SSDs in database applications [16]. Dirik analyzes the limitations of current SSDs and proposes multiple designs to improve efficiency and performance [12]. Kgil analyzes the use of SSDs as a disk cache and proposes several ways to improve cache performance [15].

Chapter 3

Research Design and Methods

This chapter will outline the design and methods used to modify and evaluate OrangeFS for both split metadata and data paths as well as for transparent compression and server-side caching analysis.

3.1 Leveraging SSDs for Fast Metadata Access

Before any modification of the file system was performed the baseline performance a typical HDD and SSD were determined on a somewhat older test system. The test system at the time of this writing is somewhat dated and is limited in a number of ways that will restrict performance significantly for certain use cases. While the test system is older it will show that even with its limitations SSDs can outperform HDDs by many orders of magnitude. The sustained read and write performance of each type of disk is important for any kind of large sustained transfers to or from the file system. The random read and write performance give an indication of the maximum speed possible for any kind of operation from metadata access to small file I/O. Both sustained and random read/write performance will be tested. As we will

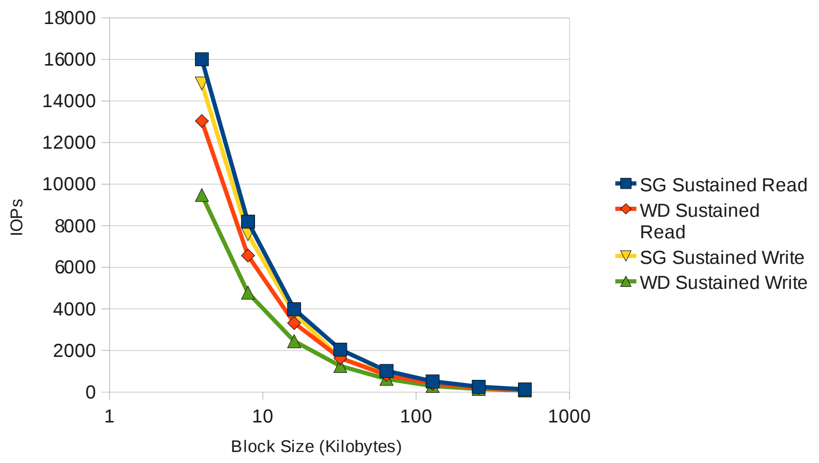


Figure 3.1: Comparison of IOPs during sustained read/write operations. SG: ST3500630NS, WD: WD800AAJS

show, the dramatic increase in random read and write rates that SSDs offer can be used to accelerate many common operations within a parallel file system.

All tests involving the use of SSDs for fast metadata access were performed on a subset of 8 nodes on a cluster system with a total of 46 nodes. The first node contains an Intel Xeon 3040 1.86 GHz CPU and the seven other nodes contain an Intel Xeon 3070 2.66 GHz CPU. The first node has 2 GB of DDR2 ECC memory and the seven other nodes contain 4 GB of DDR2 ECC memory and all use the Intel 975 Express chipset. Each node is connected via gigabit Ethernet on a gigabit switch. The first node contains a 500 GB Seagate Barracuda ES HDD (model ST3500630NS) and the other seven nodes each contain one 80 GB Western Digital Caviar Blue HDD (model WD800AAJS). The performance of the two HDD models is roughly similar as seen in Figure 3.1 and Figure 3.2. Although the Seagate model is slightly faster in sustained write rates the sustained read, random read, and random write are all comparable.

Individual testing of HDDs and SSDs was performed on the first node. All nodes were running CentOS 5.5 x86_64 with kernel version 2.6.18-194.26.1.el5 SMP.

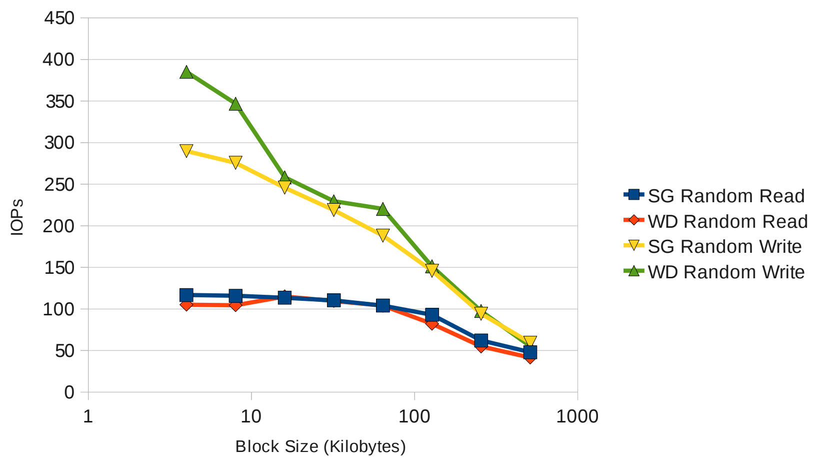


Figure 3.2: Comparison of IOPs during random read/write operations. SG: ST3500630NS, WD: WD800AAJS

Nodes 1 through 7 contain an Intel X-25E 32 GB SSD. This SSD is composed of SLC Flash and is considered enterprise-class for both its performance and reliability. All HDDs and SSDs are formatted with the ext3 file system. The defaults for ext3 were used with the exception that the SSDs are running with the *noatime* option on mount. The noatime option removes the requirement of updating a file's access time stamp on every access. The default scheduler of *completely fair queuing* (CFQ) was used for all HDDs and SSDs as well. The *noop* scheduler was briefly tested on the SSDs for performance improvements but real-world gains were minimal in initial benchmarking without *Advanced Host Configuration Interface* (AHCI) mode enabled.

Before modifying OrangeFS for integration with SSDs the disks themselves were tested on the system to create a baseline performance for each of them and to ensure that there were no unknown or unexpected factors affecting disk performance on the cluster nodes. These initial tests were performed using the *Iozone Filesytem Benchmark* [1]. This benchmark tests local file system performance using memory-mapped file I/O with POSIX threads. Test sizes were set at 8 gigabytes to avoid buffer cache effects as much as possible. The host disk controller in the Intel ICH7

Iozone Version 3.373 Settings	
4 KB block	iozone -Rb test4k.xls -i0 -i1 -i2 -+n -r 4k -s8g -t2 -F /ssd/iozone1 /ssd/iozone2
8 KB block	iozone -Rb test8k.xls -i0 -i1 -i2 -+n -r 8k -s8g -t2 -F /ssd/iozone1 /ssd/iozone2
16 KB block	iozone -Rb test16k.xls -i0 -i1 -i2 -+n -r 16k -s8g -t2 -F /ssd/iozone1 /ssd/iozone2
32 KB block	iozone -Rb test32k.xls -i0 -i1 -i2 -+n -r 32k -s8g -t2 -F /ssd/iozone1 /ssd/iozone2
64 KB block	iozone -Rb test64k.xls -i0 -i1 -i2 -+n -r 64k -s8g -t2 -F /ssd/iozone1 /ssd/iozone2
128 KB block	iozone -Rb test128k.xls -i0 -i1 -i2 -+n -r 128k -s8g -t2 -F /ssd/iozone1 /ssd/iozone2
256 KB block	iozone -Rb test256k.xls -i0 -i1 -i2 -+n -r 256k -s8g -t2 -F /ssd/iozone1 /ssd/iozone2
512 KB block	iozone -Rb test512k.xls -i0 -i1 -i2 -+n -r 512k -s8g -t2 -F /ssd/iozone1 /ssd/iozone2

Table 3.1: Settings used for SSD and HDD comparison tests.

I/O controller hub does not support ACHI mode and because of this *native command queuing* (NCQ) is disabled. This effectively limits both the SSD and HDD to disk queue depths at the host disk controller to a single request. This limit does affect performance when multiple threads or multiple requests attempt to access the drive simultaneously and can limit performance with single-threaded performance as well. With this limitation in the host disk controller the results from benchmarking will be slower than would be possible on an AHCI compliant disk controller. While this limits the maximum performance of the disks their relative performance is still relevant and provides a floor for the expected performance differences. The exact settings used for the Iozone benchmark are listed in Table 3.1.

As seen in Figure 3.3 through Figure 3.4, the sustained read speed and sustained read IOPs of the SSD at all block sizes is significantly higher than the mechanical hard drive. Figure 3.5 through Figure 3.6 show that the sustained write

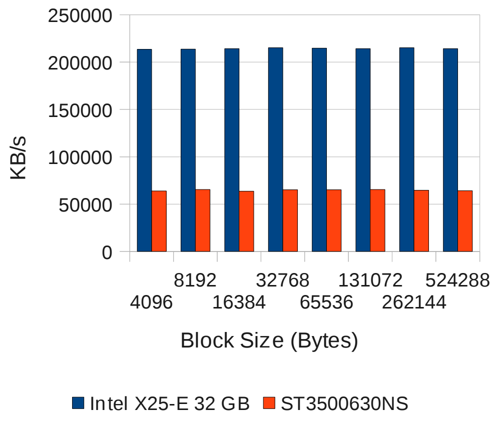


Figure 3.3: SSD versus HDD sustained read bandwidth.

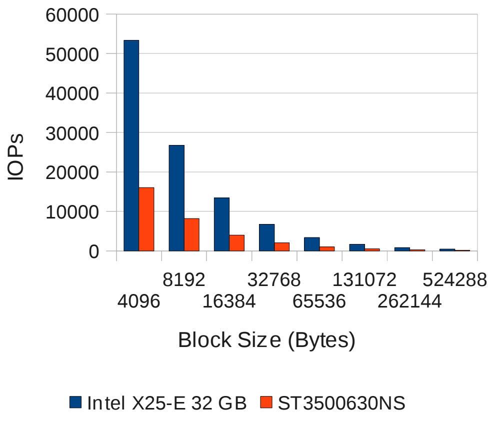


Figure 3.4: SSD versus HDD sustained read IOPs.

speeds and sustained write IOPs of the SSD are also significantly higher than the HDD. These results are expected and within line of expectations for both the SSD and HDD.

Figure 3.7 through Figure 3.8 show that the random read speeds and random read IOPs of the SSD are orders of magnitude higher than the HDD at all block sizes except for the very largest where the SSD is still over three times faster. Figure 3.9 through Figure 3.10 show that the random write speeds and random write IOPs are again orders of magnitude faster on the SSD with the exception of the largest block size where the SSD is nearly three times faster than the HDD. The large speed differences at all block sizes highlight the drastic improvement in file I/O that can be possible when utilizing a high-performance SSD.

After confirming baseline performance for the HDDs and SSDs in use, modification of OrangeFS was performed. As data and metadata were already stored in different forms on disk within OrangeFS, splitting the storage of data and metadata was conceptually simple. OrangeFS previously required a single path to specify a path on the local file system for storage of all server data. Because the modification

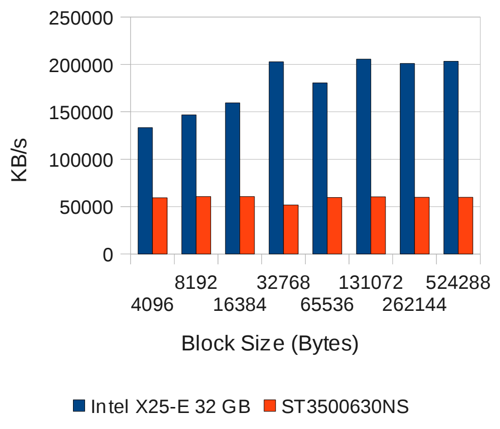


Figure 3.5: SSD versus HDD sustained write bandwidth.

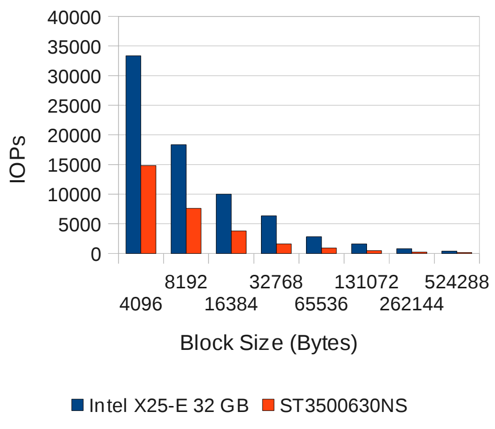


Figure 3.6: SSD versus HDD sustained write IOPs.

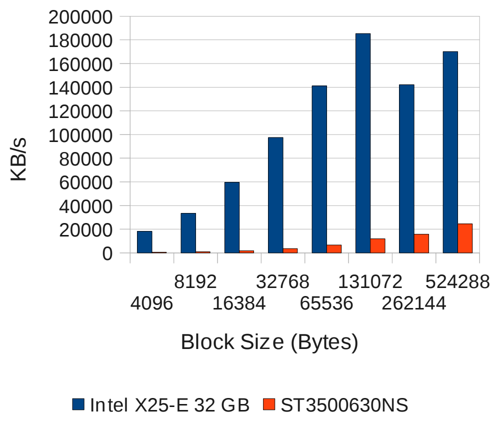


Figure 3.7: SSD versus HDD random read bandwidth.

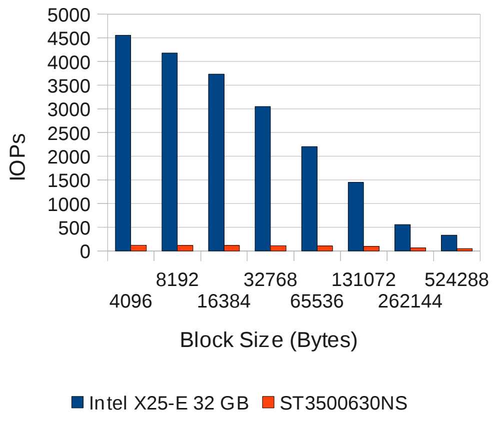
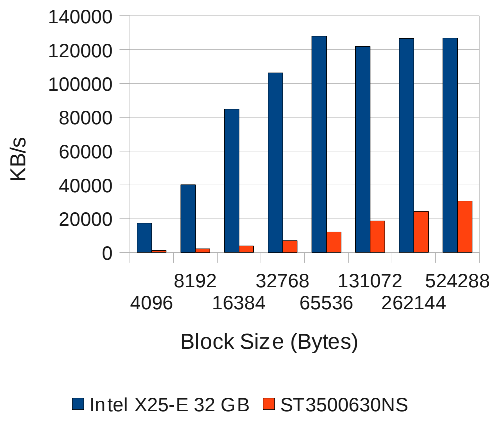


Figure 3.8: SSD versus HDD random read IOPs.



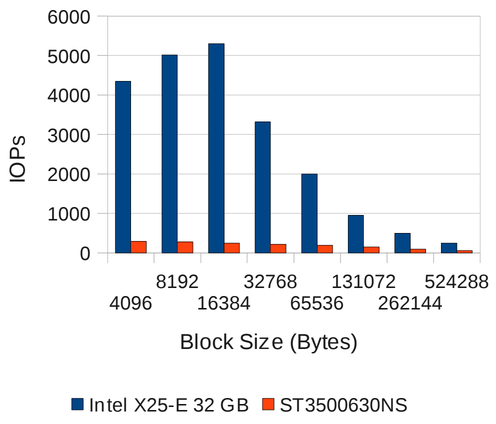


Figure 3.9: SSD versus HDD random write bandwidth.

Figure 3.10: SSD versus HDD random write IOPs.

of OrangeFS was intended as a long-term modification and not a simple feasibility test the entire structure defining where the servers stored data was revamped. At the lowest levels of OrangeFS within the Trove storage subsystem the path to server data was split into two separate arguments. Some calls to Trove required both the path to the data and the path to the metadata; these calls were modified to accept the extra metadata path argument where needed. The functions modified within the Trove storage subsystem control the initialization, creation, deletion, and migration of OrangeFS metadata and data collections.

The changes in Trove were propagated up through all other levels of OrangeFS. All server code is fully aware of the split metadata paths as is all client code. OrangeFS also contains in its distribution a set of administrative and testing programs to ease deployment and maintenance as well as to reduce downtime. These programs were also modified to reflect the changes made to the OrangeFS storage hierarchy.

The storage locations for the data and metadata differ depending upon what role the server is playing in the file system:

Role 1: Server is acting as an I/O server *and* as a metadata server.

- All metadata (for user data as well as metadata for the OrangeFS data storage) is stored at the metadata storage path.
- All binary user data streams are stored at the data storage path.

Role 2: Server is acting as an I/O server *only*.

- When the server acts only as an I/O server it must also store the metadata for its own data. This metadata can either be stored at the metadata path (if it exists) or it can be stored in the data path.
- All binary user data streams are stored at the data storage path.

Role 3: Server is acting as a metadata server *only*.

- All user metadata is stored at the metadata path. No metadata for the OrangeFS data storage is located on a pure metadata server.
- No binary user data is stored on a metadata-only server.

By default the new configuration will place all metadata in the metadata path and all data in the data path (like in role 1 above). The configuration file can be used to specify exactly where data and metadata are stored on a per-server basis if required. This is helpful in the case where a data server does not have the same storage paths as a metadata server. This case would be common if nodes were specialized by task with the metadata servers containing SSDs and the data servers running large RAIDs.

From the standpoint of a normal user of the parallel file system nothing has changed. From an administrative perspective the only real change comes when configuring the servers for initial startup and the command line arguments of some of the basic functionality test programs.

3.2 Transparent Compression in OrangeFS

Before modifying OrangeFS in any way a number of tests were performed in order to determine the feasibility and baseline performance of various compression algorithms. The ability of any given compression algorithm to compress data efficiently is essential to its performance when included within the file system. In this case, compression efficiency refers to the ability of a compression algorithm to not only compress data quickly but also its ability to reduce the size of the data. An algorithm that compresses data extremely quickly without reducing data footprint is not ideal for integration as it will not lessen the load on the underlying disk subsystem. At the opposite end of the spectrum an algorithm that compresses data into a very small footprint but takes a long time to do so is also not desirable because it will leave the disks sitting idle while compressing data. The ideal compression algorithm would both compress data well and do it quickly but trade-offs must be made. For this study it is preferable to use a compression algorithm that offers both good speed and good compression rates without focusing too much on either trait. This will allow the parallel file system to compress data quickly and write it to disk without wasting too much time doing either.

Initial benchmarks to find a suitable compression library were performed on a set of files determined to be representative for various workloads. Compression speed and decompression speed were both considered in the selection of a compression algorithm. The algorithm with the fastest average compression and decompression times was chosen with a strong emphasis on compression speed. Emphasis was placed on compression speed because writing checkpoint data needs to be done as quickly as possible and happens more often than resuming from a checkpoint. Compressed size was considered after compression speeds were taken into account. Behavior with incompressible data was also considered to avoid increasing data footprint and slowing throughput.

Transparent compression tests were performed on a Dell Precision 390 with a 1.86 GHz Core 2 Duo E6300 CPU, 2 GB of DDR2 ECC memory, running Fedora 12 x86_64 with kernel version 2.6.32.26-175 SMP. To test various compression algorithms a simple benchmark from the QuickLZ website was modified [3]. This benchmark reads a single file into memory and subsequently compresses and decompresses it with various compression algorithms shown in Table 3.3. Since this is done in-memory the disk speed of the benchmarking machine does not affect the results. Table 3.2 shows the various files used in the test and their properties. The benchmark was modified to run each compression test ten times in a row and then average the result in MB/s. The same was done for the decompression tests. The results from the various file types are shown in Figure 3.11 through Figure 3.25. This benchmark led to the choice of QuickLZ as the compression library used in further testing with OrangeFS as it displayed the highest average compression speed, respectable decompression speed, the smallest compressed file size average, and good behavior with nearly incompressible data.

After determining that QuickLZ was the most suitable for integration into OrangeFS; OrangeFS itself was modified to test performance and feasibility. The underlying I/O system of OrangeFS is Trove. Trove handles all interactions with the local file systems on the OrangeFS server for both metadata and data operations. As mentioned previously in Section 3.1, the storage collections for metadata and data were split to allow for storage of metadata on fast media (like SSDs) and data on large

Representative Files	
plaintext.txt	A simple text file containing 89,000 lines of plaintext. 2.9 Megabytes.
gdb.exe	The GNU Project Debugger executable. 8.5 Megabytes.
pic.bmp	A bitmap picture of a yellow flower. Relatively incompressible. 18.0 Megabytes.
proteins.txt	Protein data (DNA/RNA) in text form. 89,000 lines. 7.1 Megabytes.
NotTheMusic.mp4	Highly incompressible audio data. 9.4 Megabytes.

Table 3.2: Representative files chosen for benchmarking compression test.

Compression Algorithms and Settings	
FastLZ 0.1.0 1	Compression level 1
FastLZ 0.1.0 2	Compression level 2
LZF 3.1 VER	"Very fast" setting
LZF 3.1 ULT	"Ultra fast" setting
LZO 1X 2.02	Setting 1
QuickLZ 1.50	Setting 1

Table 3.3: Compression algorithms benchmarked and their settings.

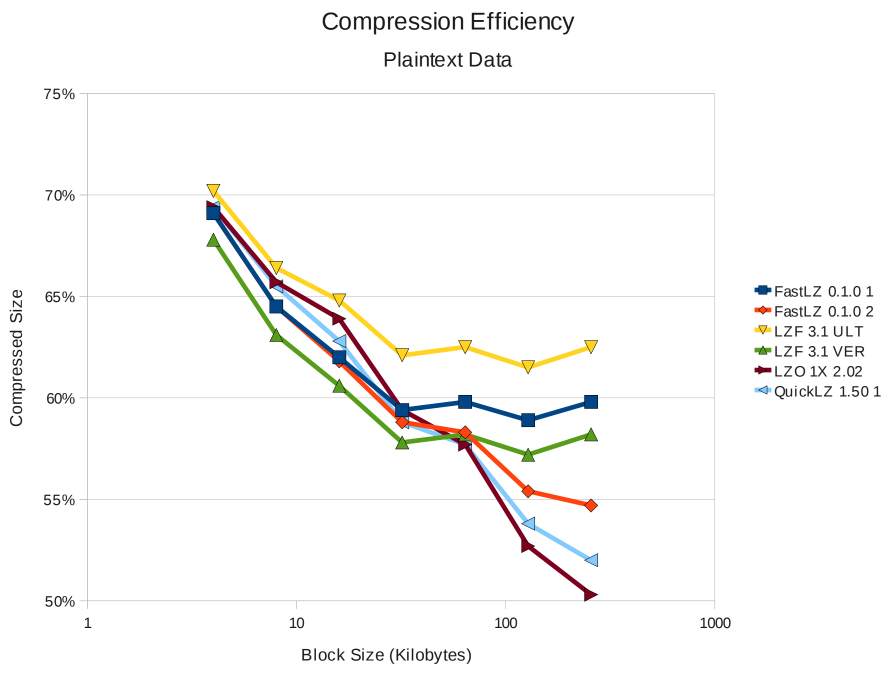


Figure 3.11: Plaintext data compressed file size by compression block size.

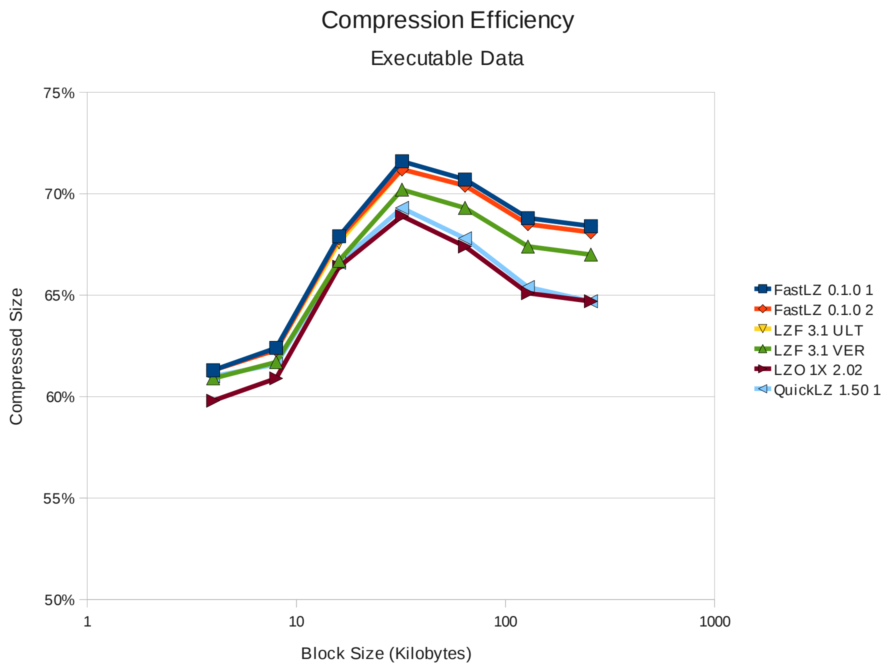


Figure 3.12: Executable data compressed file size by compression block size.

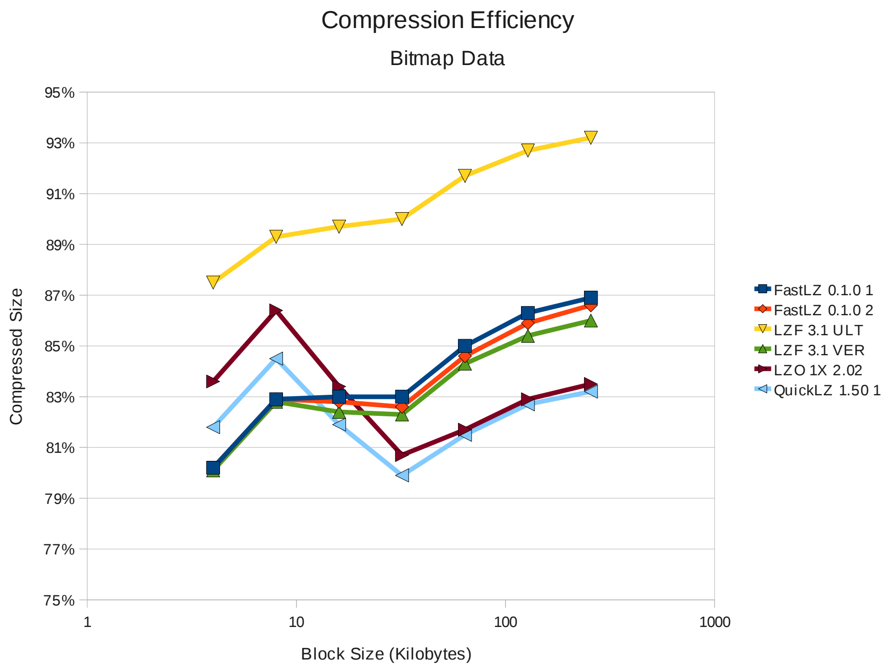


Figure 3.13: Bitmap data compressed file size by compression block size.

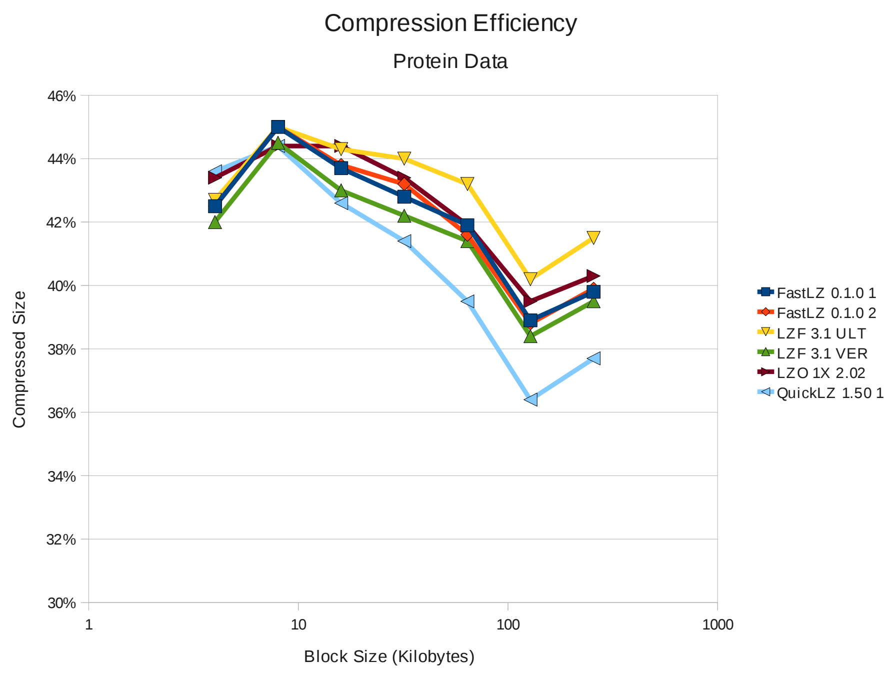


Figure 3.14: Protein data compressed file size by compression block size.

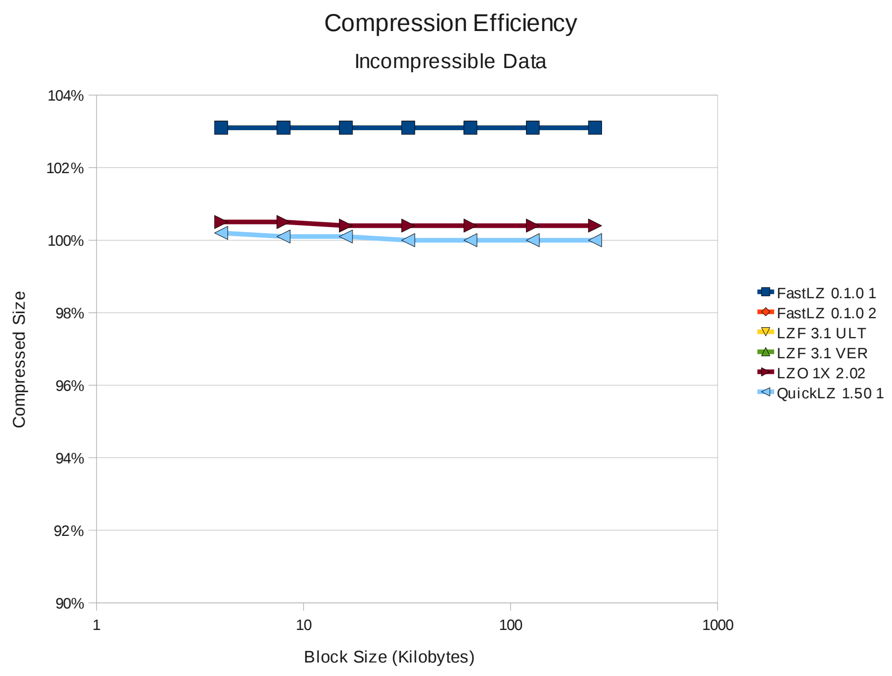


Figure 3.15: Incompressible data compressed file size by compression block size.

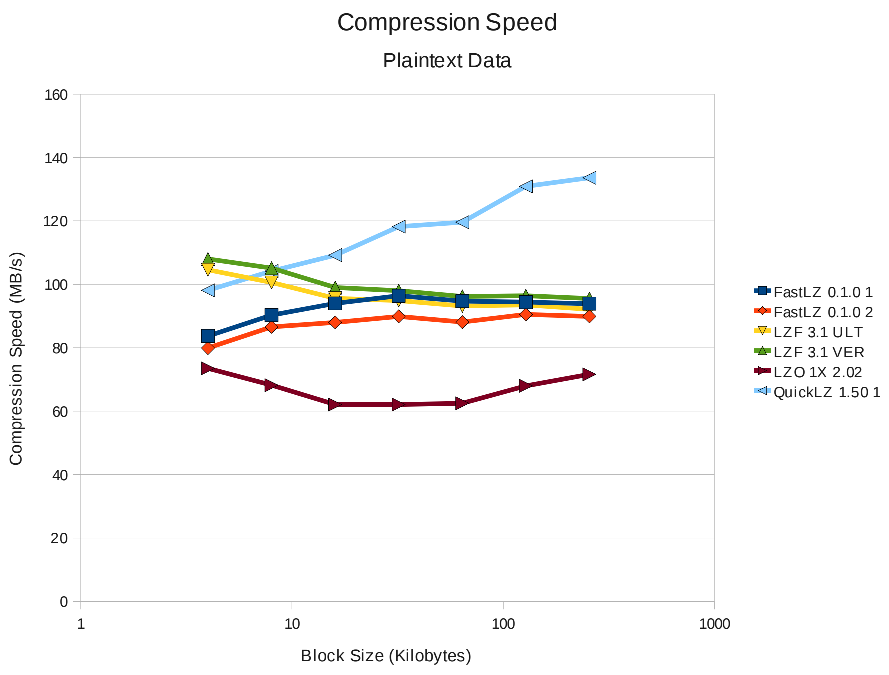


Figure 3.16: Plaintext data compression speed in MB/s by compression block size.

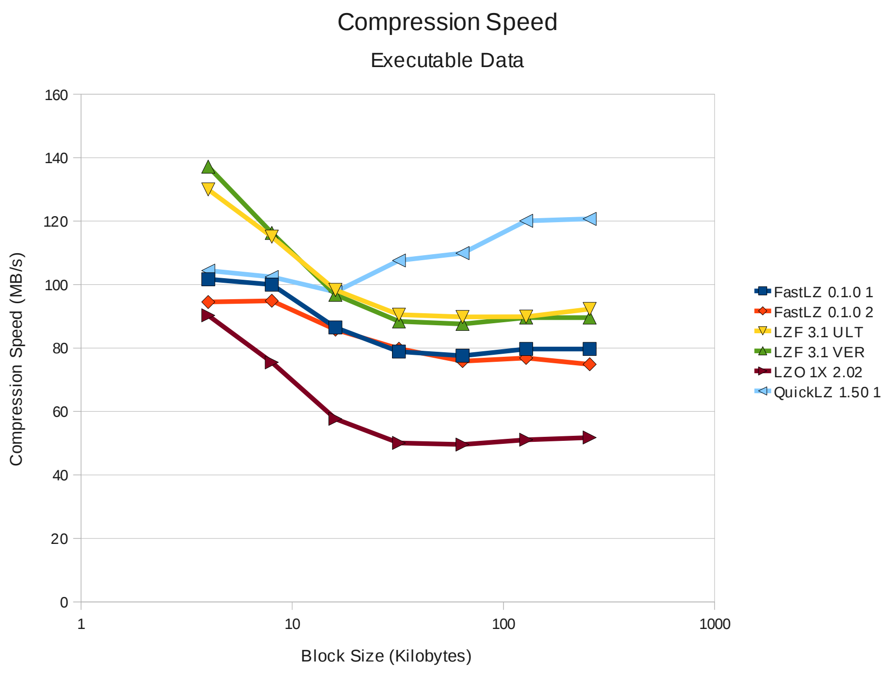


Figure 3.17: Executable data compression speed in MB/s by compression block size.

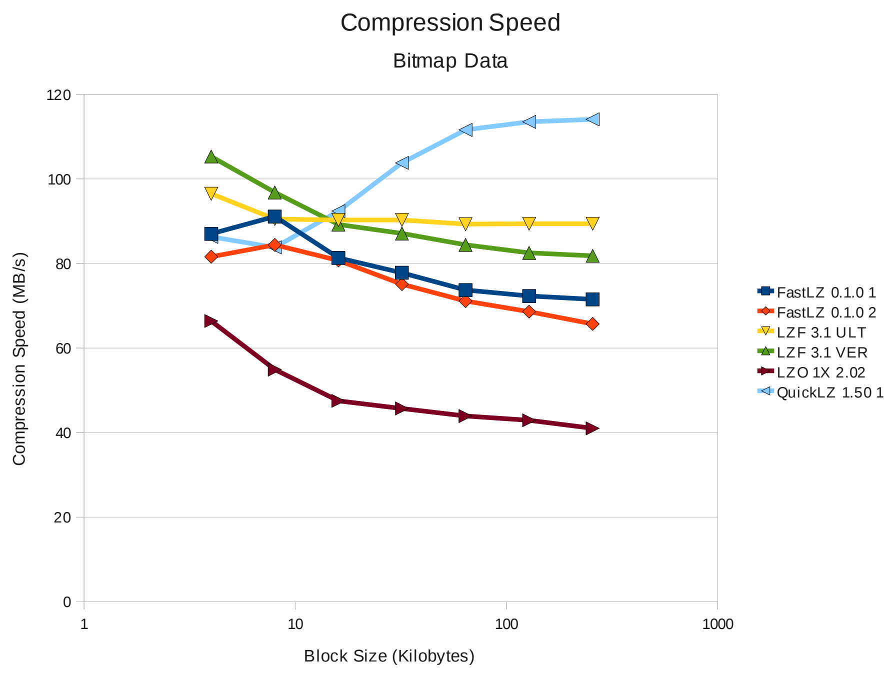


Figure 3.18: Bitmap data compression speed in MB/s by compression block size.

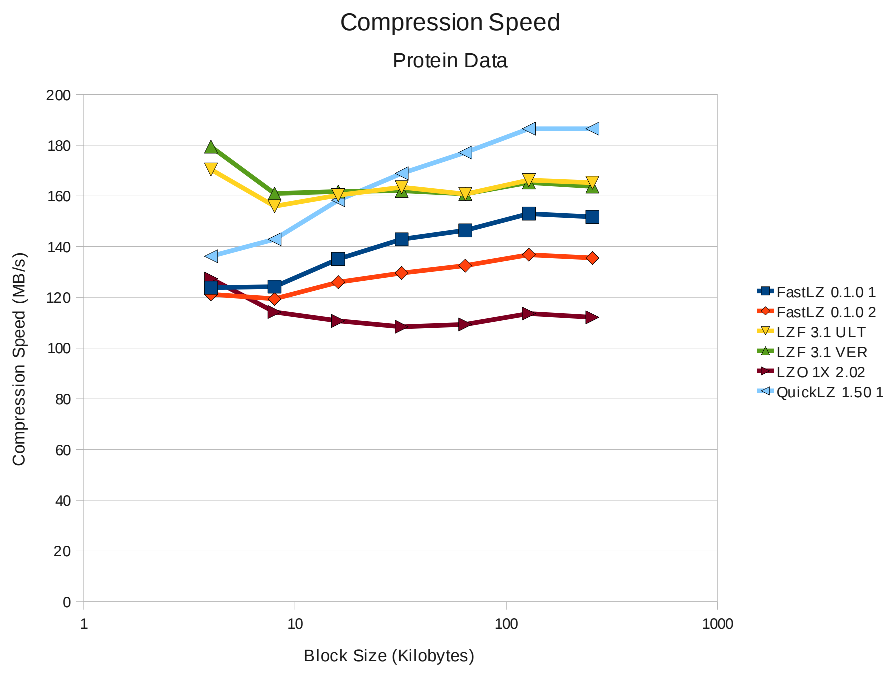


Figure 3.19: Protein data compression speed in MB/s by compression block size.

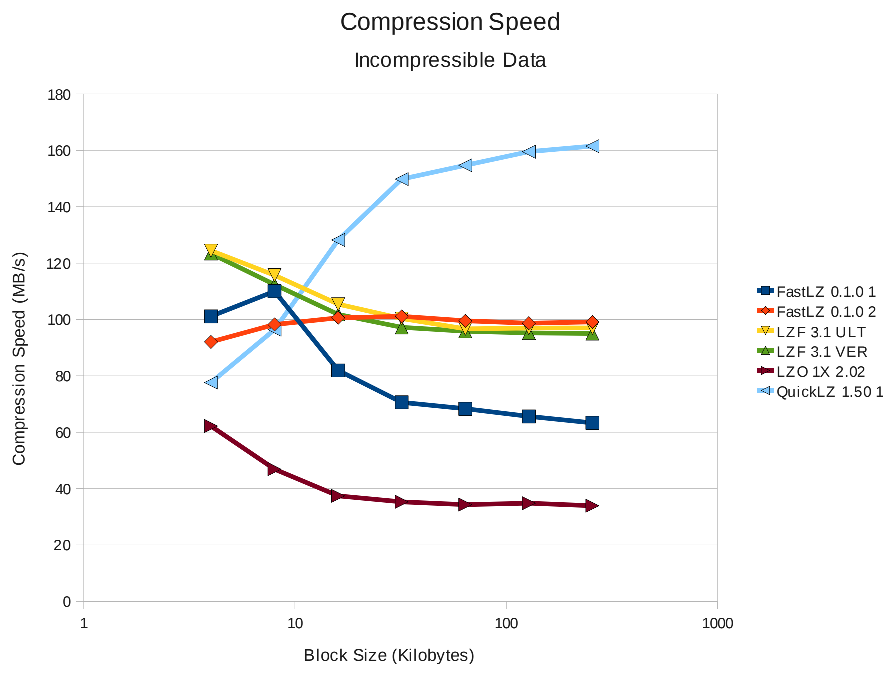


Figure 3.20: Incompressible data compression speed in MB/s by compression block size.

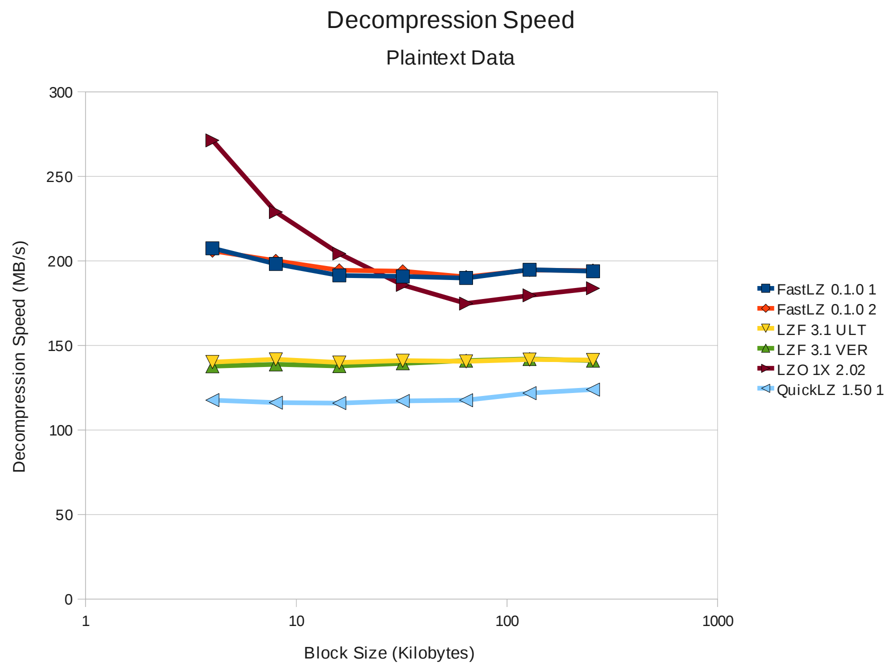


Figure 3.21: Plaintext data decompression speed in MB/s by compression block size.

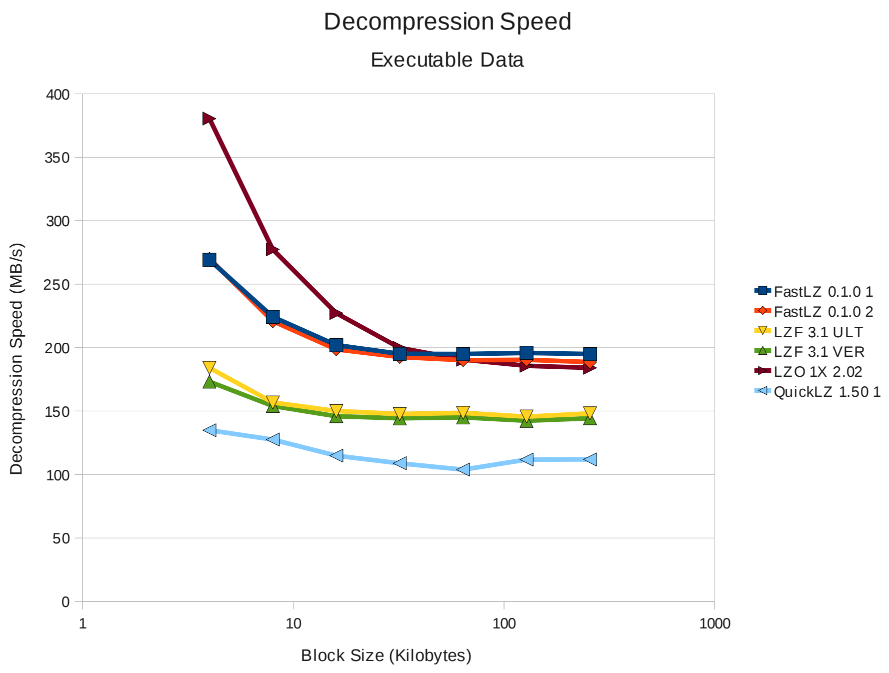


Figure 3.22: Executable data decompression speed in MB/s by compression block size.

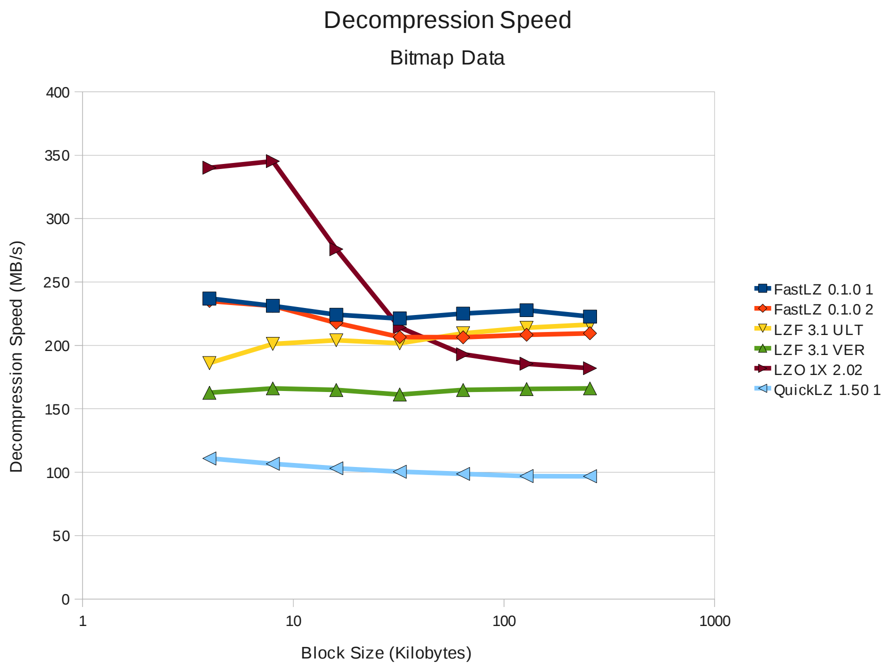


Figure 3.23: Bitmap data decompression speed in MB/s by compression block size.

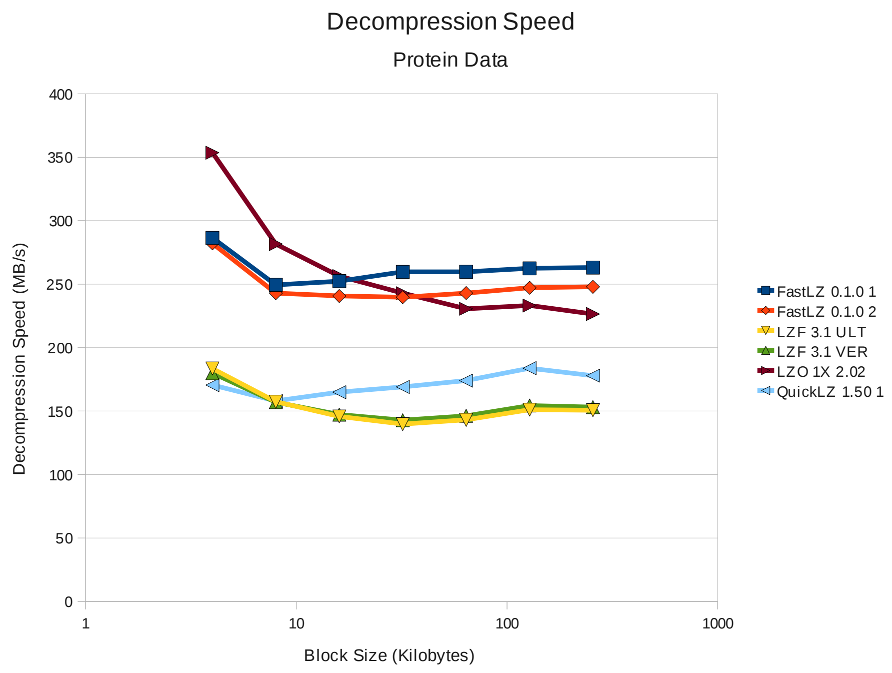


Figure 3.24: Protein data decompression speed in MB/s by compression block size.

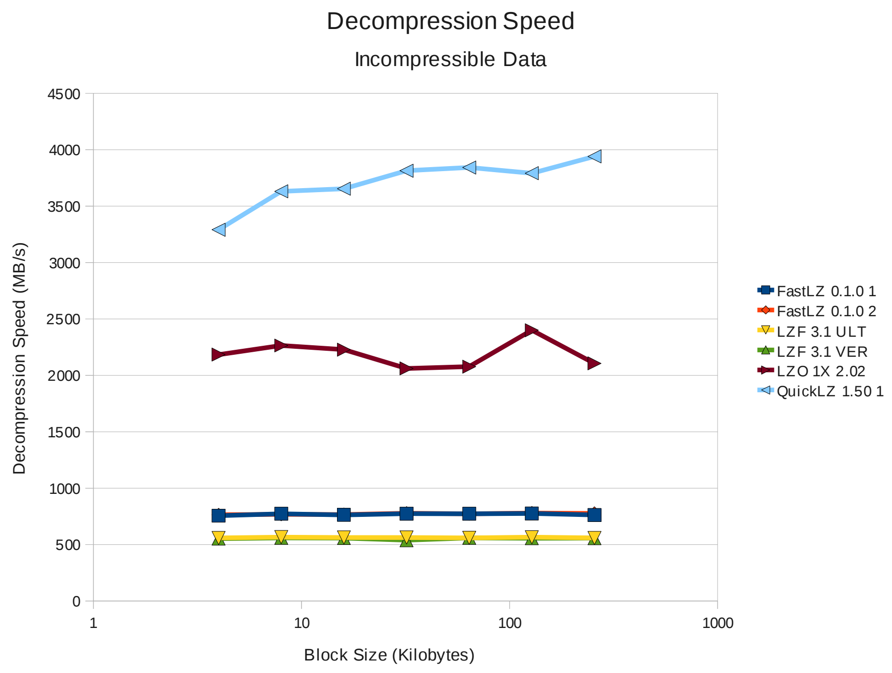


Figure 3.25: Incompressible data decompression speed in MB/s by compression block size.

spinning disks. Trove allows for various implementations for actual file management; currently the only implementation is *database plus files* (DBPF). Within the Trove DBPF implementation there are several modules than can be used for testing. The null-aio module does not actually do any disk I/O at all. This module essentially emulates infinitely fast disks for data storage while still maintaining full file system metadata. The alt-aio module is the default module for data storage and uses a threaded implementation to do asynchronous writes to disk. The alt-aio module was selected for testing transparent compression in OrangeFS.

The modifications to the alt-aio module had to take into account any activity on the file system without corrupting data. Incoming writes can be of any size up to a single stripe size in OrangeFS. The default for this strip size is 64 KB. For a write that is equal to the strip size the data can be compressed and written to disk without any further work. For a write that is not equal to the strip size there are two cases that must be addressed: a write without prior data on disk and a write with prior data on disk. The former case can be handled the same way a write of strip size is handled. The latter requires the data on disk to be decompressed, merged with the new data, and compressed then written back to disk. Because all incoming data in a single write request is written within a single strip all writes are aligned to the nearest strip size on disk. Read requests can be handled similarly. Any single read request can only address data within a single strip on disk. As such, reads are also performed aligned with the strip size.

There is a small amount of metadata (9 bytes per strip in OrangeFS) that must be stored for the chosen compression library to work properly. With standard data that is compressible this metadata can simply be written to the same storage as the data itself. With incompressible data or nearly incompressible data the QuickLZ algorithm reverts to storage-only mode where it no longer attempts to compress the current chunk and stores it uncompressed to maximize speed and minimize storage size. In this case there is no room on in the current strip for the compression metadata and it must be stored elsewhere. OrangeFS uses the *Berkeley Database* (BerkeleyDB) to store file system metadata. This construct is in place even on nodes that only contain data files because it allows for fast look-ups of the underlying file stream data. Because BerkeleyDB is already in place on every node it is the ideal place to store the compression metadata for both compressible and incompressible data for both speed and consistency. The full implementation with compression metadata stored in the BerkeleyDB is not currently complete though a basic form necessary for testing has been implemented in which the metadata is stored within the current data strip on disk. This basic implementation has limitations: incompressible data can overflow into the next data strip on disk and only transfers with a block size up to the default strip size are supported.

3.3 Server-side Caching in OrangeFS with SSDs

Server-side caching in OrangeFS was tested on the same cluster as defined in Section 3.1. To approximate the performance of a server-side cache in OrangeFS the file system was configured to store all data and metadata on the local Intel X-25E SSDs. This configuration provides an upper-bound for the performance improvements possible from integrating SSDs as a read and write cache in a parallel file system like OrangeFS. Actual implementation of a server-side cache in OrangeFS is beyond the scope of this research.

3.4 Summary of Methods

To test the feasibility of integrating SSDs as a metadata storage device in OrangeFS we have quantified the worst-case performance for both HDDs and SSDs on a relatively modern system. The performance differences were drastic enough to have warranted integration into OrangeFS. SSD metadata storage was completely implemented into the file system and is intended to be production ready.

To test the feasibility of transparent compression in OrangeFS we have implemented a basic mechanism to transparently compress and decompress data on the file system. The compression algorithm used was chosen for its high compression bandwidth and good compression efficiency. This implementation is not complete but has enough functionality to determine the worthiness of transparent compression in parallel file systems.

To test the feasibility of server-side caching using SSDs in a parallel file system we have configured OrangeFS to store all of its metadata and data on SSDs. This configuration will allow testing of the potential performance of a server-side cache without requiring a complete implementation of a cache itself.

Chapter 4

Results

This chapter will detail the experiments performed on the modified OrangeFS file system to verify performance and feasibility.

4.1 SSD Metadata Results

In order to verify the performance and scalability of the modifications described in Section 3.1 we performed a series of benchmarks to demonstrate and verify the performance difference that integration of SSDs on metadata servers can offer in OrangeFS. Our goal is to show that even with the limitations of the test platform detailed in 3.1 that the metadata performance of the file system is greatly improved under many load cases. To show the improvements in performance two separate categories of tests were run: metadata intensive tests and bandwidth intensive tests. The former category is limited nearly entirely by the ability of the file system to perform metadata operations and is intended to highlight the differences between storing metadata on HDDs versus storing it on SSDs. The second set of tests which measure sustained bandwidth to and from the file system are intended to show that while metadata performance does not drastically affect bandwidth, improved metadata performance can increase bandwidth at small access sizes. These two sets of tests combined will highlight the two extremes of performance measurements: metadata constrained performance and bandwidth constrained performance.

The defaults for the OrangeFS file system were left intact unless otherwise noted. The default configuration stripes data across all available servers. The default stripe size is 64 kilobytes. Directories themselves are not distributed but individual directories can be located on different metadata servers. OrangeFS version 2.8.4 was used for final testing and verification. All servers performed as both metadata and data servers and were set to sync metadata and data to disk. Seven OrangeFS servers were used for every test. Several configurations of OrangeFS were tested to demonstrate performance limited by the data disks as well as performance limited by the network and metadata servers:

Configuration 1: OrangeFS with no modifications.

- All metadata is stored on the local HDDs.
- All binary user data streams are stored on the local HDDs.

Configuration 2: OrangeFS with no modifications using the null-aio module for storage.

- All metadata is stored on the local HDDs.
- No user data is stored on disk. Reads and writes complete immediately. Reads return zero, writes truncate files on disk only (no data is transferred).

Configuration 3: OrangeFS with SSD metadata storage.

- All metadata is stored on the local SSDs.
- All binary user data streams are stored on the local HDDs.

Configuration 4: OrangeFS with SSD metadata storage using the null-aio module for storage.

- All metadata is stored on the local SSDs.
- No user data is stored on disk. Reads and writes complete immediately. Reads return zero, writes truncate files on disk only (no data is transferred).

The cluster listed in Section 3.1 was configured the same for the following tests with the exception of network hardware. In addition to the gigabit Ethernet previously configured and tested on an Infiniband interface was configured. Each node contains an MT25208 InfiniHost III Ex connected via an 8X PCI-Express port. These Infiniband cards are 4X SDR *Host Channel Adapters* (HCAs) that connect to a local Infiniband switch and each has dual ports. Only a single port was used for testing giving each node 10 gigabit per second connectivity to the Infiniband switch. The switch used was an InfinIO 9024 with a 480 gigabit per second backplane which should offer near-ideal network connectivity for testing.

Due to time constraints the Infiniband interface was configured to run *Internet Protocol over Infiniband* (IPoIB). IPoIB does not have the *remote direct memory access* (RDMA) engine that the Infiniband protocol natively supports. This does restrict performance because all network activity to and from the OrangeFS servers must go through the kernel TCP/IP stack instead of bypassing it with the native Infiniband protocol. Even with these restrictions the Infiniband set up to run IPoIB should provide much better results than the standard gigabit network due to its lower latency and much higher bandwidth. Both configurations are compared for most tests to show the effects of the lower latency on metadata operations and the potential performance due to increased bandwidth.

The cluster uses MPICH 2 version 1.2.1p1 for MPI communication. MPICH was configured to use the gigabit Ethernet interface for communication to avoid congesting the Infiniband network when communicating. This configuration may expose performance problems when both file I/O and MPI communication are performed on the same network due to network congestion.

OrangeFS includes in its distribution a number of test programs intended to measure various performance metrics. The test program mpi_io_test is used to measure the aggregate read and write bandwidth available to any number of clients through the standard MPI file I/O calls. By default all operations it performs are independent although it offers the ability to test collective I/O calls as well. Independent file writes were chosen to avoid limiting performance because of the latency inherent in collective write operations. Each client process calls MPI_File_write to write to a single contiguous file that spans all OrangeFS servers. These writes do not overlap in any way. The total amount of data written by each client was set to a constant 1024 megabytes regardless of transfer size. The amount of data transferred for each call of MPI_File_write was varied from 4096 bytes up to 1024 megabytes to show performance at various access sizes. This test is intended to show the maximum possible bandwidth achievable on any given configuration at a given transfer size. The OrangeFS file system was set to sync all data to disk when writing to highlight the performance differences between HDDs and SSDs without caching effects.

The second test program used is also from the standard OrangeFS distribution. The mpi_md_test is intended to be used to measure file metadata performance when creating, opening, and resizing files through the standard MPI I/O interface using the MPI_File_open and MPI_File_set_size functions. Initial performance tests showed great variation between initial and subsequent runs of the test. After analyzing the source code it became obvious that the benchmark did not clean up the files it created after running the open file benchmark. Because the benchmark left the files on disk the second run of the program measured file open performance instead of file create performance. Due to this mpi_md_test was modified to provide the additional functionality of measuring file delete performance. Because file deletion is not a collective operation in MPI it was possible to split all file deletes up equally between the number of MPI client processes. Splitting the delete operations among all available processes should help show the potential scalability of delete operations of any given configuration and allows for multiple runs without completely wiping out the file system in between each run.

Before any benchmarks were run with mpi_md_test the file system was subjected to a single run of the benchmark to create 200,000 files and another run to delete those files. This was done to allow the file system to allocate and populate any structures necessary to avoid creating them on the fly during benchmarking. All files are created in a single directory on the file system which in OrangeFS means all metadata operations are targeted to a single server. These tests are intended to provide insight into the worst possible performance for metadata operations for any given configuration. All performance measurements are based on collective MPI I/O operations unless otherwise noted. All results are averages of the performance obtained when running mpi_md_test with 25k, 50k, 100k, and 200k files per test. Performance did not vary significantly between test sizes.

Average File Creation Speed (Creates / Second)			
Clients	OrangeFS Config 1	OrangeFS Config 3	Percent Performance
1	406.39	625.26	153.85%
2	342.58	584.57	170.63%
3	305.81	548.95	179.51%
4	311.59	555.25	178.19%
5	310.43	533.33	171.80%
6	310.49	530.60	170.89%
7	305.94	515.02	168.34%
8	306.50	536.96	175.12%

Table 4.1: Collective file creation speed between configurations over gigabit Ethernet.

4.1.1 File Creation Performance

As can be seen in Table 4.1 and Table 4.2 the addition of SSDs to metadata servers increased performance significantly. This result is not unexpected with the significant increase in sustained and random read/write bandwidth available through the use of SSDs.

File creation times were cut roughly in half throughout all tests with the addition of SSDs to metadata servers. Performance was not measurably better across all cases with the Infiniband IPoIB configuration versus the gigabit Ethernet configuration. Configuration three gained more file creation performance through the lower latency network while configuration one performed roughly the same as with the gigabit Ethernet communication network. This suggests that the file creation performance of OrangeFS is relatively disk limited with standard HDDs and somewhat more network-bound with SSDs.

4.1.2 File Open Performance

File open times were not affected much by the change from HDD based metadata to SSD based metadata. This is primarily due to the effects of caching within

Average File Creation Speed (Creates / Second)			
Clients	OrangeFS Config 1	OrangeFS Config 3	Percent Performance
1	332.01	668.55	207.39%
2	319.41	636.81	199.36%
3	304.85	603.50	197.97%
4	309.57	609.76	196.97%
5	306.87	582.98	189.97%
6	304.32	584.45	192.05%
7	306.84	562.54	183.33%
8	298.42	584.91	196.00%

Table 4.2: Collective file creation speed between configurations over Infiniband IPoIB.

Average File Open Speed (Opens / Second)			
Clients	OrangeFS Config 1	OrangeFS Config 3	Percent Performance
1	2164.5	2177.07	100.58%
2	1698.75	1705.51	100.39%
3	1419.11	1421.13	100.14%
4	1427.21	1440.92	100.96%
5	1315.79	1319.84	100.31%
6	1251.56	1250.52	99.92%
7	1144.16	1143.73	99.96%
8	1195.22	1195.7	100.04%

Table 4.3: Collective file open speed between configurations over gigabit Ethernet.

the CentOS operating system on the server nodes. As seen in Table 4.3 and Table 4.4 file open performance is nearly entirely limited by network latency with the only real difference in performance coming from the change between gigabit Ethernet and Infiniband IPoIB.

4.1.3 File Resize Performance

File resize performance is also very limited by disk speed as shown in Table 4.5 and Table 4.6. The drop in network latency when switching from gigabit Ethernet to Infiniband IPoIB affects the results for both configurations nearly equally with configuration three sustaining aproximately 500-600% more resize operations per second

Average File Open Speed (Opens / Second)			
Clients	OrangeFS Config 1	OrangeFS Config 3	Percent Performance
1	2814.26	2816.90	100.09%
2	2060.44	2059.03	99.93%
3	1655.63	1654.72	99.94%
4	1675.98	1695.87	101.18%
5	1469.87	1474.20	100.29%
6	1446.48	1449.98	100.24%
7	1306.05	1308.33	100.17%
8	1374.26	1369.24	99.63%

Table 4.4: Collective file open speed between configurations over Infiniband IPoIB.

Average File Resize Speed (Resizes / Second)			
Clients	OrangeFS Config 1	OrangeFS Config 3	Percent Performance
1	269.13	1539.25	571.93%
2	263.67	1521.3	576.98%
3	323.87	1470.59	454.07%
4	257.71	1488.1	577.43%
5	262.86	1564.13	595.05%
6	268.12	1577.29	588.28%
7	337.76	1559.25	461.64%
8	277.24	1587.3	572.54%

Table 4.5: Collective file resize speed between configurations over gigabit Ethernet.

over configuration one.

4.1.4 File Deletion Performance

File deletion times were drastically affected by the underlying disk speed. As shown in Table 4.7 and Table 4.8 file deletion is mostly dependent upon disk speed. Switching from gigabit Ethernet to the Infiniband IPoIB communication network with metadata stored on SSDs increased performance by 10-30% depending on the number of clients. Switching from gigabit Ethernet to the Infiniband IPoIB configuration with metadata stored on HDDs led to very little change in performance.

Delete performance in particular is an interesting case because of the trends

Average File Resize Speed (Resizes / Second)			
Clients	OrangeFS Config 1	OrangeFS Config 3	Percent Performance
1	371.49	1689.80	454.87%
2	302.76	1822.34	601.90%
3	262.34	1666.89	635.38%
4	296.70	1613.86	543.94%
5	293.66	1660.18	565.33%
6	350.36	1805.01	515.19%
7	296.56	1670.03	563.13%
8	306.15	1843.81	602.25%

Table 4.6: Collective file resize speed between configurations over Infiniband IPoIB.

shown with metadata stored on HDDs. It appears that deletion performance scales relatively well with more clients until the HDDs get overwhelmed with requests (which causes them to spend a lot of time seeking, essentially a random write workload). This trend is shown in Figure 4.4. Random write performance on HDDs, as outlined in Section 3.1, is particularly poor when compared to HDD sustained write performance. It is likely that deletion performance would continue to degrade with an increasing number of clients accessing the file system. The floor for performance would be roughly the same as the HDDs ability to commit random writes to disk: approximately 200-300 operations per second per server accessed. Based on the nature of SSDs and their strong random write performance the opposite phenomenon is observed with deletion performance scaling nearly linearly with increasing client access. The ceiling for write performance with an exceptionally low latency network or with a very large number of clients would be approximately 5000 operations per second per server accessed.

4.1.5 Overall Metadata Results

The average results for all metadata tests are shown in Figure 4.1 through Figure 4.4. Performance for file create, file resize, and file delete all increased 200%-

Average File Delete Speed (Deletes / Second)			
Clients	OrangeFS Config 1	OrangeFS Config 3	Percent Performance
1	324.43	577.92	178.13%
2	524.61	1060.84	202.21%
3	608.77	1456.95	239.33%
4	655.98	1774.71	270.54%
5	619.56	1975.79	318.90%
6	561.80	2184.37	388.82%
7	495.96	2305.16	464.79%
8	354.34	2422.35	683.61%

Table 4.7: Distributed file deletion speed between configurations over gigabit Ethernet.

Average File Delete Speed (Deletes / Second)			
Clients	OrangeFS Config 1	OrangeFS Config 3	Percent Performance
1	321.71	657.05	204.24%
2	550.17	1199.39	218.00%
3	625.38	1615.64	258.34%
4	603.30	1875.83	310.93%
5	600.09	2080.43	346.68%
6	588.77	2270.85	385.69%
7	519.89	2357.01	453.37%
8	366.15	2523.77	689.26%

Table 4.8: Distributed file deletion speed between configurations over Infiniband IPoIB.

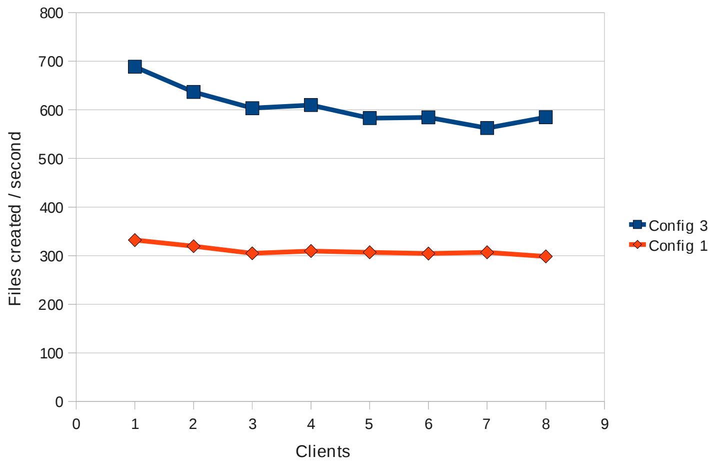


Figure 4.1: Average file creation speed between configurations over Infiniband IPoIB.

500% with delete operations increasingly performing better under heavier loads with more clients. Network latency and bandwidth affected results slightly and tended to increase performance more with configuration three than configuration one. Using null-aio with configurations two and four did not noticeably affect results in any way. File open performance is limited entirely by network latency as all configurations on the same network layer performed identically.

4.1.6 Aggregate I/O Performance

Aggregate I/O bandwidth did not vary drastically when storing metadata on SSDs versus storing metadata on HDDs for most block sizes. Read performance with mpi_io_test for all block sizes did not reliably vary by any significant amount. This is predominantly due to the effects of the CentOS buffer cache which completes read requests at speeds far in excess of any SSD or HDD. Write bandwidth did show significant improvement when storing metadata on SSDs when dealing with relatively small accesses. For block sizes below 128 kilobytes write speed for configuration three performed up to 105% better in terms of write bandwidth. Figure 4.5 shows OrangeFS

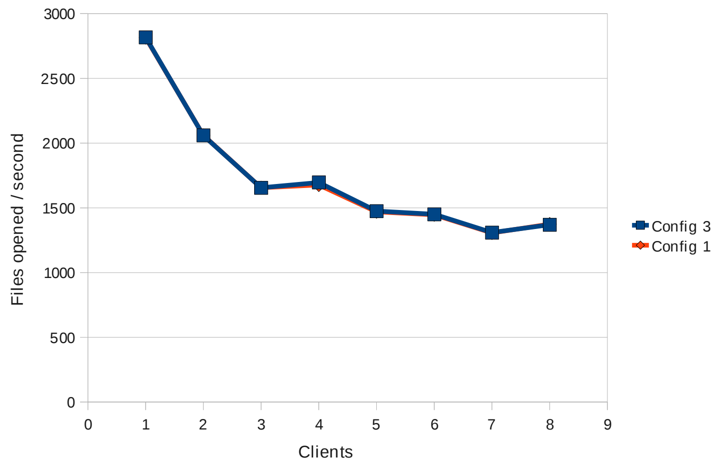


Figure 4.2: Average file open speed between configurations over Infiniband IPoIB.

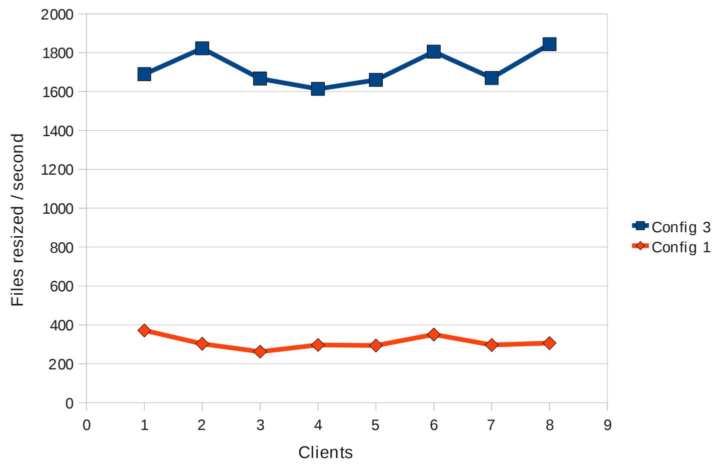


Figure 4.3: Average file resize speed between configurations over Infiniband IPoIB.

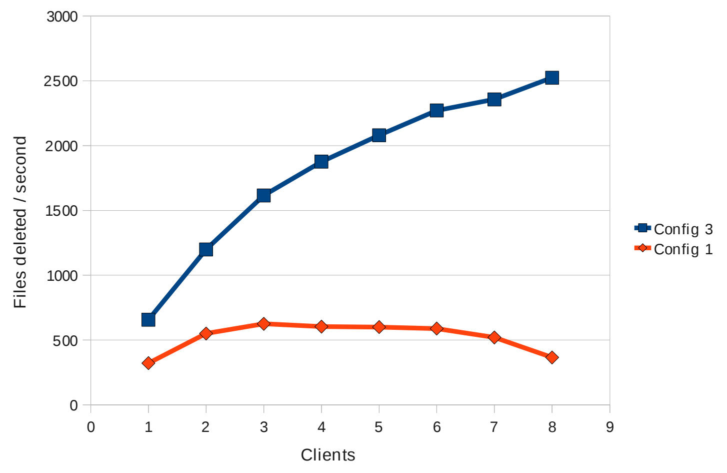


Figure 4.4: Average file deletion speed between configurations over Infiniband IPoIB.

write performance with mpi_io_test for all block sizes tested.

To further test the improvements that could be gained configuration two and configuration four were deployed on the cluster and tested with mpi_io_test. Read performance was again nearly identical between configurations regardless of block size. Write bandwidth was again significantly better when storing metadata on SSDs at block sizes below 128 kilobytes. Write bandwidth increased by up to 260% with these smaller block sizes and performed nearly the same at larger block sizes. The largest performance difference when storing metadata on SSDs was seen at 64 kilobytes for both configuration three and configuration four. Figure 4.6 shows OrangeFS read and write performance with mpi_io_test for all block sizes tested.

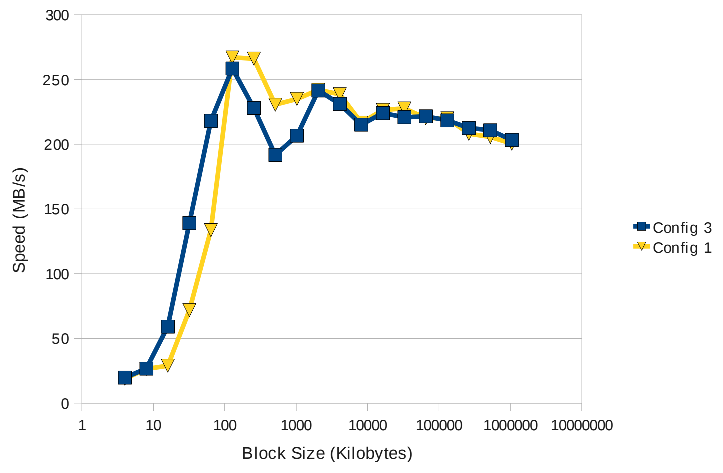


Figure 4.5: Aggregate write bandwidth comparison over Infiniband IPoIB.

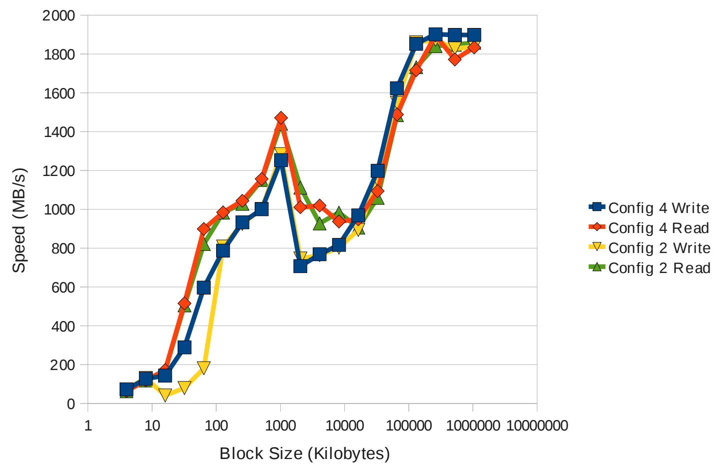


Figure 4.6: Aggregate read/write bandwidth comparison over Infiniband IPoIB with null-aio.

4.2 Compression Results

In order to show that transparent compression does not significantly impact performance for streaming workloads we have tested OrangeFS with mpi_io_test under various configurations. Streaming performance for larger block sizes with transparent compression enabled is expected to be relatively similar to streaming performance without transparent compression. Tests were also performed with the file system configured to only compress the data in memory without writing it to disk. This should expose the maximum performance possible with transparent compression as it should be entirely CPU-bound. Tests were also performed on a configuration using SSDs as the data storage devices to highlight the potential gains when compressing data to SSDs versus storing data without compression on HDDs.

All tests performed in this section were run on the same system and communication networks described in Section 4.1. All OrangeFS configuration parameters used in Section 4.1 were used in these benchmarks and disk syncing was again enabled for writes. The following OrangeFS configurations were used for all tests:

Configuration 1: OrangeFS with SSD metadata storage.

- All metadata is stored on the local SSDs.
- All binary user data streams are stored on the local HDDs.

Configuration 2: OrangeFS with SSD metadata storage and transparent compression.

- All metadata is stored on the local SSDs.
- All binary user data streams are stored on the local HDDs in compressed form.

Configuration 3: OrangeFS with SSD metadata storage and transparent compression using the null-aio module for storage.

- All metadata is stored on the local SSDs.
- No user data is stored on disk. Reads and writes complete immediately. Reads return zero, writes truncate files on disk only (no data is transferred). Compression is performed in memory on user data to test compression performance.

Configuration 4: OrangeFS with SSD metadata storage and transparent compression.

- All metadata is stored on the local SSDs.
- All binary user data streams are stored on the local SSDs in compressed form.

In order to test the performance of transparent compression in OrangeFS mpi_io_test was again used. Originally this benchmark wrote blocks of data to the parallel file system without regard to what it was. In most cases this meant sending zeroed data which would not be appropriate for testing compression speed because it is highly compressible and is not a typical workload. To show performance with the representative data presented in Section 3.2 the benchmark mpi_io_test was modified. The modifications included the option to read a file from disk for data to send out to each I/O server. This file I/O is done before any benchmarking begins to avoid skewing results. The same block is sent for each iteration of the benchmark but because the compression algorithm used is stateless between invocations this does not affect compression speed or efficiency.

Streaming write performance with 32 kilobyte, 64 kilobyte, and 256 kilobyte blocks of compressed data on disk is shown in Figure 4.7, Figure 4.8, and Figure 4.9. All representative data was tested for all block sizes and extremely compressible data was also tested with the 64 kilobyte block size. In general the 64 kilobyte compression block setting performed the best for the largest variety of representative data when data was being stored to standard HDDs. At the larger 256 kilobyte block size the time to perform the compression outweighed the decreased disk bandwidth used by a large amount. At the smallest 32 kilobyte compression block performance was reasonable compared to the non-compressed bandwidth but for most data types at the largest block size performance was significantly slower. The 64 kilobyte compression block provided the most consistent performance over the range of block sizes for all data types. Figure 4.8 also highlights the speed that can be obtained when transferring highly compressible data to the file system; compression of zeroed data is much faster than disk at every block size by a significant amount. This performance could be seen in practice when checkpointing with data sets that are very sparsely laid out in memory.

Streaming write performance with 32 kilobyte, 64 kilobyte, and 256 kilobyte blocks of compressed data using the null-aio module is shown in Figure 4.10, Figure 4.11, and Figure 4.12. It is apparent that smaller sizes for the compression blocks lead to higher maximum compression rates. With 32 kilobyte blocks compression speed is well above disk speed at all block sizes. With 64 kilobytes block sizes there is a small reduction in overall compression speed but the ability to perform compression on 64 kilobyte blocks makes up for this with drastically increased compression speeds. When compressing 256 kilobytes blocks performance drops below the peak levels seen when using 64 kilobyte blocks and is much lower for smaller transfer block sizes. This is because the latency inherent in doing larger compression blocks is much higher

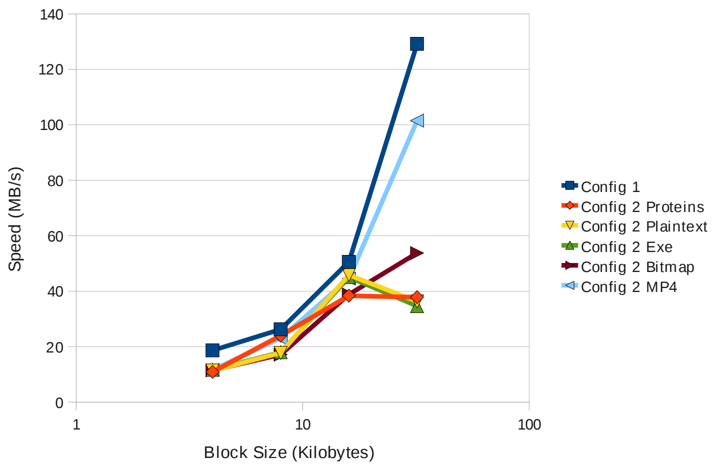


Figure 4.7: Aggregate write bandwidth comparison over Infiniband IPoIB with transparent compression and 32 KB compression blocks.

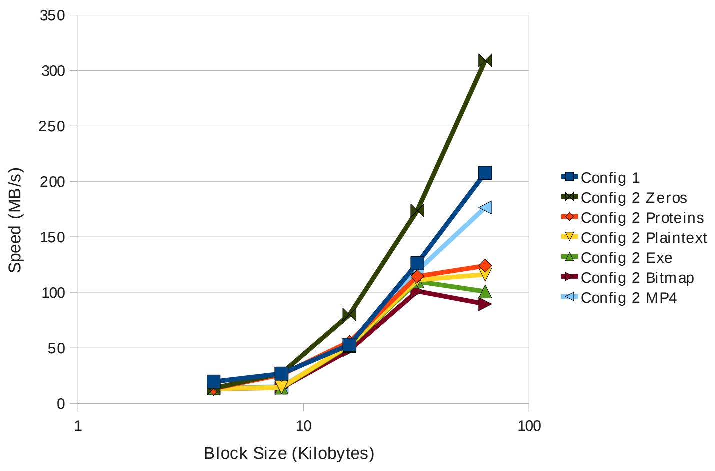


Figure 4.8: Aggregate write bandwidth comparison over Infiniband IPoIB with transparent compression and 64 KB compression blocks.

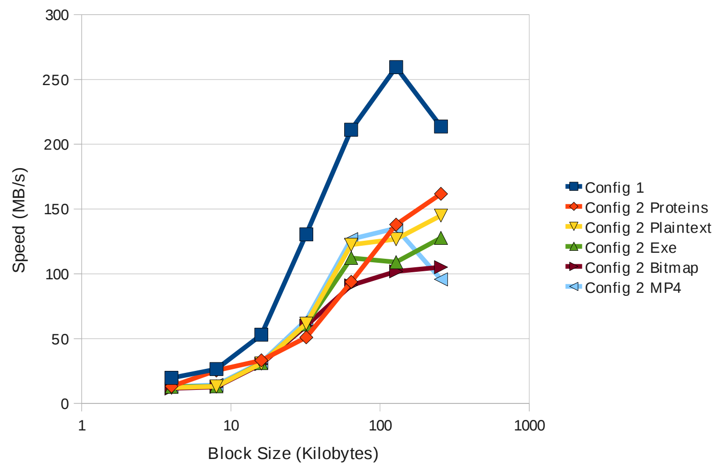


Figure 4.9: Aggregate write bandwidth comparison over Infiniband IPoIB with transparent compression and 256 KB compression blocks.

than it is for smaller compression blocks. At this large compression block size the HDD itself can store data faster than the compression algorithm can compress it in memory; this is not ideal for any situation.

Figure 4.13 shows the performance of OrangeFS with transparent compression when storing data on SSDs. This is intended to show the performance increase that can be had when compressing data transparently and storing it on SSDs to both gain capacity and performance. Even at the smallest block sizes the performance when storing compressed data on SSDs exceeds the performance of simply storing data on the local HDDs.

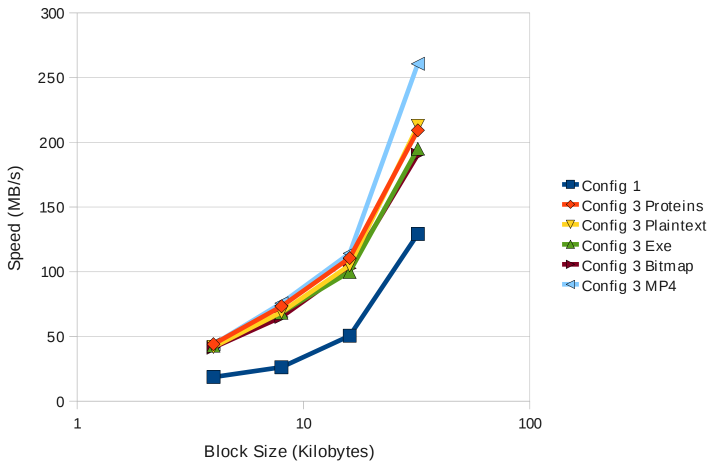


Figure 4.10: Aggregate write bandwidth comparison over Infiniband IPoIB with transparent compression and 32 KB compression blocks using null-aio.

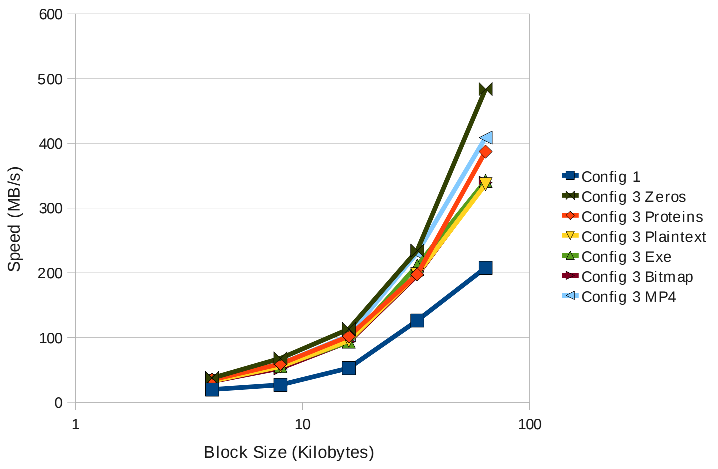


Figure 4.11: Aggregate write bandwidth comparison over Infiniband IPoIB with transparent compression and 64 KB compression blocks using null-aio.

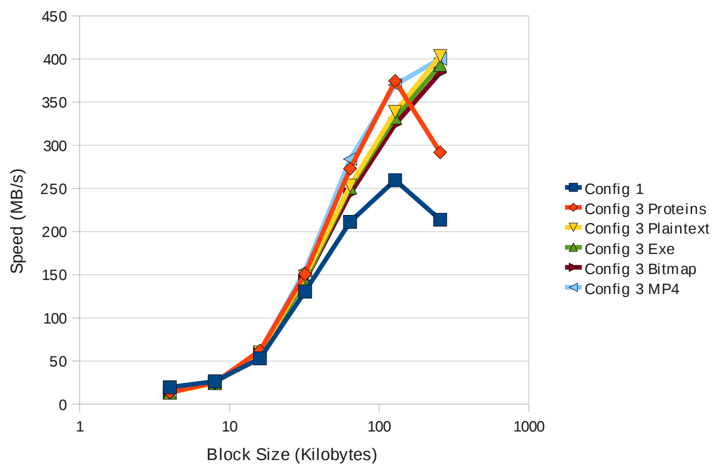


Figure 4.12: Aggregate write bandwidth comparison over Infiniband IPoIB with transparent compression and 256 KB compression blocks using null-aio.

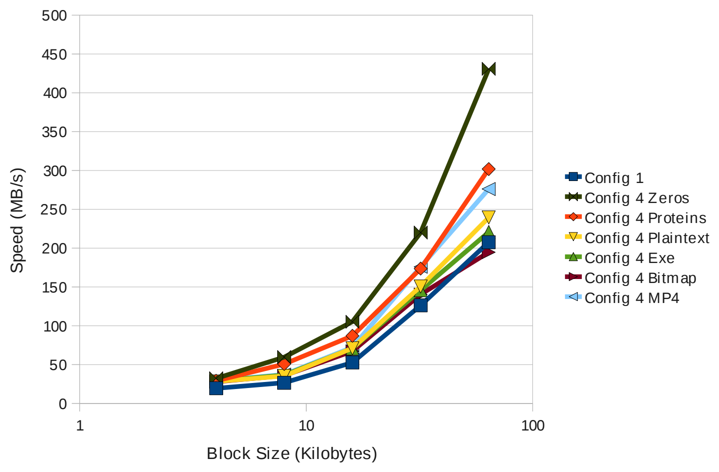


Figure 4.13: Aggregate write bandwidth comparison over Infiniband IPoIB with transparent compression and 64 KB compression blocks using SSDs to store data.

4.3 Server-side Cache Results

In order to show the performance benefits possible from a server-side SSD cache in OrangeFS we have run a series of tests designed to expose the inherent performance improvements that could be gained with an ideal SSD cache on an OrangeFS server. All tests performed in this section were run on the same system and communication networks described in Section 4.1. All OrangeFS configuration parameters used in Section 4.1 were again used in these benchmarks. The following OrangeFS configurations were used for all tests:

Configuration 1: OrangeFS with SSD metadata storage.

- All metadata is stored on the local SSDs.
- All binary user data streams are stored on the local HDDs.

Configuration 2: OrangeFS with SSD metadata storage.

- All metadata is stored on the local SSDs.
- All binary user data streams are stored on the local SSDs to simulate a perfect cache.

Initial performance tests with mpi_io_test did not reveal any significant performance differences between configuration one and configuration two. The buffer cache in CentOS was absorbing all writes and reads without hitting the local disks. To expose the real performance of the underlying disks when writing OrangeFS was set to sync all data to disk for both metadata and data after every write.

Figure 4.14 shows the potential performance improvement that could be gained if all writes to an OrangeFS file system were absorbed by the SSD cache. Write

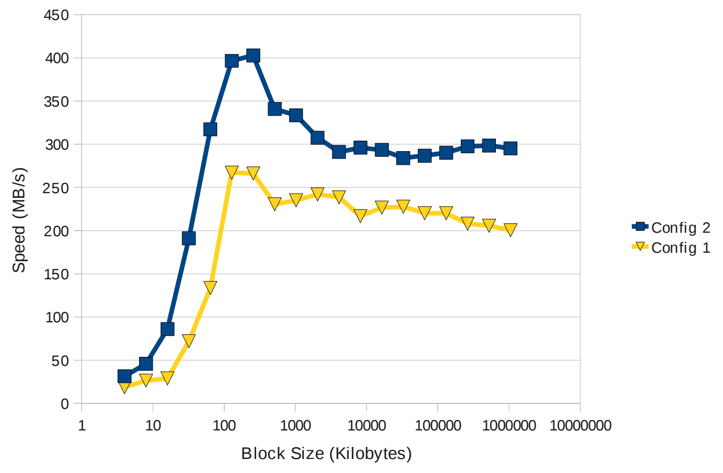


Figure 4.14: Aggregate write bandwidth comparison over Infiniband IPoIB with an ideal SSD cache.

performance is measurably better for all block sizes by an average of 46% and by over 100% for small block sizes below 128 KB.

While write performance of the drives themselves could be exposed by forcing syncs between writes, read requests were still being fulfilled out of the buffer cache. To avoid this effect the OrangeFS servers were configured to use the *directio* Trove method to force all read and write requests to bypass the operating system buffer cache. This setting is generally recommended only for use on large shared storage volumes (because of the performance loss on small RAIDs and single HDDs) but it will expose the lower-level disk behavior. Disk caching with the directio storage method is minimal and limited to the on-disk caches only as it opens files with the O_DIRECT flag. CPU consumption when reading and writing directly to disk is slightly lower because data is read and written directly from disk to the application memory space instead of passing through the buffer cache.

Figure 4.15 and Figure 4.16 show the performance for streaming write and read operations for configurations one and two. Both read and write performance of the ideal SSD cache are much faster than with data stored on HDDs. These results

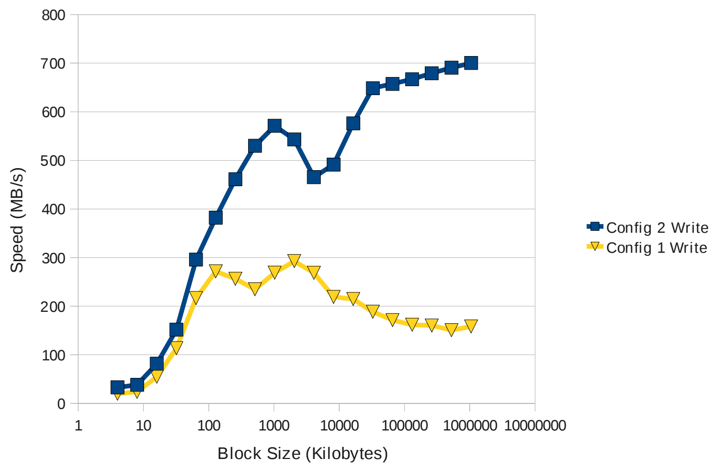


Figure 4.15: Aggregate write bandwidth comparison over Infiniband IPoIB with an ideal SSD cache using directio.

highlight the much higher performance that could be gained through the use of an SSD cache in a high performance parallel file system. Write bandwidth for configuration one when using directio is faster at small block sizes and slower at large block sizes than when using the default alt-aio module. This is likely due to implementation and overhead differences between the two algorithms. Write bandwidth for configuration two is nearly twice as high with block sizes over 256 kilobytes when using the directio module versus the alt-aio module. This improvement in performance is likely due to the decreased overhead when writing to disk using directio as data does not need to be copied around in memory from kernel to user space as it does in the alt-aio module implementation. These differences can be seen more clearly in Figure 4.17 and Figure 4.18.

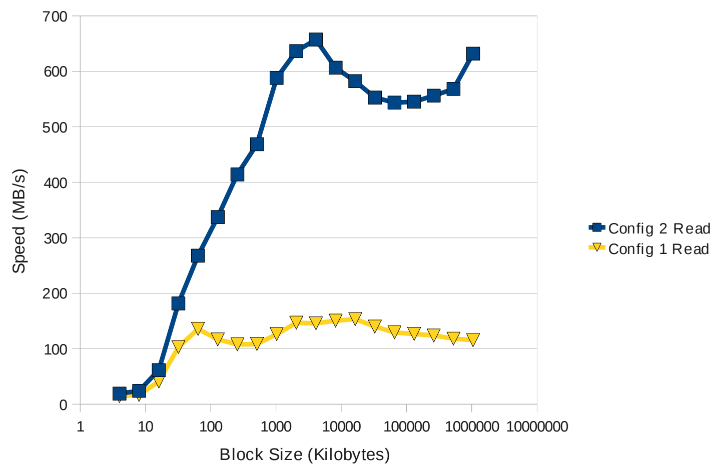


Figure 4.16: Aggregate read bandwidth comparison over Infiniband IPoIB with an ideal SSD cache using directio.

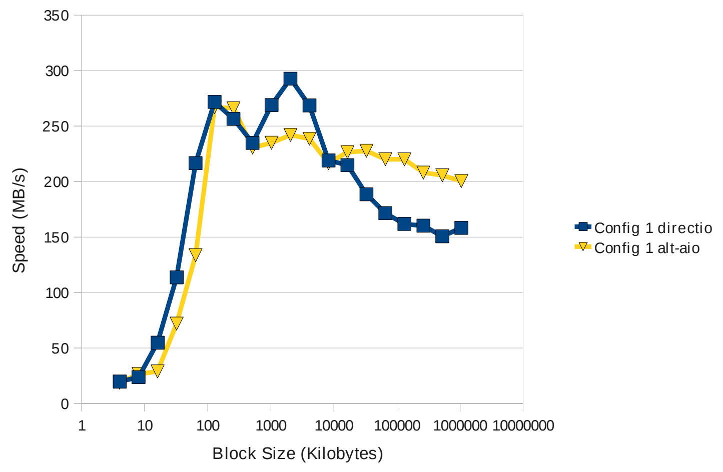


Figure 4.17: Aggregate write bandwidth comparison over Infiniband IPoIB using HDD storage between directio and alt-aio.

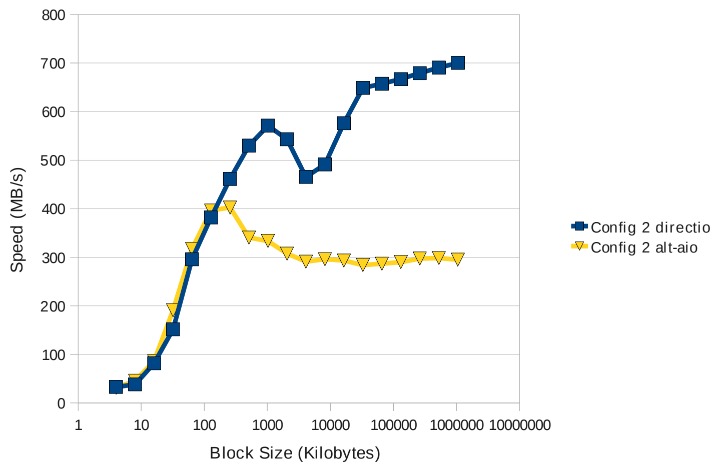


Figure 4.18: Aggregate write bandwidth comparison over Infiniband IPoIB with an ideal SSD cache between directio and alt-aio.

Chapter 5

Conclusion

We presented various methods for integrating SSDs into parallel file systems to meet the need for increased I/O bandwidth for large-scale computation. These methods offer both the ability to improve performance and to improve reliability through shortened job runtimes.

The technique used to store metadata within a parallel file system is key in overall metadata performance. We have shown that splitting metadata and data storage to separate suitable storage devices is an effective method to increase performance dramatically. Not only does this method increase performance on metadata-constrained operations but it also increases performance when dealing with workloads that utilize small block sizes. The method presented is production-ready, easy to configure, and most importantly, requires no extra effort by the end-user to take advantage of the additional performance.

The method of transparent compression within a parallel file system presented offers many interesting results. We have shown that performance improvements can be had under certain configurations but for most cases transparent compression offers the ability to store more data within a given system without increased hardware costs or large performance losses even on older processors. As processors continue to grow faster the ability to transparently compress data will improve. Any improvements in compression algorithm speed or efficiency will directly translate into higher performance for a larger variety of workloads and more usable storage on a given system. These improvements would be largely transparent to the end-user and require no modification of their programs to gain the benefits.

We also observed the performance improvements that could be taken advantage of when utilizing SSDs as a server-side cache in a parallel file system. Disk-constrained performance for both reading and writing with all workloads improved significantly with the use of an ideal cache made up of SSDs. This large improvement in bandwidth allows the file system to absorb the bursty I/O that is common with many applications. After the bursty I/O is complete the file system can migrade data to and from the SSD cache to prepare for the next set of I/O while the application continues computation. Speeding up those short bursts of reads and writes lead directly to higher overall performance for the application and require no extra effort on the part of the end-user.

5.1 Future Work

There is still much research and development to be done with regards to integrating SSDs and transparent compression into parallel file systems. Random write performance was not tested with any of the implementations discussed in this work but is critical for certain application workloads. It is very likely that further work would need to be done to improve transparent compression performance in such a scenario. Additionally, a full implementation of transparent compression using the key/value database present within the OrangeFS server would allow for more flexibility and speed. It is obvious from the results in Section 4.3 that server-side caching could provide large performance increases for many workloads and could also improve performance in random write and random read workloads based on the nature of SSDs and the performance results obtained in Section 3.1. Additional performance improvements could be seen when storing metadata on SSDs if the metadata database was tuned to be aware of the access patterns that perform well on SSDs.

Bibliography

[1] Iozone file system benchmark.
http://www.iozone.org/, 2006.

[2] Top500 supercomputing list 2010.
http://www.top500.org/list/2010/11/100, 2010.

[3] Quicklz compression library.
http://www.quicklz.com/index.php, 2011.

[4] CEPH: Designing a Cluster.
http://ceph.newdream.net/wiki/Designing_a_cluster, 2011.

[5] IEEE Standards Association. 1003.1-2008 portable operating system interface (posix).
http://standards.ieee.org/findstds/standard/1003.1-2008.html, 2010.

[6] P. H. Carns. Achieving scalability in parallel file systems. Technical report, Clemson University, 2005.

[7] Peter M. Chen, Edward K. Lee, Garth A. Gibson, Randy H. Katz, and David A. Patterson. Raid: High-performance, reliable secondary storage. *ACM COMPUTING SURVEYS*, 26:145–185, 1994.

[8] IBM SAN Volume Controller.
http://www-03.ibm.com/systems/storage/software/virtualization/svc/features.html, 2010.

[9] EMC Corporation. Dmx-4 press release.
http://www.emc.com/about/news/press/us/2008/011408-1.htm, 2008.

[10] Standard Performance Evaluation Corporation. Spec cpu2006 benchmark results.
http://http://www.spec.org/cpu2006/results/, 2010.

[11] Laurel Davis Dan Davis and Summer Allen. Educational extensions of large-scale simulations enabled by high performance computing. 2006.

[12] Cagdas Dirik and Bruce Jacob. The performance of pc solid-state disks (ssds) as a function of bandwidth, concurrency, device architecture, and system organization. In *Proceedings of the 36th annual international symposium on Computer architecture*, ISCA '09, pages 279–289, New York, NY, USA, 2009. ACM.

[13] Susan Graham and Marc Snir. Getting up to speed: The future of supercomputing. 2004.

[14] Business Systems International. Unleashing application performance with ssds and sun servers. *Sun Blueprints Online*, 2009.

[15] Taeho Kgil, David Roberts, and Trevor Mudge. International symposium on computer architecture improving nand flash based disk caches, 2008.

[16] Sang-Won Lee, Bongki Moon, Chanik Park, Jae-Myung Kim, and Sang-Woo Kim. A case for flash memory ssd in enterprise database applications. *SIGMOD '08*, June 2008.

[17] T. Makatos, Y. Klonatos, M. Marazakis, M. Flouris, and A. Bilas. Using transparent compression to improve ssd-based i/o caches. In *EuroSys*. ACM, 2010.

[18] Lawrence McIntosh and Michael Burke. Solid state drives in hpc: Reducing the i/o bottleneck. *Sun Blueprints Online*, 2009.

[19] Inc. Micron Technology. Nand flash product sheets. http://www.micron.com/products/nand_flash/, 2011.

[20] Sun Microsystems. Lustre file system: High-performance storage architecture and scalable cluster file system. *White Paper*, 2007.

[21] E L Miller and R H Katz. Input/output behavior of supercomputing applications. In *In Proceedings of Supercomputing 91*, pages 567–576, 1991.

[22] D. Narayanan, E. Thereska, A. Donnelly, S. Elnikety, and A. Rowstron. Migrating server storage to ssds: Analysis of tradeoffs. In *EuroSys*. ACM, 2009.

[23] Panasas. Pas-9 product description. http://www.panasas.com/docs/PAS9-DataSheet-PW-10-41601-v1-lores.pdf, 2010.

[24] Fabrizio Petrini, Darren J. Kerbyson, and Scott Pakin. The case of the missing supercomputer performance: Achieving optimal performance on the 8192 processors of asci q. 2003.

[25] Sandforce. Sf-1500 enterprice ssd processors. http://www.sandforce.com/userfiles/file/downloads/SandForce_1500ENT_PB_110124.pdf, 2009.

[26] Frank Schmuck and Roger Haskin. Gpfs: A shared-disk file system for large computing clusters. In *In Proceedings of the 2002 Conference on File and Storage Technologies (FAST*, pages 231–244, 2002.

[27] B. Schroeder and G. Gibson. Disk failures in the real world: What does an mttf of 1,000,000 hours mean to you? In *USENIX Conference on File and Storage Technologies (FAST)*, pages 1–16, 2007.

[28] Seagate. X15 hdd product manual.
http://www.seagate.com/www/en-us/, 2000.

[29] F Wang, Q Xin, B Hong, S A Brandt, E L Miller, D D E Long, and T T McLarty. File system workload analysis for large scale scientific computing applications. In *In Proceedings of the 21st IEEE / 12th NASA Goddard Conference on Mass Storage Systems and Technologies*, pages 139–152, 2004.

[30] Sage A. Weil, Scott A. Brandt, Ethan L. Miller, Darrell D. E. Long, and Carlos Maltzahn. Ceph: A scalable, high-performance distributed file system. In *In Proceedings of the 7th Symposium on Operating Systems Design and Implementation (OSDI)*, pages 307–320. ACM, 2006.