Computer Science Department
University of Crete

Performance analysis and scaling of networked, shared, block-level storage

Master's Thesis

Dimitrios Xinidis

November 2005
Heraklion, Greece

Performance analysis and scaling of networked, shared, block-level storage

by

Dimitrios Xinidis

Master's Thesis

Department of Computer Science
University of Crete

Abstract

iSCSI is proposed as a possible solution to building future storage systems. However, using iSCSI raises numerous questions about its implications on system performance. This lack of understanding of system I/O behavior in modern and future systems inhibits providing solutions at the architectural and system levels. First of all in this work, we try to understand the behavior of the application server (iSCSI initiator), to evaluate the overhead introduced by iSCSI compared to systems with directly-attached storage, and to provide insight about how future storage systems may be improved. We examine these questions in the context of commodity iSCSI systems that can benefit most from using iSCSI. Our analysis shows that building next generation, network-based I/O architectures, requires optimizing I/O latency, reducing network and buffer cache related processing in the host CPU, and increasing the sheer network bandwidth to account for consolidation of different types of traffic.

Based on this knowledge we provide our own solution for a scalable distributed storage system, Orchestra. Orchestra*, is a system for cluster storage virtualization that allows sharing of storage volumes at the block level by providing block locking and free block allocation services. To allow file-based applications access to* Orchestra *block devices, in this work we design and implement a* stateless, pass-through *file system,* Orchestra-*fs, that relies on* Orchestra *for locking and block-allocation support and allows multiple existing applications to use a single* Orchestra *volume.*

In this work we present the implementation of Orchestra-fs *under Linux and evaluate it over* *various setups using both single application and multiple storage nodes. We find that since most functionality is implemented at the block layer, which is well structured, there is little overhead beyond TCP/IP communication costs and existing kernel overheads for disk I/O. Finally, our results show that performance scales as the number of storage nodes grows for experiments where there is no file sharing between applications.*

Keywords: iSCSI, virtual timers, storage virtualization, distributed, stateless filesystem

Thesis Supervisor: Angelos Bilas
Title: Associate Professor

Ανάλυση της απόδοσης και κλιμάκωσης δικτυακού, διαμοιραζόμενου αποθηκευτικού χωρου σε επίπεδο *blocks*

Δημήτριος Ξυνίδης

Μεταπτυχιακή Εργασία

Τμήμα Επιστήμης Υπολογιστών
Πανεπιστήμιο Κρήτης

Περίληψη

Το πρωτόκολλο iSCSI έχει προταθεί ως μια πιθανή λύση για την κατασκευή μελλοντικών συστημάτων αποθήκευσης. Παρ'ολα αυτά, η χρησιμοποίηση του δημιουργεί διάφορες ερωτήσεις σχετικά με την επίπτωση του στην απόδοση του συστήματος. Αυτη η έλλειψη κατανόησης της συμπεριφοράς της εισόδου/εξόδου των συστημάτων αυτών παρακωλύει την παροχή λύσεων στο επίπεδο της αρχιτεκτονικής των συστημάτων αυτών. Πρωτα απ'ολα, σάυτή την δουλειά προσπαθούμε να κατανοήσουμε την συμπεριφορά του ¨πελάτη¨, να αξιολογήσουμε τις επιβαρύνσεις που προκαλεί το πρωτόκολλο αυτό σε σχέση με τα συστήματα που χρησιμοποιούν τοπικό χώρο αποθήκευσης και να προτείνουμε λύσεις, τρόπους βελτίωσης των μελλοντικών συστημάτων αποθήκευσης.

Η αναλυσή μας αποδεικνύει οτι για τη κατασκευή αρχιτεκτονικών διαδικτυακών συστημάτων αποθήκευσης επόμενης γεννιάς, προυποθέτει την βελτιστοποίηση του χρόνου ανταπόκρισης εισόδου/εξόδου, την μείωση του χρόνου που ξοδεύεται σε κομμάτια του λειτουργικού συστήματος όπως είναι αυτο του δίκτυο και της buffer cache στην μεριά του ¨πελάτη' καθως επίσης και την αύξηση της συνολικής χωριτικότητας του δικτύου για την ομαλή υποστήριξη, απομόνωση διαφορετικών τύπων κίνησης πάνω απο αυτό.

Στηριζόμενοι σε αυτη τη γνώση, προτείνουμε τη δική μας λύση για την κατασκευή ενός κλιμακώμενου, καταναμημένου αποθηκευτικού συστήματος, το οποίο ονομάζουμε Orchestra. Το Orchestra είναι ενα σύστημα εικονικοποίησης που επιτρέπει τον διαμοιρασμό χώρου αποθήκευσης στο επίπεδο του block παρέχοντας υπηρεσίες κλειδώματος και δέσμευσης αυτών των blocks. Για να επιτρέψουμε τις εφαρμογές να χρησιμοποιούν τον αποθηκευτικό χώρο του Orchestra σε επίπεδο αρχείου, σχεδιάσαμε και υλοποιήσαμε το δικό μας σύστημα διαχείρησης αρχείων, Orchestra-fs, το οποίο βασίζεται στο Orchestra για το κλείδωμα και τη δέσμευση blocks.

Παρουσιάζουμε την υλοποίηση του Orchsestra-fs και τα αξιολογούμε εκτελώντας διάφορα πειράματα χρησιμοποιώντας πολλαπλούς κόμβους ¨εξυπηρετητών' που παρέχουν τον χώρο αποθήκευσης και έναν κόμβο ¨πελάτη' που χρησιμοποιεί τον χώρο αυτον. Βρίσκουμε ότι επειδή η περισσότερη λειτουργία του συνολικού συστήματος βρίσκεται στο επίπεδο του block, το οποίο ειναι καλά δομημένο, υπάρχει πολυ μικρή επιβάρυνση εκτός απο το κόστος του πρωτοκόλλου TCP/IP και των ήδη υπάρχοντων του λειτουργικού συστήματος για την είσοδο/έξοδο απο το σύστημα του δίσκου. Τέλος, τα αποτελέσματα μας με το Orchestra-fs δείχνουν οτι η απόδοση του συστήματος αυξάνεται ανάλογα με τον αριθμό των ¨εξυπηρετητών' που παρέχουν τον αποθηκευτικό χωρο.

Επόπτης Μεταπτυχιακής Εργασίας: Άγγελος Μπίλας, Αναπληρωτής Καθηγητής

Acknowledgments

First of all, I would like to thank my supervisor, Angelos Bilas who gave me the possibility to work on the subject of scalable storage systems. I would also like to thank him for his guidance and support thoughout this work. Our discussions helped me understand how brilliant people are thinking and how a problem must be handled in order to be solved. I won't forget that many times he helped me for many hours in practical issues of my work, especially in the beginning, where no one else could help me. I feel lucky for having the chance to work with not only a great scientist but a great person as well. He was always there when i needed him. His advice helped me many times to overcome obstacles that seemed unsurpassable. His simple, yet scientific, way of thinking became valuable guide for me during the last few years.

I would also like to thank Michail D. Flouris, who helped me very much I was very lucky to work with and I truly believe that Michail is the ideal co-worker. I would like to thank him for his continuous help, his suggestions and observations throughout all this work. I'm also greatful to Manolis Marazakis for reading previous drafts of this thesis and for helping me to improve the contents of it. Many thanks also to Renaud Lachaize, for helping me improve the contents of both the presentation and my thesis. Our discussions helped me overcome many obstacles that came up these last months of my work. Thanks go also to Sven Karlsson for his valuable comments and directions for the presentation of my thesis.

I would also like to thank my fellow students at ICS-FORTH for their help. G. Panayiotakis, G. Kotsis, S. Passas, G. Passas, Y. Giannakopoulos, E. Kounalakis, M. Polyhronakis, D. Koukis, S. Antonatos, H. Athanasopoulos, D. Antoniadis, M. Moschous, M. Athanatos, Ch. Papachristos, M. Stamatogiannakis provided me with a pleasant and productive working environment.

I also thank my parents Charalambos and Kiriaki for their love and support during all these years. They sacrificed everything in order to help me reach my goals and I'm sure that they will do it again in the future. My young brother Marios-Christos who despite being so young, he gave me strength, while he was a source of joy for me. Many thanks to my older brother Kostas. Kostas and I study together all these years. He was my motivation and he was the one who helped me many times when I was desperate. Kostas is and will always be a source of inspiration for me. Many thanks are also to my grandparents Chrysoula and Dimitris. Their endless love and care enforced me with patience and strength to continue this journey.

Thanks are also to all my friends that were there when I needed them. Especially, I would like to thank Charalambos Papamanthou and Theofanis Oikonomou for being there in good and bad. Both of them carried the burden of the difficult moments i had and showed me that there is nothing more important in life than having close friends. Last but not least I would like to thank Kostas Dimitriou. Kostas was the first person I meet in Heraklion. He was the person who introduced me to the concepts of programming and computers in general. The greatness of his soul make me feel lucky to know him. Unfortunately, Kostas left us early for the final journey. Kostas, I hope you are happy wherever you are. I will never forget you ...

This work was financially supported by the Institute of Computer Science (ICS) of the the Foundation for Research and Technology - Hellas (FORTH), Heraklion, Greece. In addition, I would like to acknowledge both the fellowship support and the computer equipment that ICS-FORTH provided me during the course of my work.

To my parents Charalambos and Kiriaki
To my brothers Kostas and Marios-Christos

Στους γονείς μου Χαράλαμπο και Κυριακή
Στα αδέρφια μου Κώστα και Μάριο-Χρήστο

Contents

List of Figures

List of Tables

Chapter 1

Introduction

Future storage systems are required to scale to large sizes due to the amount of information that is being generated and the increasing capacities and dropping prices of magnetic disks. Network-attached, block-level storage, also called the SAN-approach due to the use of storage area networks (SANs), is proposed as one method for addressing these issues. In this approach, large numbers of magnetic disks are attached to a network through custom storage controllers or general-purpose PCs and provide storage to application servers. One of the main issues in this approach is the protocol used for gaining access from application servers to remote storage over the network. Traditionally, specialized interconnection networks and protocols have been developed for this purpose. For instance, SCSI [12] and Fiber Channel [11] are among the most popular such interconnects and associated protocols.

Although these approaches have been used and are still used extensively for building storage area networks, many problems have emerged due to changes in underlying technologies. First, these interconnects use custom network components and thus are not able to take advantage of the steep technology curves and dropping costs of commodity, IP-based networks. Moreover, the fact that they require specialized equipment leads to building storage systems and data centers with multiple interconnects. On one hand this does not allow for dynamic sharing of resources, since they need to be partitioned statically based on the type of interconnect servers are attached to. On the other hand, it increases significantly management overhead because multiple interconnects need to be maintained and optimized.

Thus, there is recently a lot of interest in examining alternative solutions that would both be able to reduce the cost of the underlying equipment and management, as well as better follow the technology curves of commodity networking components. One such approach is using IP-type networks and tunneling storage protocols on top of IP to leverage the installed base of equipment as well as the increased bandwidth available, especially of local area networks. It is currently projected that 10 Gigabit Ethernet interconnects will soon become commodity providing at least as much bandwidth as is available in high-end storage area networks.

iSCSI [25] is a storage networking standard that provides a transport layer for SCSI commands over TCP/IP. The main premise of iSCSI is that it can provide a familiar API and protocol (SCSI) to the application and storage nodes utilizing low-cost, commodity IP infrastructure and taking advantage of the technology curves. Thus, it functions as a bridge between the popular SCSI storage protocol and the popular TCP/IP local area network family of protocols. With the advent of 1 and 10 Gigabit Ethernet networks, there is increased interest in using iSCSI for local access.

Our final target is a data-center environment, where applications servers and user workstations are able to access the storage, which is efficiently distributed over a cluster. Such a system should have two main features. The first one is an efficient way of accessing the data on this storage system. iSCSI is a protocol that permits this kind of access. The other feature is a way of having safe sharing of data between multiple applications servers that run on top of the same storage pool.

In our work, from the prespective of the efficient data access of a storage system, we examine the implications of iSCSI on system performance and whether or not this protocol should be used for the development of novel data storage architectures. Next from the prespective of the efficient sharing between concurrent access of applications, we use an existing distributed storage system, *Orchestra* [20], and we built a filesystem on top of it to implement sharing between multiple application accesses on the storage pool.

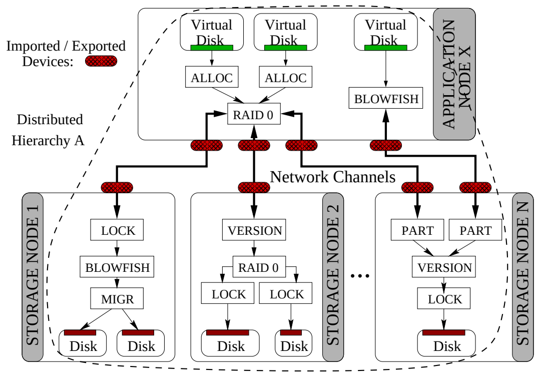
Specifically, *Orchestra* follows the current trend of pushing the functionality from the filesystem layer towards to the block layer. It is a system that virtualizes storage distributed over a commodity storage cluster. In particular, it allows extensions to storage functionality by providing hierarchies consisting of virtual devices layered over physical storage devices distributed in a commodity cluster. In addition, it can use virtual volumes that are mapped to physical devices but offer higher level semantics than those of physical volumes, such as versioning , encryption etc. Moreover, it enables sharing of the constructed hierarchies as storage volumes through a new block-level API or a traditional filesystem API. Overall, the features of *Orchestra* are:

- It provides a novel approach for building extensible, large-scale storage systems and consolidates system complexity in a single point, *Orchestra* deals with metadata persistence and allows for hierarchies to be distributed almost arbitrarily over application and storage nodes, providing a lot of flexibility in the mapping of system functionality to available resources.

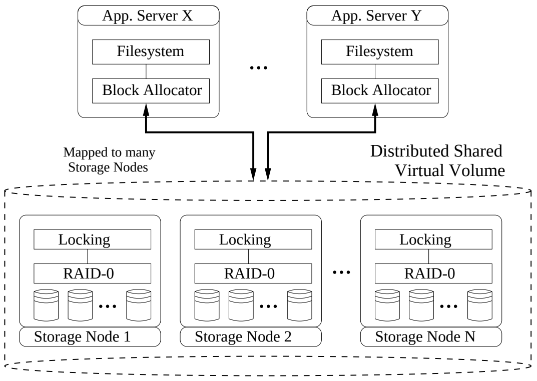
- It provides support for sharing at the block-level. Specifically, it provides block-locking and allocation facilities that can be inserted in *Orchestra* hierarchies where required. *Orchestra* currently provides simple but scalable policies for locking and allocation. More involved policies can similarly be implemented through additional virtual modules.

A sample distributed *Orchestra* hierarchy is shown in Figure 1.1(a).

Since, *Orchestra* provides sharing on the block level, we design and implement a new stateless, pass through filesystem above *Orchestra*, *Orchestra-fs*, in order to exploit this sharing of data for applications that operate on the filesystem layer. *Orchestra-fs* relies on *Orchestra* for block locking and block allocation services. *Orchestra*-fs provides only grouping of blocks in files and of files in directories, it does not use any internal distributed state and does not require explicit communication among its instances. We choose to avoid internal state and instance communication in order to avoid consistency problems between instances of the same filesystem that are distributed over the cluster. Figure 1.1(b) shows how *Orchestra*-fs is combined with *Orchestra* to provide file-level sharing over distributed *Orchestra* volumes.



(a) A distributed *Orchestra* hierarchy.



(b) The allocator and locking layers allow many FS clients to share a distributed volume mapped on many nodes.

FIGURE 1.1: Orchestra's hierarchy and layers

In order to identify the overheads of *Orchestra-fs*, we evaluate *Orchestra* with a small-scale storage system, that consists of three storage nodes and a single application node using **IOzone**[36].

The results show that *Orchestra-fs* provides good scalability as the number of server nodes grows in cases where there is no sharing conflicts. They also show that the performance of *Orchestra-fs* is comparable to the performance of the ext2 [2] for most of the workloads we try.

The contributions of this work are[1]:

- We evaluate iSCSI, using micro and application benchmarks. To achieve this, first we implement a framework of high-accuracy timers in kernel-space for measuring kernel code paths. Second we instrument the linux kernel using this timer framework in order to get a detailed breakdown of the system time spent by applications (TPC-H, Postmark, SPEC-SFS) that run over iSCSI.
- We develop a user-space filesystem (*Orchestra-fs*) to allow applications that access storage through a file system interface to take advantage of the advanced features of the *Orchestra* block-level framework. Finally, we evaluate the performance of *Orchestra-fs* and we compare it with ext2 [2] using various setups.

The rest of this thesis is organized as follows. Chapter 2 presents the evaluation of the iSCSI protocol including the methodology we use and the results of our evaluation. Chapter 3 presents the design, implementation and evaluation of *Orchestra-fs*. Finally, chapter 4 draws our conlusions and discusses limitations and future work.

[1]Parts of this work are appeared in [54] and [55].

Chapter 2

iSCSI Performance Analysis

2.1 Introduction

Although the adoption of iSCSI appears appealing, it is not clear at this point what is its impact on system performance. The introduction of both a new type of interconnect as well as a number of protocol layers in the I/O protocol stack may introduce significant overheads. Our work aims to examine iSCSI-related overheads in the I/O path. We investigate the iSCSI overheads associated with storage systems built out of commodity PCs and commodity Gigabit Ethernet interconnects. Although iSCSI may be used with customized systems as well, our storage nodes are commodity PCs with multiple (5-8) disks. Given today's technologies, such nodes may host about 1.5-2.5 TBytes of storage and in excess of 10 TBytes in the near future. Storage nodes are connected with application servers and application clients through a Gigabit Ethernet network. In our evaluation we use both microbenchmarks as well as real applications to examine the impact of iSCSI-based storage compared to directly-attached disks. We instrument the Linux kernel (2.4 series) and in particular the block-level I/O path to gain detailed information about I/O activity, and we examine both application- and system-level metrics. Overall, we see that the most significant kernel overheads in the I/O path are not only TCP and interrupt processing, as previous work has shown, but buffer cache processing as well. This suggests that novel I/O architectures should not only consider TCP-related costs, but buffer cache processing as well.

In this chapter we present an extended performance evaluation of the iSCSI protocol that is widely used for SAN environments. The following section describes our platform, as well as, the suite of the benchmarks that are used. Section 2.3 discusses the methodology we have used while section 2.5 presents our results.

2.2 Methodology

2.2.1 Experimental testbed

Our iSCSI testbed consists of 16 dual-processor (SMP) commodity x86 systems. Each system is equipped with two Athlon MP2200 processors at 1.8 GHz, 512 MBytes of RAM and a Gigabyte GA-7DPXDW motherboard with the AMD-760MPX chipset. The nodes are connected both via a 100 MBit/s (Intel 82557/8/9 adapter) and a 1 GBit/s (D-Link DGE550T adapter) Ethernet network. All nodes are interconnected via a single 24-port switch (D-Link DGS-1024) with a 48 GBit/s backplane. The 100 MBit network is used only for management purposes. All traffic related to our storage experiments uses the GBit Ethernet network.

The AMD-760MPX chipset supports two PCI 64-bit/66 MHz bus slots, three PCI 32-bit/33 MHz slots and two on-board IDE controllers, a system IDE controller with two ATA-100 channels for up to four devices, and an IDE Promise PDC20276 RAID controller with two ATA-100 channels for up to four devices. It is important to note that the IDE controllers are connected through a *single PCI 32-bit/33 MHz link* to the memory, which limits the aggregate IDE bandwidth to about 120 MBytes/sec. Each node has an 80-GByte system disk (Western Digital WD800BB-00CAA1, 2 MB cache) connected to the system IDE controller. Three of the system nodes are equipped with five additional disks of the same model. Three of the disks are connected to the system IDE controller and the other two to the Promise IDE RAID controller. All disks (except the system disk) are configured in RAID-0 [39] mode using the Linux MD driver (software RAID). The hardware RAID functionality of the Promise controller is not used. Since we are interested in examining commodity platforms, we use IDE/ATA disks. However, in the case of iSCSI we need to use the SCSI I/O hierarchy in the Linux kernel to generate SCSI commands for the iSCSI modules.

The operating system we use is Linux RedHat 9.0, with a kernel version 2.4.23-pre5 [4]. Not all Linux kernel versions provide support for fast disk I/O. In versions 2.4.19 to 2.4.22 a bug in the kernel driver for the AMD IDE chipset disables support for fast data transfers and results in low raw disk throughput. According to our measurements, versions up to 2.4.18 or later than version 2.4.22 offer the expected disk I/O performance. In our work we use version 2.4.23-pre5 for all the experiments. Furthermore, we use the Linux `hdparm` utility to set each disk to 32-bit I/O and to UDMA 100 mode. These result in an increase of maximum disk throughput from 25 to 45

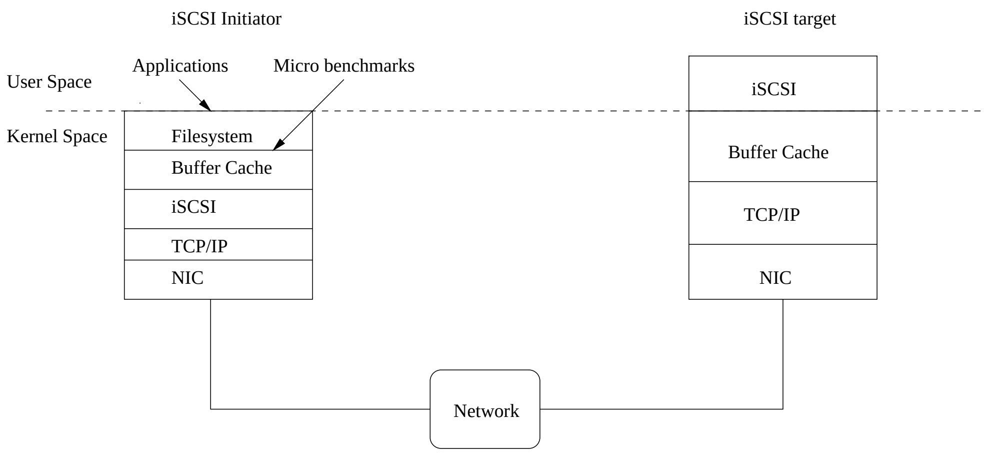


FIGURE 2.1: Intel iSCSI architercture

MBytes/s.

2.2.2 iSCSI implementation

During the course of this evaluation, we have experimented with various Linux iSCSI implementations [38, 7, 13]. [13] suffers from low performance, as we verified through several microbenchmark experiments. On the other hand, [38] requires SCSI disks on the target side. This makes it unsuitable for our work, since our goal is to build commodity storage nodes based on inexpensive IDE disk technology.

The Intel iSCSI implementation [7] we chose for our work has the fewest limitations. Mainly, that it supports only non-SMP kernels, so our kernel is built with no SMP support (i.e. only one processor is used). Secondly, we found that the Intel iSCSI target was originally developed for block devices up to 4 GBytes (32-bit byte addressing). Since we would like to build storage nodes with significantly higher capacity (currently 5x80 GBytes/node) and to experiment with datasets that exceed 2 GBytes, we have modified the iSCSI target to support 32-bit *block* addressing, which is adequate for our purposes[1]. In the Intel implementation, the iSCSI target runs at user level, whereas the iSCSI initiator runs in the kernel as figure 2.1 shows.

[1]The main issue is that the iSCSI target, which is implemented in user space, opens the target devices with an `lseek` call that repositions the fi le offset. The specifi c call used allows only 32-bit offsets and needs to be replaced with the `_llseek` call that handles 64-bit offsets.

2.2.3 Workload

To examine system behavior we use a set of microbenchmarks and applications: IOmeter [26], a workload generator that has been used extensively for basic evaluation of I/O subsystems, Postmark [27], a benchmark that emulates the I/O behavior of an e-Mail server, the TPC-H [49] decision support workload on MySQL [53], a popular open-source database, and Spec-SFS [45], a widely-accepted NFS server benchmark. Next we examine each benchmark in more detail.

IOmeter: IOmeter [26] is a configurable workload generator. The parameters we vary are: access pattern, mix of read and write operations, number of outstanding requests, and block size. We choose four workloads that represent extreme access patterns: all sequential or all random, 100% reads or writes, and two mixed workloads with 70-30% reads and writes. Finally, for each workload we vary the number of outstanding requests between 1 and 16 and the block size between 512 Bytes and 128 KBytes. The results we report are with 16 outstanding I/O requests, unless stated otherwise. In our discussion we use the average throughput, the average response time for each I/O request, and the total CPU utilization.

PostMark: PostMark [27] simulates the behavior of an Internet mail server. PostMark creates a large number of small files (message folders) that are constantly updated (as new mail messages arrive). PostMark can be configured in two ways [27]; The number of files to create and the total number of transactions to perform on this set of files. In our experiments we use inputs 50K/50K, 50K/100K, 100K/100K, and 100K/200K. A new filesystem is created with `mkfs` before each experiment to ensure that the filesystem state does not affect the results.

MySQL: MySQL is a popular open-source database [53]. To examine the behavior of this important class of applications we use the TPC-H workload [49]. The TPC-H benchmark models a decision support system and consists of a suite of 22 business-oriented ad-hoc queries and concurrent data modifications. In our work we use TPC-H queries 1-3, 5-8, 11, 12, 14, and 19. Since we need to perform multiple runs for each query, we omit the rest of the queries that take a long time to complete (in the order of hours). A new filesystem is also created with `mkfs` and the database is reloaded before each TPC-H experiment (each experiment includes all the queries we use). The size of the database we use is 10 GB.

Briefly, the characteristics of the TPC-H queries we use are:

Q1: The “pricing summary report query” reports the amount of items billed, shipped and returned as of a given date.

Q2: The ”minimum cost supplier query” finds which supplier should be selected to place an order for a given part in a given region.

Q3: The “shipping priority query” retrieves the 10 unshipped orders of the highest value.

Q5: The “local supplier volume query” lists for every nation in a region, the revenue volume done through local suppliers.

Q6: The “forecasting revenue change query” quantifies the amount of revenue increase that would have resulted from eliminating certain company-wide discounts in a given percentage range in a given year.

Q7: The “volume shipping query” determines the value of goods shipped between certain nations to help in the re-negotiation of shipping contracts.

Q8: The “national market share query” determines how the market share of a given nation within a given region has changed over two years for a given part type.

Q11: The “important stock identification query” finds the most important set of suppliers’ stock in a given nation.

Q12: The “shipping modes and order priority query” determines whether selecting less expensive shipping modes is negatively affecting the critical-priority orders by causing more parts to be received by customers after the committed date.

Q14: The “promotion effect query” monitors the market response to a promotion such as a TV ad or a special campaign.

Q19: The “discounted revenue query” reports the gross discounted revenue attributed to the sale of selected parts handled in a particular manner. This query is an example of code such as might be produced programmatically by a data mining tool.

Spec-SFS: Spec SFS97_R1 V3.0 [45] measures the throughput and response time of an NFS server. The first iteration starts with a total NFS server load of 500 operations/sec, while each consecutive iteration increases the load by 100 operations/sec until the server is saturated. In our work we measure total response time and response rate for NFS operations.

2.3 Timing Measurements

Finally, in our work we are interested in examining both application-specific as well as system-wide metrics. In particular, we are interested in presenting detailed breakdowns of system I/O activity in each kernel layer. To obtain the breakdown of kernel time in various layers we instrument the Linux kernel source with our own stop-watch timers. Next we describe in details our kernel level framework of timers.

2.4 Virtual Timers

In this section we describe the framework of **Virtual Timers** we developed in order to measure the kernel code paths, especially the ones related to the block I/O.

2.4.1 Timer Semantics and API

Most CPUs today provide cycle counters that can be used for profiling purposes. For instance, the x86 architecture provides an 64-bit cycle counter that can usually be read with a single assembly instruction. Figure 2.2 shows how this is achieved in the x86 architecture. This instruction reads the value of a processor register that counts clock cycles since the last boot of the system. Clock cycles normalized with the processor frequency is a very accurate measure of time. This profiling method is not only very accurate, but also incurs very low overhead, requiring a single assembly instruction to read the cycle counter. In contrast, using OS facilities such as `do_gettimeofday()` that use the system real-time clock have two main disadvantages: (i) They need several instructions to compute the time and thus incur high overhead and (ii) The granularity they provide is in the order of milliseconds, while cycle counters can potentially measure nanosecond intervals on modern high-frequency processors.

Figure 2.3 shows the timer API that we developed. This API is developed into the kernel and especially for 2.4 kernel versions. A stop-watch timer has essentially six primitive operations: `alloc/dealloc(), start(), stop(), clear()` and `read_time()`. `alloc()` creates a new timer. `start()` starts counting time by reading the current timer value and storing it. `stop()` reads the current timer value, subtracts from it the previously-stored timer value, and stores their difference. This difference indicates the time elapsed between the start and stop points. The timer can then be restarted and stopped, accumulating the total time for all intervals. `read_time()` returns the current total, i.e. the elapsed time. `Clear()` resets the total time

```
inline long long GetCurrentCycles(void) {
   LARGE_INTEGER val;

   __asm__ __volatile__("rdtsc ":
       "=a" (val.split.LowPart),
       "=d"(val.split.HighPart));

  return val.QuadPart ;
}
```

FIGURE 2.2: Function for reading cycle counter in the x86 architecture.

```
int alloc (_timer_ctx_t *tc,char *name,char flag);
int dealloc (int timer_id, _timer_ctx_t *tc);
void start (int timer_id, _timer_ctx_t *tc);
void stop (int timer_id, _timer_ctx_t *tc);
void clear (int timer_id, _timer_ctx_t *tc);
void read_time (int timer_id, _timer_ctx_t *tc);
```

FIGURE 2.3: Virtual timers API.

measured by the timer. Finally, when the timer is not needed, we can free it using `dealloc()`.

Using the above timer concepts, profiling a code path requires merely including the path in a pair of `start_timer(), stop_timer()` calls. Using multiple timers, allows us to measure independent code paths.

This simple profiling approach, however, becomes more complicated when applied to modern operating systems, mainly because of two features: (a) multi-tasking and preemption, and (b) interrupts.

Modern OSes try to minimize blocking and wait time by overlapping execution of concurrent kernel and user tasks as much as possible. For this reason, they switch between tasks whenever waiting on events is required. Processes are placed in various event queues and the OS scheduler selects the next process to run on the CPU. Since stop-watch timers use physical time between the start and stop operations, their measurement of elapsed time will include any time between these two points, i.e. the time other processes may have run on the CPU. A similar situation occurs with interrupts (software or hardware) that are issued during the run time of a process. In this case, the interrupt handler routine will be included in the time of the specific code path.

To deal with these issues in a transparent manner, our framework offers two kinds of timers: *physical* and *virtual*.

- *Physical*: Physical timers measure all time between start and stop operations.
- *Virtual*: Virtual timers automatically remove all wait and interrupt times.

Furthermore, timers can be categorized as private or global:

- *Private*: The timer is visible only inside the context of a single task and cannot be accessed by other tasks.
- *Global*: The timer is visible from all tasks. Such timers can be started and stopped by any task running in the system.

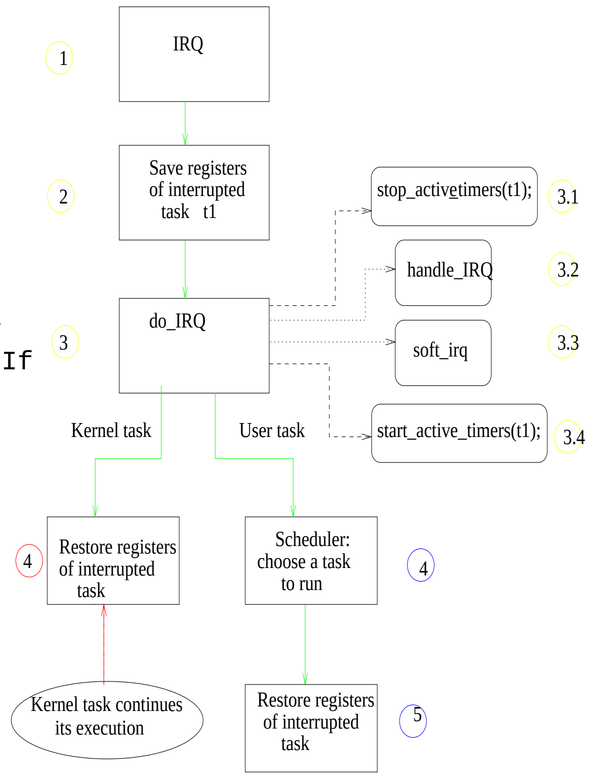
Thus one can have the following timer semantics:

- *Physical-Global*: These timers are the simplest timers traditionally available for profiling.
- *Physical-Private*: These timers are useful when a task needs to profile other activity (e.g. wait time) that occurs during its execution.
- *Virtual-Private*: This type of timer can be used to profile paths in a single task without interfering with other system or user tasks.
- *Virtual-Global*: These timers do not provide useful semantics, since profiling global executions path requires physical timers.

Virtual-private timers offer the advanced semantics described earlier and are the most challenging to implement. Supporting virtual timer semantics requires two additional internal functions that are not part of the framework API: `pseudo_start, pseudo_stop.` These functions are internal and they are called either from the scheduler or from interrupt handler in order to stop/start active timers of a task during a scheduling or interrupt event.

```
schedule() {

need_resched_back:
    prev = current;
    if ((prev->state == TASK_INTERRUPTIBLE)
        && signal_pending())
      prev->state = TASK_RUNNING;
    else
      del_from_runqueue (prev);

repeat_schedule:
    next = idle_task ();

    search_list();

    /* c  is the  goodness value.
       The  biggest the more likely
       for a process to  run next. If
       it is  0 the process cannot
   run  on the CPU
       that has used until now. */

    if (c == 0)
        goto repeat_schedule;

    if (prev == next)
        goto same_process;
    prepare_To_switch ();

    stop_active_timers (prev);
    start_active_timers (next);
    switch (prev,next);

same_process:
    if (current->need_resched)
goto need_resched_back;
    return;
}
```

(a)


IRQ
1
Save registers of interrupted task t1
2
do_IRQ
3
stop_activetimers(t1);
3.1
handle_IRQ
3.2
soft_irq
3.3
start_active_timers(t1);
3.4
Kernel task
User task
Restore registers of interrupted task
4
Scheduler: choose a task to run
4
Kernel task continues its execution
Restore registers of interrupted task
5

(b)

FIGURE 2.4: Left figure is the pseudo code of the Linux Kernel. Right figure is the generic interrupt execution path in Linux.

2.4.2 Timer Implementation

In this section we describe our implementation of virtual timers for Linux. First we describe the Linux scheduler, for kernel versions of 2.4 [8]. Linux uses separate contexts for entities that can be scheduled independently. Such entities are called *tasks*. A task may be a user process, a kernel thread or a signal. Interrupts are distinguished from tasks and run in their own context. All user tasks are preemptible. The scheduler may suspend the execution of the current user process at any time and select another process to run, according to a scheduling algorithm. On the other hand all kernel tasks are non-preemptible (in the 2.6.x kernel they are preemtible). However kernel tasks yield the CPU to improve responsiveness when waiting for I/O to complete [8].

Figure 2.4(a) shows the scheduler pseudo-code. Initially, the scheduler checks the state of the current task. If the state is set to `TASK_INTERRUPTIBLE`, the scheduler checks if there are signals that may require processing. If there are no such signals, the task is removed from the queue of runnable tasks. In both cases, the flag that indicates if the task needs to be rescheduled, is disabled. After the scheduler has applied the scheduling algorithm to all tasks in the runnable queue, it selects the most suitable task to run or the `idle task` if the queue is empty. At this point, it is guaranteed that the selected task will run. The context switch merely saves the context of the previous task and restores the context of the task to run.

To implement virtual timers we need to: (i) `pseudo_stop` all timers of the previously running task that were active when the scheduler run and (ii) `pseudo_start` the timers of the task that will run next and that were active when the process was preempted. For this we need a bitmap per task indicating the task's active timers. The bits in this bitmap are set when the timer is started and are reset when the timer is stopped.

Next we describe how we deal with interrupts. When an interrupt happens, the task that holds the CPU is suspended and a general interrupt handling routine is called [1], as shown in Figure 2.4(b). After the completion of the interrupt there are two possibilities: (i) If the task that was interrupted is a user task, the scheduler is called and selects a task to run. (ii) If the task is a kernel task, the control of the CPU returns to this task. Consider, for example, that a task T1 uses the CPU. When an interrupt arrives, the OS saves the context of T1 in the top-most interrupt handler and calls the `do_IRQ` function, which is the entry point of all interrupts. `do_IRQ` calls other functions, including the interrupt handler of the specific interrupt. After the interrupt is handled, the OS decides which task should be scheduled next based on the reschedule flag of the interrupted task. To deal with interrupts, we instrument the generic interrupt handler of the Linux kernel. In the beginning of this function we `pseudo_stop` the active timers of the interrupted task. When the interrupt routine is completed we `pseudo_start` the active timers of the next task to run. We note that we could avoid the `pseudo_start` operation in the case where the scheduler is called, however we prefer to include it because it is easier to implement in the current Linux code.

In the case of nested interrupts, we will try to `pseudo_stop` the active timers of the previous task. In this case, the previous task is the task that was originally interrupted and not the interrupt context. Since `pseudo_stop` accumulates the timer difference, we need to avoid multiple accumulation of a single time difference. For this reason `pseudo_stop` checks if a particular active timer has been pseudo_stopped already. Similarly, `pseudo_start` checks if the timer is already pseudo_started. This scenario happens only in nested interrupts. We measured the frequency of this scenario during our experiments and found that it accounts for only 3% of the total `pseudo_start and pseudo_stop` calls.

Finally, another scenario we need to address is the case where an interrupt occurs while the scheduler itself is running, and the framework has pseudo_started the active timers of the next task to run, just before the actual context switch. In this case, execution is still in the context of the current task, but the framework has started the timers of the next task to run. As mentioned previously, the virtual timers code in the interrupt handler will stop the active timers of the task that was interrupted. In this case the interrupted task appears to be the task that was preempted and not the next task to run. This means that the virtual timers for the next task to be scheduled will continue to measure time, during interrupt handling. This occurs because starting virtual timers and performing the context switch are not atomic operations. The result is that the next task to run will include the interrupt time. However, this will not result in, otherwise, corrupted or incorrect measurements. One way to fix this, is to make the two operations atomic by disabling interrupts. However, since this is an infrequent case (it has never occurred in our experiments) we prefer to keep our implementation simple and not deal with this atomicity issue.

Our virtual timers framework use two main data structures. The first is general and is used for all tasks running on the system. Its elements store information about the execution time of all tasks and all timers in the system. This structure is allocated at system initialization (when the idle process is created).

The second data structure is unique per task and is used to implement the virtual-private semantics. The elements and information are similar to the first, global data structure. This second structure is attached to the kernel's `task_struct`, which contains all information for each task. The structure is allocated at the entry point of every new task in the system, `do_fork.` Every task has its own instance of this struct, thus the data for the timers is also unique for each task. Allocation of this structure happens at the entry point of every new task in the system, `do_fork`.

Finally, all timing information is available through the `/proc` filesystem. The global statistics, i.e. all timers for all tasks, are accessible through the `/proc/ktimers` file, while separate task

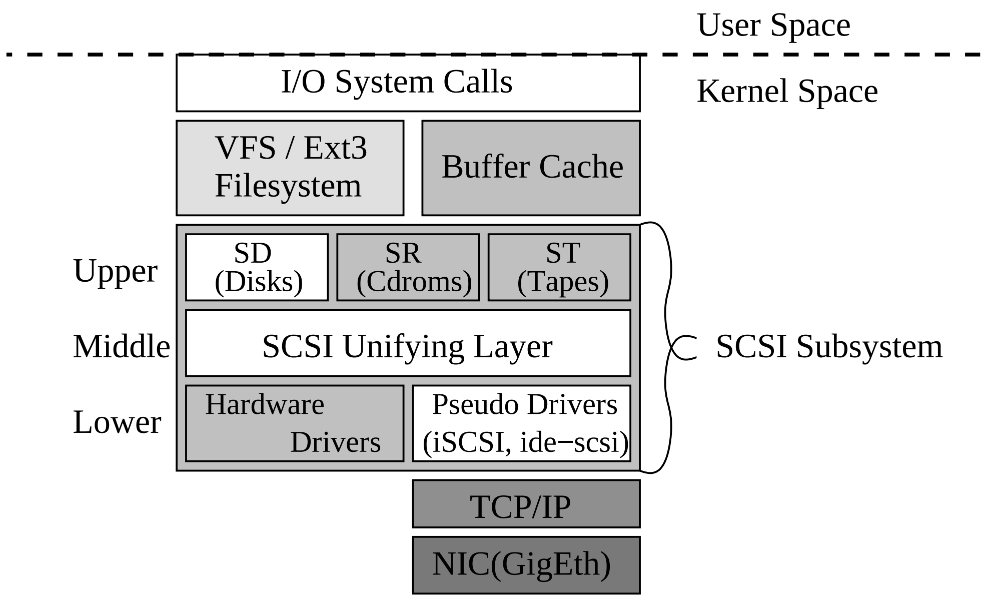


FIGURE 2.5: Kernel Layer Hierarchy.

timings are accessed through `/proc/ktimers_"taskid"`.

2.4.3 Kernel Instrumentation

In our instrumentation we have placed timers in the borders of consecutive layers of the I/O path. Thus, starting from the high-level `read()` and `write()` system calls, we are able to measure times spent in each layer as an I/O call is forwarded from the user process to the actual disk device driver (or network device driver in the case of an iSCSI device).

Figure 2.5 illustrates the kernel layers that are of interest for the iSCSI call path. The *system call* layer is the interface to the user-space world. The I/O system calls use VFS (or generic filesystem) calls to perform their tasks, while individual filesystems, such as ext3, plug their code into the VFS handlers. The buffer cache code is used for managing memory pages that cache file inodes or disk blocks. The buffer cache functions are used both by VFS and the lower-level filesystem code sometimes in a recursive manner. Thus, it is generally hard to distinguish system time spent in VFS, Ext3 and buffer cache code since their calls are interdependent. However, in our analysis we do not aim at a detailed analysis of the filesystem time, but instead on understanding iSCSI-related overheads. For this reason we measure two separate times for the VFS/Ext3 layer: (a) the time for read/write calls, labeled as *"FS: read/write"* in our breakdown graphs and (b) the time for performing file and directory operations (e.g. create, delete, locate), labeled as *"FS:*

File Mgmt”. These two components are possible to distinguish because they are initiated through different system calls.

Below the filesystem level is the SCSI hierarchy that consists of three layers: upper, middle, and lower SCSI layers. We combine the time spent by the upper and middle layers in a single component, labeled as *“SCSI”*. The low-level SCSI driver is the iSCSI module, which we time separately. We also quantify what happens below the iSCSI layer, namely in the TCP/IP layer and the network device driver and the associated interrupt handler (the DL2k module in our systems). Our measurements of the TCP/IP stack, labeled *“TCP”*, contain all the time spent in both the send and receive paths only accounting for the iSCSI traffic. Finally, we measure the time spent in the network device interrupt handler separately (labeled *“NIC IRQ”*).

It is important to identify the kernel layers where memory copying of data related to iSCSI traffic occurs. There are two such cases: one in the *FS:Read/Write* component, where data is copied between the kernel’s buffer cache and the application buffer and second in the *TCP* layer, where data is copied between the NIC’s buffer and kernel memory. These two copies occur for all data stored on iSCSI storage. In the directly-attached disk case, the NIC copy does not occur and is replaced with a DMA to the buffer of the hard disk controller.

2.4.4 System Configurations

For our evaluation we have examined three system configurations.

(A) *Direct-attached (or Local) Disk.* Five IDE disks directly attached to the application server.

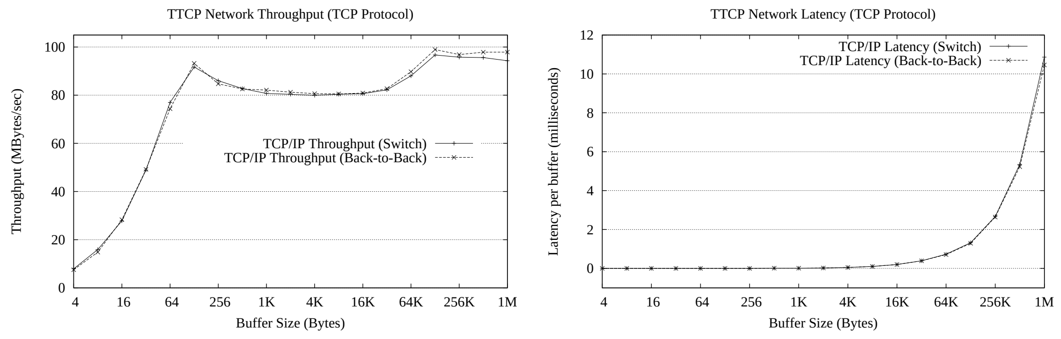
(B) *iSCSIx1.* One storage node with five disks connected to the application server through Gigabit Ethernet. The storage node (iSCSI target) exports a single RAID-0 volume through iSCSI.

(C) *iSCSIx3.* Three storage nodes (iSCSI targets) with five disks each connected to the application server through Gigabit Ethernet. Each storage node exports a single RAID-0 volume through iSCSI. Since, each storage node exports five disks, the total number of disks for this configuration is fifteen. The three iSCSI volumes are concatenated with software RAID-0 on the application server.

2.5 iSCSI Results and Analysis

In this section we describe the experimental results and analysis of the iSCSI protocol. It is divided in three parts. In the first part we report some basic measurements and costs of our platform. The next part consists of the results of the micro and application benchmarks we have used. The last part reports a breakdown of the time spent into the kernel during the execution of the application benchmarks.

2.5.1 Basic measurements



(a) TCP Throughput (b) TCP Latency

FIGURE 2.6: TCP/IP throughput and latency over D-Link DGE550T Gigabit Ethernet NIC, measured with ttcp.

Figure 2.6 shows the throughput and latency of the Gigabit Ethernet network we use, measured with `ttcp`. The system achieves a maximum bandwidth of about 800 MBit/s using 128K packet sizes. Our system configuration does not use jumbo frames since many commodity network interfaces and switches do not support this feature. The size of a jumbo frame is 9000 bytes while the default one that we use is 1500 bytes.

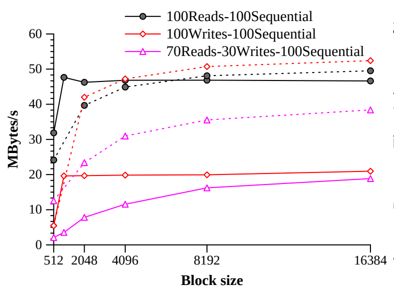
Figure 2.7 shows the basic throughput for the disks and controllers we use, measured with IOmeter. We see that each disk is capable of 45 MBytes/s throughput for sequential read accesses and about 20 MBytes/s for sequential write accesses. A mix of 70% read and 30% write operations achieves also about 20 MBytes/s throughput. Each IDE controller can transfer about 120 MBytes/s. As mentioned in Section 2.2, the PCI bus in our systems is a 33MHz/32Bit bus, resulting in a theoretical peak of 125 MBytes/s. Given that we use two IDE controllers in each system we expect that the maximum I/O throughput in each node is limited by the PCI bus. CPU utilization and response time for a single disk are depicted in Figure 2.7(a).

Finally, figure 2.7 shows the throughput, CPU utilization and response time for iSCSI when using a RAM-disk on the iSCSI target with one outstanding I/O request. We see that the initiator reaches 90% CPU utilization for reads and about 80% for writes for 1K or larger I/O request sizes. Thus, when using iSCSI, I/O throughput is limited by network bandwidth to about 50 MBytes/s (with 16 KByte requests) at a system utilization between 80-90%. Minimum response time (512 Byte requests) is less than a 100 μs.

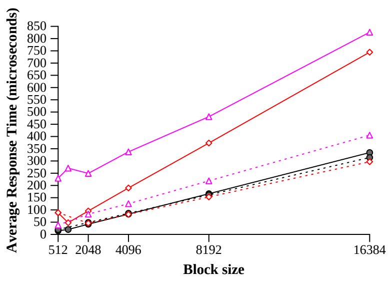
2.5.2 Microbenchmarks

First, we look at IOmeter to understand basic aspects of the local and iSCSI configurations. Figure 2.8 shows that for directly-attached disks and sequential read requests, maximum throughput approaches 120 MBytes/s, limited by the PCI bus in our systems. For write requests, maximum throughput is about 50 MBytes/s. The read performance is higher than write due to the aggressive prefetching (read-ahead) the software RAID driver performs. In the iSCSIx1 configuration, maximum throughput is limited to 40 MBytes/s and maximum read throughput is limited to about 25 MBytes/s. The writes in the iSCSI configuration are faster because of the write-back cache on the iSCSI target nodes. Reads, however, are slower because data must be read from the remote disk. In both cases, maximum throughput is achieved at 4 or 8KByte I/O requests. When using 70% read-30% write mix, throughput is about 50 MBytes/s for the local case and about 20 MBytes/s for the iSCSIx1 configuration. iSCSIx3 achieves about the same write throughput as iSCSI, but performs much better in reads. This is due to the larger buffer cache that three nodes have compared to one node in the iSCSIx1 case. Random access patterns exhibit a significantly lower throughput in all cases.

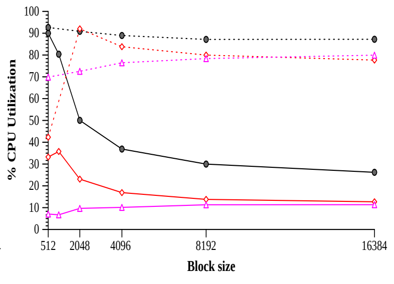
Figure 2.9 shows that the local and iSCSIx1 configurations have similar response times. In contrast, the iSCSIx3 setup shows higher latencies for larger block sizes, mainly due to network congestion from the three iSCSI connections. Finally, Figure 2.10 shows the CPU utilization in the initiator (application server). Maximum CPU utilization with sequential requests is between 75-90% in the local case and between 60-70% in the iSCSIx1 case depending on I/O request size. We note that given the achievable throughput, iSCSIx1 utilization is higher compared to the local configuration. iSCSIx3 shows very high CPU usage, especially in the case of sequential reads. This is due to increased network throughput, which results in high network processing times (including memory copies of data from the NIC).



(a) Throughput

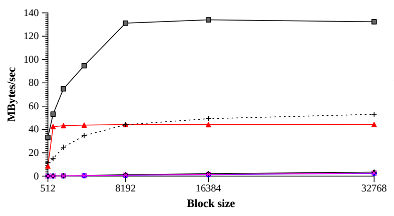


(b) Average Response Time

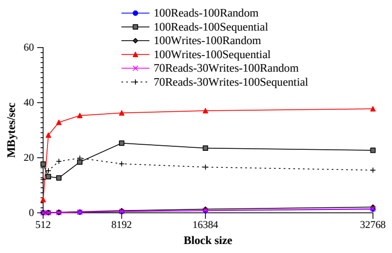


(c) Cpu Utilization

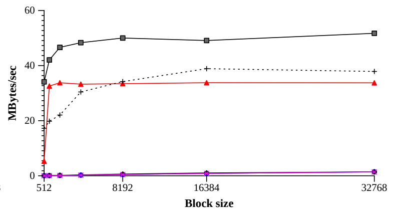
FIGURE 2.7: IOmeter statistics for a single local disk vs. an iSCSI RAM-disk. Dotted lines denote the single local disk and solid lines the iSCSI RAM-disk.



(a) Directly-attached disks

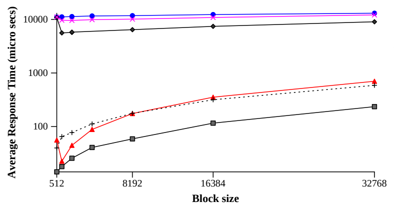


(b) iSCSIx1

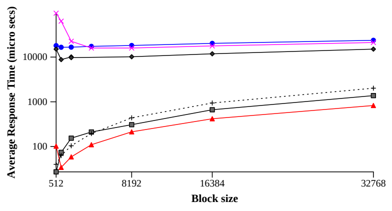


(c) iSCSIx3

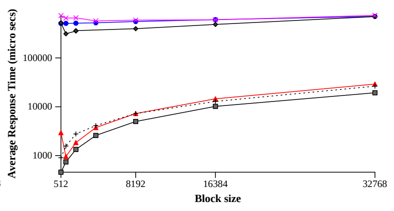
FIGURE 2.8: IOmeter throughput.



(a) Directly-attached disks

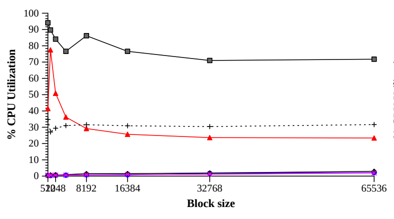


(b) iSCSIx1

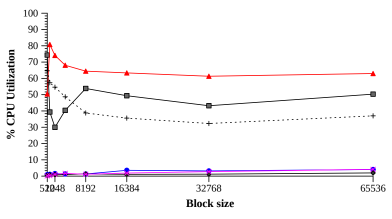


(c) iSCSIx3

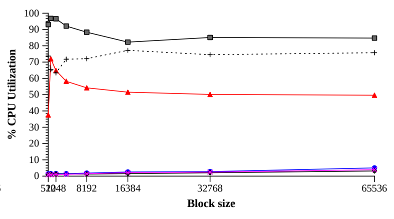
FIGURE 2.9: IOmeter average I/O response time.



(a) Directly-attached disks



(b) iSCSIx1



(c) iSCSIx3

FIGURE 2.10: IOmeter CPU utilization.

TABLE 2.1: Postmark results. Throughput is in KBytes/sec.

Input Size (#files/#trans)	Tx/sec	Read Throughput	Write Throughput
50K/50K	168	208.68	655.27
50K/100K	168	303.55	629.10
100K/100K	83	120.58	375.79
100K/200K	78	158.45	329.16

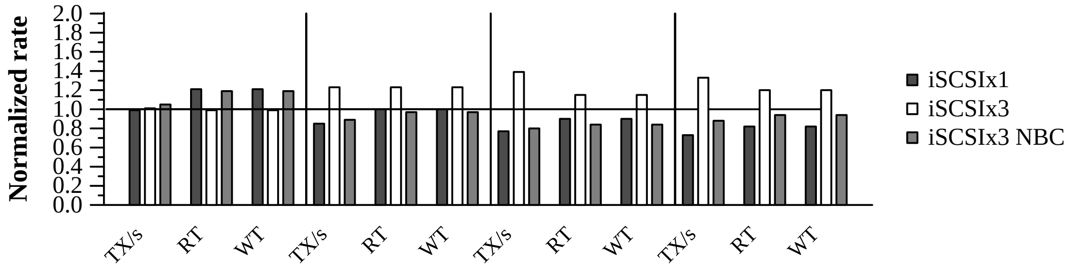
2.5.3 Application Performance

PostMark

Overall, we find that Postmark is sensitive to I/O latency and iSCSI reduces performance up to 30%. For the same reason, however, in iSCSIx3 performance improves up to 40% due to the increased target buffer cache that reduces I/O latency. Also, iSCSIx3 is able to saturate the host CPU, something that is not possible in the direct and iSCSI configurations. Next we present a more detailed analysis.

Table 2.1 shows the performance of PostMark for the local configuration. Figure 2.11 shows the performance of PostMark for the rest system configurations(iSCSIx1, iSCSIx3, iSCSIx3 without buffer cache) compared to the local one. We notice that using iSCSI reduces transaction rate between 0% and 25% compared to the direct configuration. In iSCSIx3 all application metrics improve compared to direct by 0% to 40%. Moreover, we note that the largest improvement occurs in the configurations that use a larger number of transactions for a given number of files. This is due to the fact that when increasing the number of transactions and keeping the number of files constant, the larger iSCSIx3 target buffer cache becomes more effective.

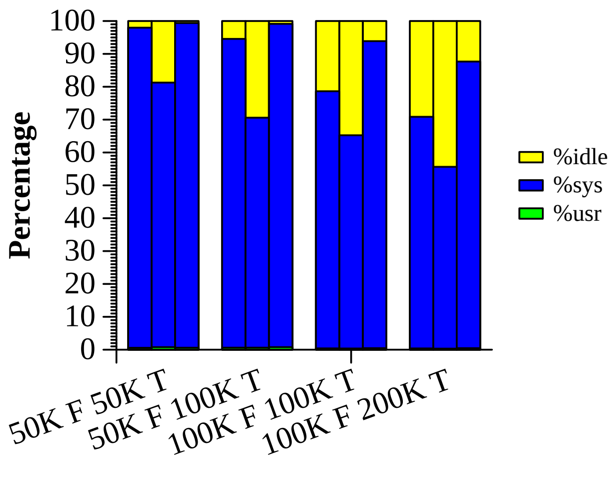
Figure 2.12 shows the execution time breakdown for PostMark. We see that system time drops by 17-35% in iSCSIx1, compared to the direct case. This is due to the increased response time in iSCSIx1. PostMark performs synchronous I/O, and thus, is sensitive to I/O response time. In iSCSIx3, response time improves due to the larger target cache, which results in higher system utilization that reaches almost 100% for all input sizes. To verify this we disable the buffer cache in the Linux kernel in all iSCSI targets. Because there is no way to completely disable the buffer



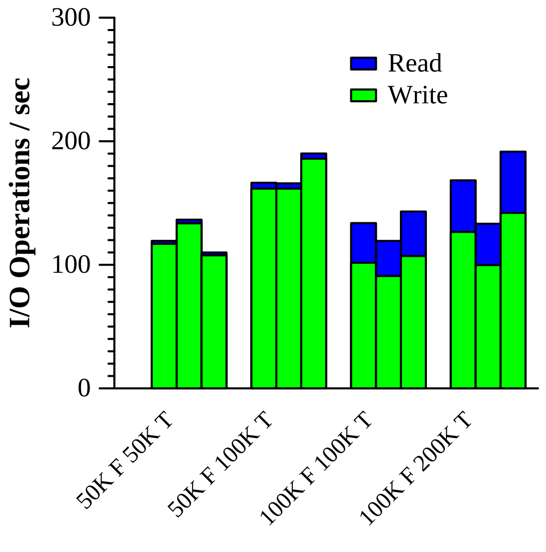
(a)

FIGURE 2.11: Postmark results normalized to the direct confi guration. Each graph represents one input size. Each group of bars refers to one application metric: Transactions/s, Read Throughput, and Write Throughput. Each bar refers to one system confi guration: direct, iSCSIx1, iSCSIx3 and iSCSIx3 without buffer cache (left to right).

cache, we decrease the memory of the target to 96 MBytes. This is the lowest possible size of memory the target machine needs to boot. Figure 2.11 shows that PostMark performance drops for larger input sizes (transaction number), even below the direct configuration (Table 2.1). Thus, PostMark benefits mostly from the presence of the increased I/O subsystem cache in the iSCSI and iSCSIx3 configurations, demonstrating one of the advantages of using iSCSI.



(a) Execution Time Breakdown



(b) I/O rate (IOs/s)

FIGURE 2.12: Postmark execution time breakdown and I/O rate. The left bar in each pair refers to the direct confi guration and the middle bar to the iSCSIx1 confi guration and the right to the iSCSIx3.

TABLE 2.2: TPC-H query execution time in MySQL (in seconds).

Query	q1	q3	q5	q6	q7
Exec time (s)	74.37	54.25	59.74	12.33	60.92
Query	q8	q11	q12	q14	q19
Exec time (s)	137.59	23.19	16.32	24.11	18.11

Figure 2.12 shows the I/O rate at the initiator (application server) for each system configuration and input size. This rate shows the number of requests that reach the physical disks in the direct configuration and the iSCSI layer in iSCSIx1 and iSCSIx3. The I/O rate in PostMark follows the same pattern as the transaction rate reported by the application.

MySQL

Overall, TPC-H is sensitive to disk I/O throughput and CPU utilization. Thus, iSCSI reduces performance by up to 95%. However, iSCSIx3 is able to scale the number of disks (and the total size of I/O cache). This improves disk I/O throughput and reduces the gap with the direct configuration, in some cases even improving performance up to 25% (Q14). Next we discuss our results in more detail.

Table 2.2 shows the execution time for the TPC-H queries we use for the local configuration. Figure 2.13 shows the execution time for all setups compared to the execution time of the local configuration. We see that using iSCSI increases execution time between 1% (Q8) and 95% (Q6). In the iSCSIx3 configuration, execution time reduces significantly compared to the iSCSIx1 case and is within 40% of the direct case (and usually within 20%). Similarly to PostMark, iSCSIx3 is able to recover the performance degradation of using iSCSI with a significant increase in resources.

To distinguish whether the performance improvement with iSCSIx3 is due to the increased target cache or the larger number of disks we also run experiments with the iSCSIx3 configuration where the target buffer is disabled (Figure 2.13). We see that, except for query Q14, in all cases the performance degradation is fairly small (within 7%) which suggests that TPC-H benefits mostly from the increased number of disks under iSCSIx3.

Figure 2.14 shows the execution time breakdown for each query. First, we note that between direct and iSCSIx1 system time increases by up to 95% (Q14) which indicates that iSCSI intro-

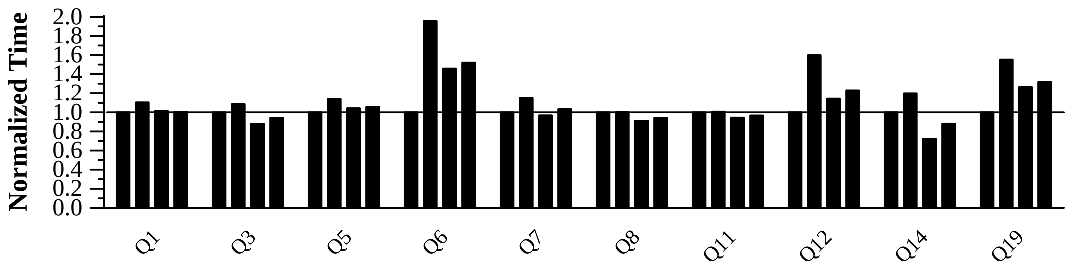


FIGURE 2.13: TPC-H execution time normalized to the direct confi guration. Each group of bars refers to one query, whereas each bar refers to one system confi guration: iSCSIx1, iSCSIx3, and iSCSIx3 without target cache (left to right).

duces a significant overhead. Moreover, user time reduces in all queries and by up to 112% (Q6) indicating that either system time takes up useful cycles from the host CPU or that the increase in iSCSI response time results in lower CPU utilization.

Similarly to iSCSIx1, iSCSIx3 exhibits a higher user time compared to the direct configuration for all queries where it performs better. Idle time in iSCSIx3 is almost 0% in all queries. Thus, application server CPU is saturated and further improvements in application performance may only be achieved with lowering CPU utilization.

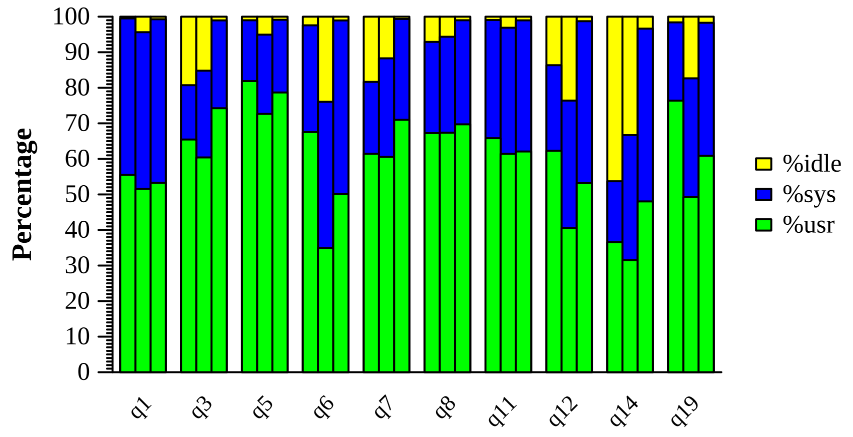
Finally, Figure 2.14 shows the I/O rate for each configuration. We see that the large differences in user time, especially in queries Q6, Q12, and Q19 are reflected to differences in the I/O rate.

Spec-SFS

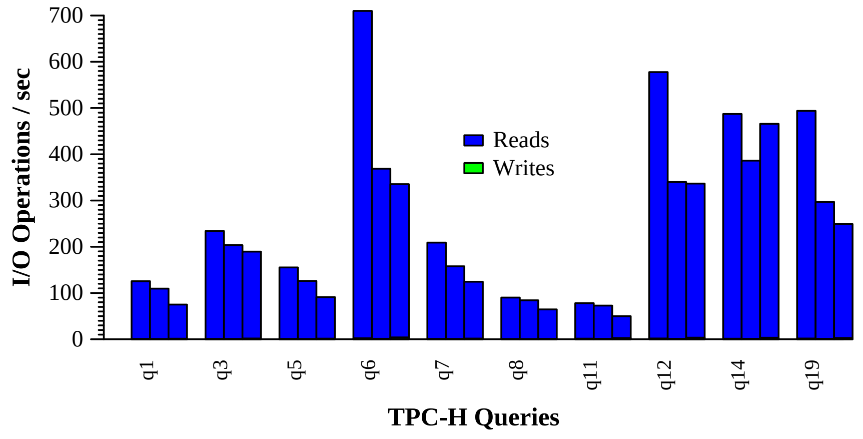
Overall, we find that Spec-SFS is hurt by mixing client-server and iSCSI traffic over the same network. Although consolidating all traffic on top of a single network is considered an advantage of iSCSI, this has an adverse effect on Spec-SFS performance.

Figure 2.15 shows that the iSCSIx1 configuration saturates faster than the direct configuration at 600 I/O requests/s, as opposed to 700 I/O requests/s (14% difference). Moreover, I/O response time is larger in the iSCSIx1 configuration for all I/O request loads by about 25% to 40%. However, the I/O rate is similar in both the direct and iSCSI cases for the the loads that do not saturate iSCSI (up to 600 IOs/s). The iSCSIx3 configuration behaves similarly to iSCSIx1, however, I/O response time improves and is within 25% of the direct case in most cases.

The execution time breakdown for Spec-SFS (Figure 2.15) shows that the Spec-SFS server is idle most of the time, which suggests that the network bandwidth of the Spec-SFS server is limiting



(a) Execution Time Breakdown



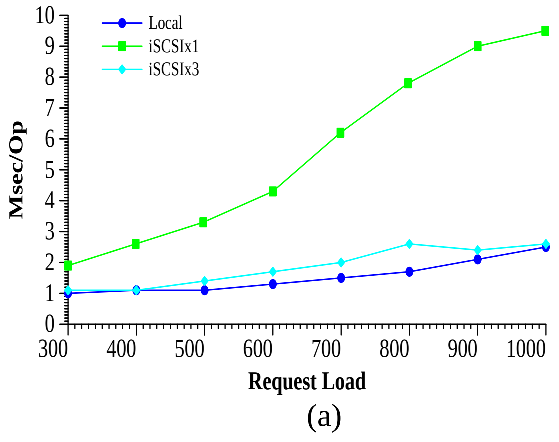
(b) I/O rate (IOs/s)

FIGURE 2.14: TPC-H I/O rate and execution time breakdown. The left bar in each pair refers to the direct configuration, the middle bar to iSCSIx1, and the right to iSCSIx3.

system performance. Note that all systems in this experiment are attached to the same Gigabit Ethernet switch, meaning that both Spec-SFS and iSCSI traffic traverse on the same link that connects the Spec-SFS server (iSCSI initiator) to the switch. For this reason, in the iSCSIx1 and iSCSIx3 configurations the system saturates at a lower number of I/O requests, since using iSCSI increases the traffic on the network.

2.5.4 System Overhead Breakdown

Next we examine the overhead introduced by iSCSI in the initiator's I/O protocol stack.



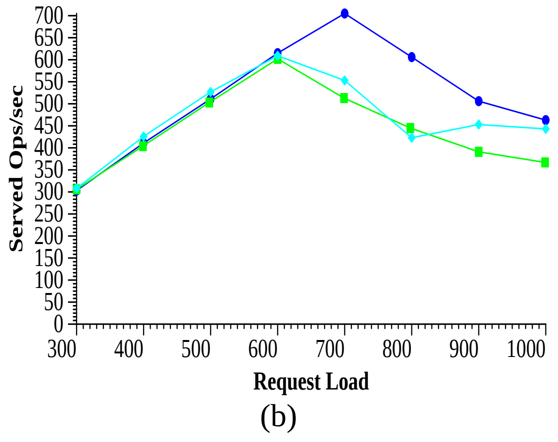
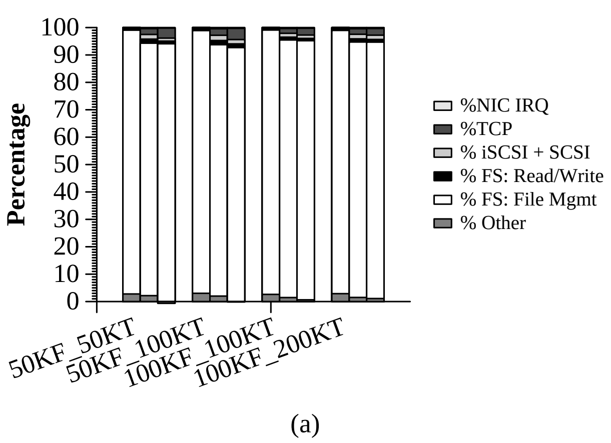


FIGURE 2.15: Spec-SFS results.



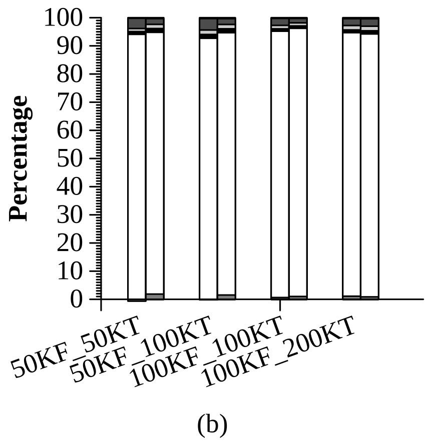
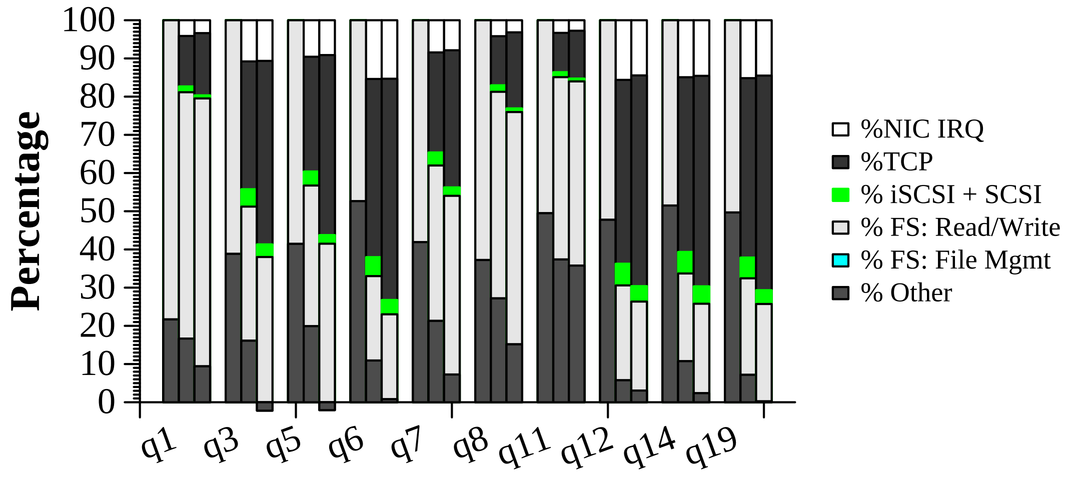


FIGURE 2.16: Postmark system time breakdowns with and without (NBC) buffer cache in the storage targets. Each group of bars refers to one system confi guration: direct (left), iSCSIx1 (middle), and iSCSIx3 (right).

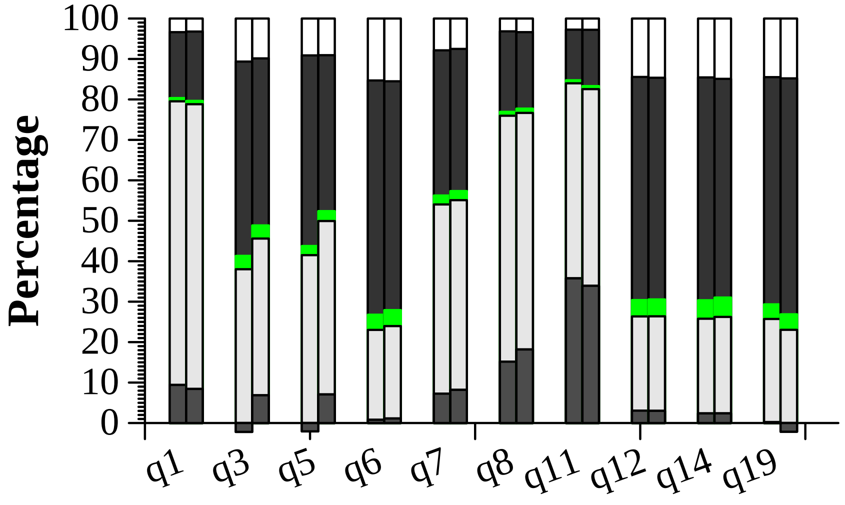
Figure 2.16(a) and 2.17(a) shows the breakdown of system time [2] in the major components of the I/O stack (Figure 2.5). We see that in PostMark most (90%-95%) of the system time is spent in the filesystem component. This is due to the fact that PostMark represents mail folders as directories and writes mail messages to separate files. Thus, each mail operation results in (multiple) file and directory operations (`open, close, search, delete, create`) that account for most of the system overhead.

In TPC-H queries, the file management overhead is minimal, because during query execution the only file operations that take place are read and write operations that are passed directly to the

[2]The negative components in some bars are due to measurement errors, which, however, do not affect our results.



(a)



(b)

FIGURE 2.17: TCP-H system time breakdowns with and without (NBC) buffer cache in the storage targets. Each group of bars refers to one system confi guration: direct (left), iSCSIx1 (middle), and iSCSIx3 (right).

block I/O hierarchy. On the other hand, buffer cache management is in many cases a significant component of system time (up to 80%).

In terms of the block I/O stack, the overhead is divided almost equally between block I/O and network components. The major components of the block I/O stack involved in the I/O path is the buffer cache management (between 22% and 63% of system time). The major network components are the send and receive paths of the TCP/IP stack (between 10% and 44% of system time) and the NIC interrupt handler (between 4% and 18% of system time).

Thus, we see that a significant part of the I/O overhead for TCP-H is in the block-level I/O hierarchy and that using iSCSI has a significant impact due to TCP and NIC interrupt handler overheads. Also, when comparing iSCSIx3 to iSCSIx1 we see that block-level I/O overheads increase significantly and up to 18%. When disabling the buffer cache in all targets in the iSCSIx3 configuration (Figure 2.17(b)) trends remain the same as before, since TPC-H is not affected significantly by the available target cache size. Finally, Spec-SFS exhibits very little system time in the initiator (Spec-SFS server) and thus, we do not examine the related breakdowns.

Overall, we see that the most significant kernel overheads in the I/O path are not only TCP and interrupt processing, as previous work has shown, but buffer cache processing as well. This suggests that novel I/O architectures should not only consider TCP-related costs, but buffer cache processing as well.

2.6 Related Work

The authors in [35] evaluate the performance of iSCSI when used for storage outsourcing. Similarly to our work, the authors use both microbenchmarks and real applications (TPC-C and Postmark). However, unlike our work, they focus on network issues. They examine the impact of latency on application performance and how caching can be used to hide network latencies. The authors in [10] examine the performance of iSCSI in three setups: An optimized, commercial-grade back-end in a SAN environment, a commodity-type back-end in SAN environment, and a commodity-type back-end in a WAN environment. They perform high-level experiments and examine system throughput with microbenchmarks. The authors in [31] use simulation to examine the impact of iSCSI and network parameters on iSCSI performance. They only examine throughput of a simple test and the consider iSCSI PDU size and network Maximum Segment Size, TCP Window Size, and Link Delay. The authors in [14] present the design and implementation of iSCSI for Linux and perform preliminary evaluation of their system with a simple microbenchmark. The authors in [15] examine the impact of TCP Window Size and iSCSI request size for LAN, MAN, and WAN environments. The authors find that the default TCP parameters are inappropriate for high-speed MAN and WAN environments and that tuning of these layers is required.

In contrast, our work focuses on the impact of iSCSI on application server performance and we examine in detail, by instrumenting the Linux kernel, system (kernel) overheads introduced by iSCSI. We also examine how adding system resources in an iSCSI configuration impacts application and server performance.

The authors in [33] examine the impact of iSCSI on application server performance. They use a simple microbenchmark directly on top of block-level storage or through NFS. Similarly to our results they find that iSCSI impacts system behavior significantly and that Ethernet interrupt cost is the most significant source of overhead. Our results show similar behavior for the network-related costs. However, we find that other, iSCSI-related costs can also be very high when using real applications. The authors in [33] conclude that using jumbo frames reduces interrupt overhead by about 50% but state that this may not be a practical option in real systems where not all components in the network path may support jumbo frames. In our work, we do not consider Jumbo frames, since we also feel that this may not be representative of practical setups.

The authors in [48] discuss an implementation of iSCSI that avoids a copy on commodity network adapters. They use simple microbenchmarks to examine the performance of their implementation and find that it reduces CPU utilization from 39.4% to 30.8% for reads and that it does not have a significant impact for reads.

During the course of the performance evaluation of the iSCSI protocol we developed a framework of virtual timers for profiling kernel code paths. Below we present the related work for this part of the work.

Specifically, there exist several system profiling tools. They can be first categorized based on the events they are able to profile:

(i) OS-level tools that profile operating system events such as page faults, context switches, network traffic etc. Such tools usually profile coarse grain events. The Microsoft Windows System Monitor [32] counts and logs OS events and hardware resources. Similar statistics are exported through the /proc filesystem in Linux and are displayed by utilities such as “top” [9]. Such monitoring tools are intended for system administration and not kernel development, which requires kernel code profiling information.

(ii) Code-level tools that profile user or kernel code. Today, such tools almost exclusively rely in some type of cycle counter to provide fine-grain measurements: Our work is in this second category. Most of the tools used in this category are currently based on sampling methods. Oprofile [5] is a tool that enables OS profiling using the symbol sampling techniques to get statistics for the various symbols that are executed. Oprofile periodically (at very short intervals) generates an interrupt that samples the value of the PC register in the CPU. Using the PC, oprofile determines in which function (symbol) the CPU was executing at that time. At the end of the experiment it reports a breakdown of the percentage of execution time that was spent in every function. Another sampling profiler using similar techniques is PAPI [6]. PAPI is designed for portability and provides an interface to access the hardware performance counters found on most modern processors. VTune [3] is a sampling profiler for x86 systems (both for Linux and Microsoft Windows). VTune also reports other CPU events by means of the x86 hardware event counters.

These sampling profilers though can neither provide a detailed breakdown of execution path nor report the exact time of execution of a kernel function, since they use approximations based on frequency counters to get the samples.

2.7 Summary

In summary, we see that using iSCSI without increasing system resources compared to a local configuration has a significant impact in all applications we examine. However, the impact of iSCSI differs in each case.

iSCSIx3 is able to scale system resources and recover and in some cases improve system performance. Postmark benefits from the increased target buffer cache, TPC-H by the increased number of disks (and to a lesser extend by the increased buffer cache), whereas Spec-SFS remains limited by the available network bandwidth in the unified interconnect.

Chapter 3

The *Orchestra-fs* Filesystem

3.1 Overview

Orchestra provides a block-level API, while many applications access storage through a file system interface. To allow such applications to take advantage of the advanced features of *Orchestra* and to demonstrate the effectiveness of our volume sharing mechanisms it is necessary to provide a file system interface on top of *Orchestra*.

One approach to achieve this is to use an existing distributed file system. Depending on the distributed file system, *Orchestra* can be used to either provide a number of virtual volumes each residing in a single storage node [44], or a single distributed virtual volume built out of multiple storage nodes [47]. However, in both cases, many of the features that are intended to be provided by *Orchestra* would be replicated by the file system. More importantly, current distributed file systems are fairly complex and difficult to extend in any way.

For these reasons, we provide our own file system API on top of *Orchestra* using a user-level library that provides only core file system functions, mainly grouping of blocks in files and of files in directories. *Orchestra*-fs is a *stateless, pass-through* file system that uses its internal metadata to translate file calls to the underlying *Orchestra* block device. Our approach demonstrates also that by extending the block layer we are able to significantly simplify file system design in distributed environments and especially when it is necessary to support system extensibility. Thus, the block allocator uses the block volume facilities for free-list allocation and locking.

Orchestra-fs supports most file system operations required by applications and that are present in the POSIX interface:

- File operations: `open, close, remove, creat, read, write, stat, rename, symlink, link`
- Directory operations: `mkdir, rmdir, opendir, readdir, chdir, getcwd`

A main feature of *Orchestra*-fs is that it does not require explicit communication between multiple instances running on the same or different application nodes. Usually, such communication is required for two purposes: (i) mutual exclusion and (ii) metadata consistency. *Orchestra*-fs uses the corresponding mechanisms provided by *Orchestra* volumes for this purpose.

- Mutual exclusion: *Orchestra*-fs uses the multiple-reader single-writer locking mechanism provided by *Orchestra* to achieve mutual exclusion between multiple applications accessing a single file system through a single or multiple application nodes. Currently, *Orchestra*-fs locks and unlocks files during the `open/close` calls respectively. This granularity of locking is coarse, however, it is realistic for many applications. Although finer grain locking can also be provided by protecting individual read and write operations, this is beyond the scope of our work.
- Metadata consistency: The only metadata required by *Orchestra*-fs are i-nodes and the directory structure. To avoid maintaining internal consistent state in memory, *Orchestra*-fs does not perform any caching of metadata or data but uses the underlying *Orchestra*-fs block device. Thus, accesses to files or directories in *Orchestra*-fs may result in multiple reads to the underlying block device for the corresponding i-nodes and directory blocks. The maximum number of such reads that may happen in the case of very large files are four. *Orchestra*-fs relies on the underlying *Orchestra* block device to reduce accesses to the physical disks.

The design goal of *Orchestra-fs* is to exploit the sharing potential that *Orchestra* provides for applications that operate on the filesystem layer. To achieve this, it relies on two main services that are provided by *Orchestra*. The first is the *block allocation* service which manages the blocks along the volumes of *Orchestra*. The second is the *locking mechanism*, which is used in order to guarantee the integrity of data in the distributed storage sytem.

One main feature of *Orchestra-fs* is that it supports 64-bit addressing scheme and large files. For 32-bit addressing the maximum file size permitted is 4 Gigabytes. Considering that applications

```
typedef struct inode
{
        long long           i_ino;
        umode_t             i_mode; //Type of inode. i.e file,directory
        unsigned int        i_count; // Usage Counter. How many fs instances
                                    // have this inode opened.
        size_t              i_size; //File length in bytes
        time_t              i_atime; // Last access time
        time_t              i_mtime; // Last modifications of content of file
        time_t              i_ctime; // >>>>  attributes >>
        long long           directs [ DIRECT_LEVELS ];
        long long           first_indirect;
        long long           second_indirect;
        long long           third_indirect;
} inode_t;
```

FIGURE 3.1: Inode Structure

like video streaming may use files with sizes up to hundreds of Gigabytes, we choose to support 64-bit addressing.

3.2 Filesystem Entities

In this section we describe the main entities in *Orchestra-fs*.

3.2.1 File

Orchestra-fs provides a persistent mapping between file names and stored data blocks. File size can vary from several bytes to the total capacity of the volume. The *file* has also metadata that uniquely specifies it and resides in its *i-node* structure as described above. In fact, an *i-node* contains all the information needed to connect the data of the *file* with its name. *i-node* holds pointers to all the data and metadata blocks that constitute the file. Such metadata are the size, the creation-modification time, the reference count etc. An *i-node* needs only a few bytes (256 bytes) but we don't want to access a whole filesystem block (4 KB) to access only a few bytes. For this reason, *Orchestra-fs* uses a separate block size (256 bytes) for accessing *i-nodes*. This block size is named *small block size* and is provided by *Orchestra*'s block layer. Figure 3.1 shows the contents of the *i-node* structure that *Orchestra-fs* uses. *Orchestra-fs* stores a limited number of pointers, that points directly to data blocks, into the *i-node* structure in order to quickly access very small files. *Orchestra-fs*, also reserves some bytes of the *i-node* to hold some references to blocks that stores pointers to data blocks. To access larger files, *Orchestra-fs* uses the technique of indirection levels.

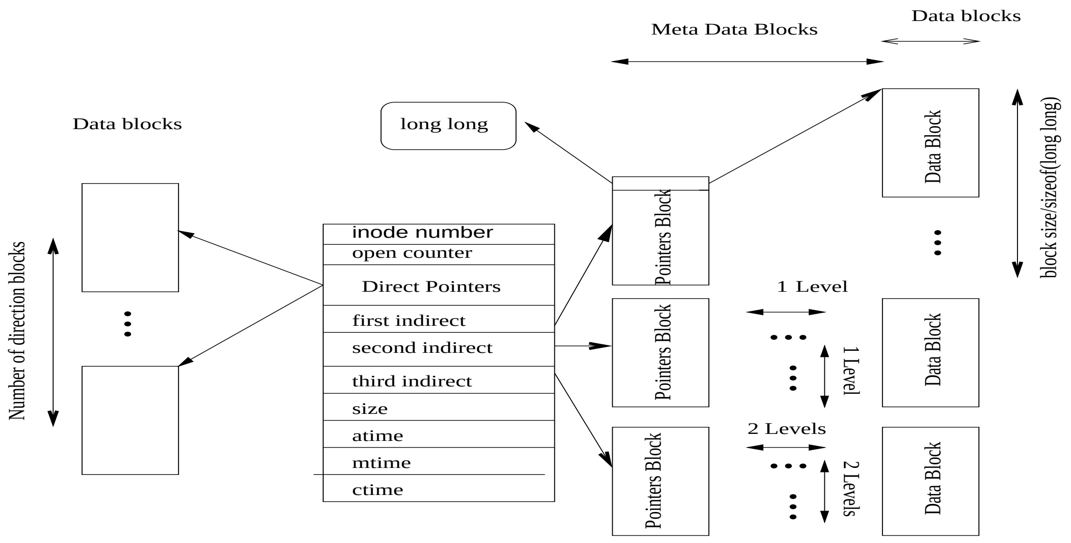


FIGURE 3.2: Contents of an inode

Specifically, every *i-node* contains an indirect pointer. This pointer points to a block that contains pointer blocks to data blocks. Thus, using only a single block, the filesystem is able to point to a larger number of data blocks. This number $N(x)$ is equal to filesystem's block size divided by the size of block addresses. This kind of indirection is called first indirect. To extend the file size beyond this size, *Orchestra-fs* applies this technique twice. This time, the block is called second indirect block. An *i-node* contains the address of the double indirect block which in turn contains pointers to first indirect blocks. The number of indirect blocks a double indirect block can store is the same with the number of data blocks a first indirect block can point to. However, there are many cases in which the file size should extend to hundreds of Gigabytes. In such cases the triple indirect technique is applied. A triple indirect block contains $N(x)$ second indirect blocks. The total file size achieved using direct, first, double, and triple indirect blocks is $T(x)$:

$$N(x) = \frac{blocksize}{sizeof(longlong)} \tag{3.1}$$

$$T(x) = DirectPointers * blocksize + N(x) * blocksize + N^2(x) * blocksize + N^3(x) * blocksize \tag{3.2}$$

For a filesystem block size of 4 KB, the max file size is 550 GB.

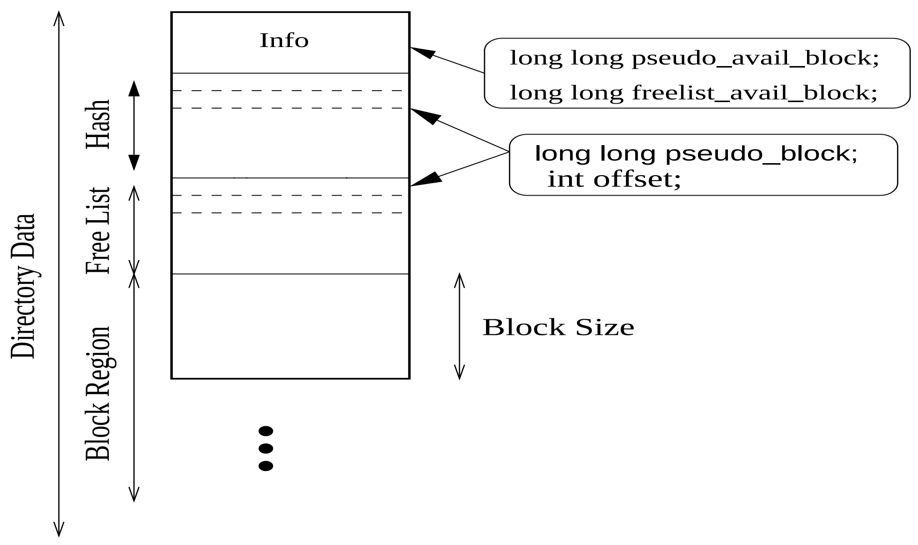


FIGURE 3.3: Directory structure

3.2.2 Directory

Directories are used by *Orchestra-fs* to create and present hierarchical structures of all available files. *Orchestra*-fs internally treats files and directories in a very similar manner. *Orchestra*-fs locates directory i-nodes and blocks exactly in the same manner as file i-nodes and blocks. The directory data is simply a list that holds all directory entries (files and directories). To allow for efficient search of files in directories that contain very large numbers of entries, *Orchestra*-fs uses a hash table to store entries in a directory. The directory hash table is always stored in the first blocks of a directory, followed by the free-list of available hash entries. Currently, the hash table occupies a (statically) configurable numbers of blocks. When accessing a file or directory, *Orchestra*-fs hashes its name to a bucket in the containing directory hash table. Each bucket contains one (no hash conflict case) or more (hash conflict case) blocks that may contain the file i-node. Then, *Orchestra*-fs reads the specific block and examines its contents to locate the i-node for the directory entry. In case of conflicts, each hash table bucket can grow dynamically up to the maximum space allowed for the full hash table in single directory.

When a hash entry becomes full, we need to select another hash entry in a transparent manner. For this purpose we reserve another region, which is called the free list section and consists of hash entries, similar to the hash section. Whenever a hash entry within the hash section becomes full it is redirected to the first available hash entry from the free list section. To implement this we hold a pointer to the hash entry of the hash section to the next available hash entry in the free list section.

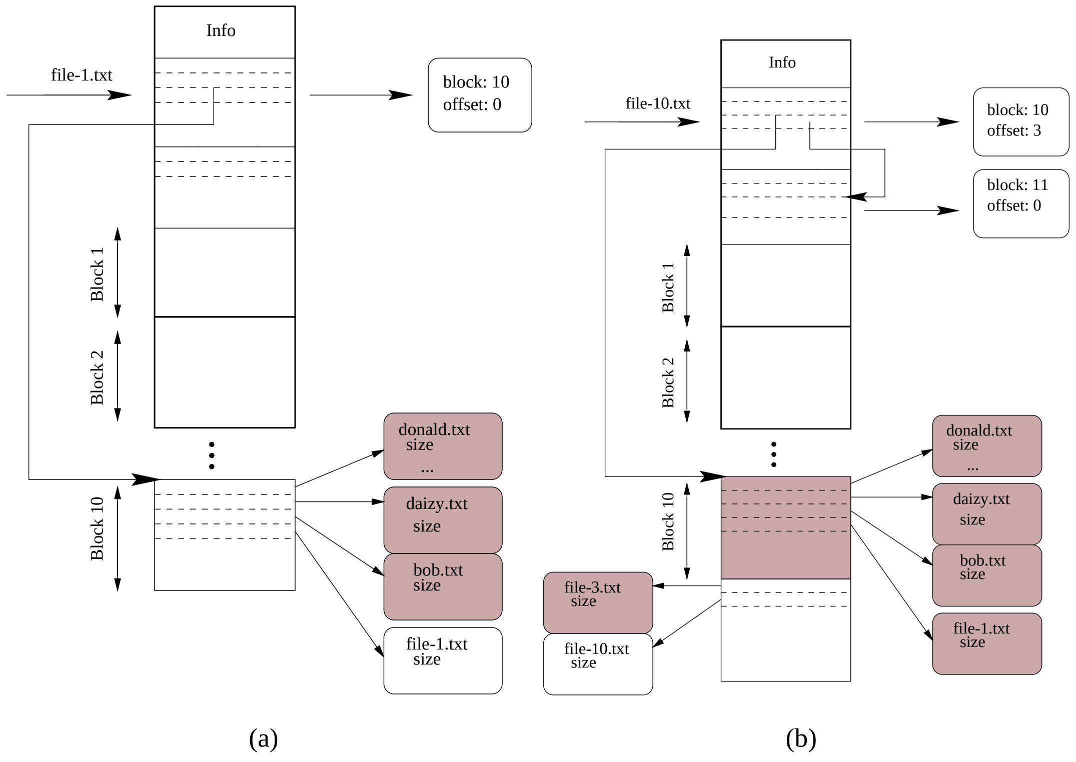


FIGURE 3.4: Creation of a fi le examples.

We explain in more details this procedure using two examples. Assume that the parent directory is in a state where some entries (files or directories) are already stored under its scope. Figure 3.4 (a) shows the first scenario. We first create “file-1.txt”. *Orchestra-fs* passes this file through the hash function of the parent directory, which produces an integer that specifies the position of file’s hash entry within the hash region. Block 10 already stores some directory entries. “donald.txt”, “daizy.txt”, and “bob.txt” occupy the first three slots of block 10. *Orchestra-fs* stores the file “file-1.txt” into the fourth slot of block 10, as it is the first empty slot available. Whenever *Orchestra-fs* looks up for this file, it will follow the same procedure.

In the next scenario we create “file-10.txt” as shown in Figure 3.4(b). However, in this case the selected hash entry is full, so a new one must be found. The third hash entry of the free list section is selected for this purpose. This entry stores its directory entries in the block 11 of the pseudo block region. The second slot is the first available one within this block. This is where *Orchestra-fs* stores the file “file-10.txt”

3.3 Filesystem Operations

This section describes the basic operations that are provided by *Orchestra-fs*.

Initialization: The superblock of *Orchestra-fs* holds information such as the block size of the block layer, the block size that the filesystem uses (it must be a multiple of the block's layer block size) and the block size used for handling i-nodes. The above initialization procedure takes place when a filesystem is first created by means of a separate filesystem utility (mkfs).

Mount/Unmount: Prior to accessing the filesystem, the information, that is stored in the super block, must be restored in memory. Before applications can access the *Orchestra-fs*, it must restore the information of the *superblock* to the memory, by using the *mount* utility. When traditional file systems are mounted, they allocate a memory region that holds the information mentioned above. Since, these filesystems reside in kernel, this region is always accessible from processes until this filesystem is explicitly unmounted. This is not the case for a user space implementation. The mount process will act similarly with the kernel space implementation but when it finishes its execution, it will die, along with its memory region. To address this problem, every application, that uses *Orchestra-fs*, reads the superblock structure and retrieves it to its memory address space at its first operation. When a filesystem is not needed any more, it can be *umounted* by using *umount* utility. This operation flushes the contents of the memory region of the mount process into the *superblock*.

Create/Delete: Creating a file is the most fundamental operation of *Orchestra-fs*. It is divided in two phases. The first one is the allocation of an i-node that represents the new file. This i-node is also filled with its metadata as described in section 3.2.1. At the time of its creation, the file is empty and its size is zero. Next, the file needs to be connected with the parent directory. Specifically, a new entry that contains the pair of file and i-node must be added to the parent directory's entry list.

The same procedure for the creation of a file is followed for the creation of a directory. The only difference is that during the creation of a directory its contents must be initialized as described in section 3.2.2. As mentioned above a major concept of a filesystem is its hierarchy. To implement this when a new directory is created, a reference to its parent directory must be created as well. In Unix-like environments, the parent directory is expressed with an entry named “..”. Accordingly, the current directory is expressed with an entry named “.”. This mechanism enables traversing the whole filesystem from the root node to its leaves.

When a file is deleted, *Orchestra-fs* traverses the filesystem hierarchy from the root downwards to find the parent directory. If the file is found into the parent directory, the filesystem tries to grant write access to this file in a way described in the section 3.4. The deletion operation is divided in two parts. First, the file is removed from the entry list of the parent directory. Then, tha metadata, data, and i-node of the file are deallocated.

Open/Close: When a file is opened, *Orchestra-fs* traverses the filesystem hierarchy to find the parent directory. Next, the target file i-node is read, as discussed in Section 3.2.2. If the file is found, a file desciptor is returned to the process that opened the file. The file descriptor is used for all subsequent I/O to the file.

Orchestra-fs treats opening of a directory in a similar manner. The meaning of opening a directory is to provide a mechanism for easily accessing the contents of that directory. It returns a handle that is used from *readdir* to gather the entries of the directory. The *close* operation merely frees the locks and the structures of an opened file.

Lseek: *lseek* is used to change the current offset pointer in an opened file. This pointer will exist until the file is closed.

Write/Read: The *write* operation translates an offset to a physical block number, and performs a disk I/O operation to the data blocks. The *read* operation reads a file and uses the same mechanism with write to map the file offset to a physical block number. Then, it performs an I/O operation to get the data from the data blocks.

Rename: During a *rename* operation, the source and destination file are validated. Then, *Orchestra-fs* verifies whether the source file and the destination directory of the new file exist. If they exist, *Orchestra-fs* deletes the file from the source directory and creates a new file into the destination directory with the new name.

Stat: *Orchestra-fs* supports the *stat* call, which returns the size, creation-modification time, and permissions of a file.

Readdir: In order to facilitate the applications to get a view of the filesystem hierachy, *Orchestra-fs* provides the *readdir* operation, that fills in a structure with all the entries of the directory. Using this structure, applications can traverse the filesystem hierarchy.

Link: *Orchestra-fs* supports both soft and hard links. Soft links offer the ability to link a named entity that refers to an existing entity file or directory. This is achieved by creating an entry into the directory's list and substituting the name of its i-node structure with the path of the existing file. Whenever the filesystem accesses this new entity, it transparently redirects it to the original one. Hard links store the inode number of the existing file or directory. Thus, the "link" is achieved by using the same i-node structure. In this approach, since i-nodes are used by multiple hard links the space of the file can only be released after the deletion of the hard link.

3.4 Locks

Orchestra provides a storage pool that is distributed accross the nodes of the cluster. Multiple *Orchestra-fs* instances of the same volume can run above this nodes in order to provide applications the same view of the volume. Every operation that takes place in one filesystem instance must be visible from the other instances that run on top of other nodes. To ensure consistensy of data among different file system instances across *Orchestra*s nodes, *Orchestra-fs* uses locking. *Orchestra-fs* supports multiple readers - single writer locks.

When creating a file or directory, *Orchestra-fs* traverses the filesystem hierarchy to find the parent directory. Then, *Orchestra-fs* allocates and locks a new i-node in order to fill it properly. Then, it locks the parent directory and adds this entity to the directory list of the parent directory. Finally, *Orchestra-fs* first unclocks the i-node of the parent directory and then the i-node of the new file. *Orchestra-fs* follows the similar procedure with locks to remove a file or directory.

Orchestra-fs locks files at open/close time rather than at read/write calls. After the file is opened, applications can read/write this file without worrying for data corruption. This lock is released only when the file is closed. Figures 3.5 and 3.6 show the pseudo code that implements the open and close operations on a file respectively.

When *Orchestra-fs* renames an entity, two operations must happen. First, the deletion of the source file and then the creation of the destination file. The locks of these operations are handled in the way described above.

```
open()
{
        if READ permission requested (READ flags on open)
       {
            lock (whole inode);
            /* I know there are no others accessing the same file
             in the inode  I'm the first reader
            */
            if ( lock ok )
            {
                reader_counter = 1;
                unlock( 2nd half of inode );
                enter file descriptor in list with READ permission;
                return ok;
            }
            lock ( 2nd half of inode );
            /* I know there are other readers (multiple reader access) in the inode
               I'm NOT the first reader (someone else is reading the file too)
            */

            if ( lock ok )
            {
                reader_counter++;
                unlock( 2nd half of inode );
                enter file descriptor in list with READ permission;
                return ok;
            }
            else
            {
                return failure; // a single writer exist (whole inode locked)
            }
        }
        else if WRITE permission requested (WRITE flags on open)
        {
            lock ( whole inode );
            /* I know there are no others accessing the same file */
            if ( lock ok )
            {
               return ok;
            }
            else
            {
               return failure; // another writer/reader exists (part of inode locked)
            }
        }
}//open
```

FIGURE 3.5: Open File or Directory

```
close()
{
        if WRITER
        {
            unlock( whole inode );
            return;
        }
        else if READER
        {
               /* 2 algorithms: using many single-byte locks OR
   using a reader counter in the inode. */

                lock ( 2nd half of inode );
                if ( reader_counter == 0 )
                    unlock( whole inode ); // I'm the last reader...
                else
                {
                    reader_counter--; // in the inode
                    unlock( 2nd half of inode );
                }
        }
}//close
```

FIGURE 3.6: Close File or Directory

3.5 Preliminary Results

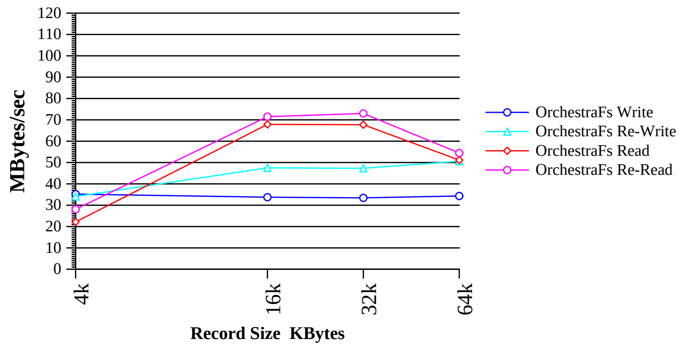
In this section we examine the overheads associated with *Orchestra*-fs. Our evaluation platform for *Orchestra-fs* is a cluster of commodity x86-based Linux systems. All the cluster nodes are equipped with dual AMD Opteron 242 CPUs and 1 GByte of RAM. Storage nodes have two Western Digital WDC WD800BB-00CAA1 ATA Hard Disks with 80 GByte capacity and 2 MByte cache. All nodes are connected both with a 100 Mbit/s (Intel 82551 adapter) and a 1 Gbit/s (Broadcom Tigon3 adapter) Ethernet network. The nodes we use in our experiments are connected through a single 48-port GigE switch. The 100 Mbit network is used only for management purposes. All systems run Fedora Core 3 Linux with the 2.6.12-1.1378_FC3smp kernel.

We use IOzone [36] to examine the basic overheads in *Orchestra*-fs. *IOzon*e is a file system benchmark tool that generates and measures a variety of file operations. We use *IOzone* version 3.242 to study file I/O performance for the following workloads: Read, write, re-read, and re-write. We vary block size between 4 and 64 KBytes and we use a file size of 3 GBytes. We choose this file size in order to minimize the impact of the buffer cache to the results of *IOzone*. Specifically, *IOzone* first creates and writes sequentially a file of the specified size. It then writes the file once more starting from the beginning. When writes are completed *IOzone* reads the file from the beginning and then it reads it again. The size of every read and write operation is the same and it is specified prior running *IOzone*.

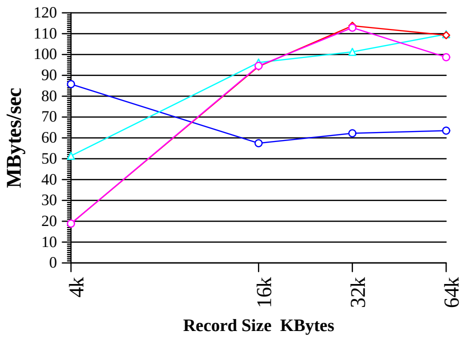
The first setup we use for our experiments is a single storage node with one disk. In the first experiment we run IOzone above *Orchestra-fs* over the physical disk. In the second experiment we run the *IOzone* on *Orchestra-fs*, using the *Orchestra* block layer on top of the disk. We use *Orchestra* layer in order to identify the overheads that *Orchestra* incurs in the local system.

The second setup we use for our experiments uses two disks configured in RAID-0 mode with 64 KByte stripe size. We use RAID-0 mode in order to have the maximum parallelism we can get. In the case of the physical RAID device without the *Orchestra* driver we use the linux MD driver [16] while in the second case we use *Orchestra*'s RAID-0 module. We perform the experiment with two disks in order to see whether *Orchestra-fs* exploits the presence of a larger number of disks in the system.

In all our experiments we choose to use the buffer cache of the application node in order to have a fair comparison with ext2. Otherwise, every operation of *Orchestra-fs* goes to the disk, which leads to very poor performance. The results of the above experiments are shown in Figures 3.7 and 3.8. These results are an estimation of the base line performance of *Orchestra-fs* and they show that in the local setup when we use two disks, configured in RAID-0, the performance of *IOzone* is approximately doubled compared to the single disk case for all operations and request sizes. We also find that *Orchestra* incurs very low overhead in the local system as the results, when we run *Orchestra-fs* above physical disk and *Orchestra-fs* above *Orchestra*, are similar.



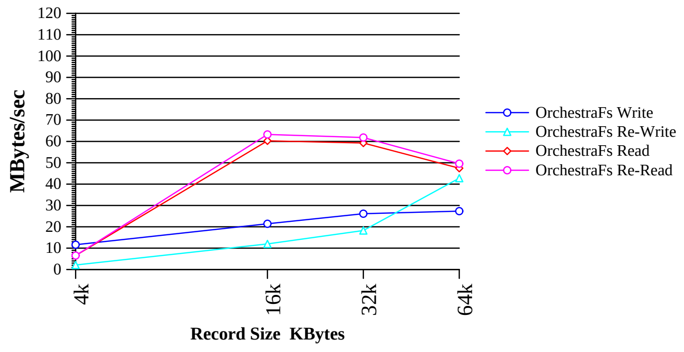
(a) Local confi guration over one physical disk



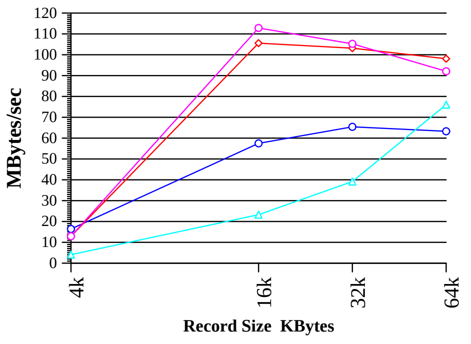
(b) Local confi guration over one physical RAID-0 disk

FIGURE 3.7: IOzone (*Orchestra*-fs) throughput for the local case above physical disk.

Next, we present three additional setups with multiple nodes. We vary the number of storage

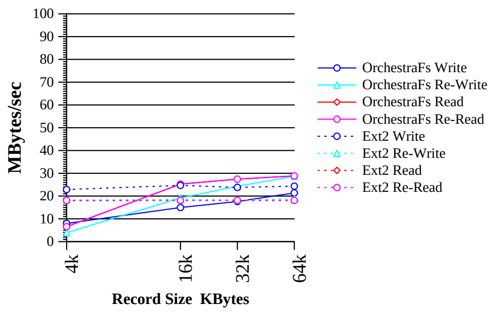


(a) Local confi guration over an *Orchestra* over one disk

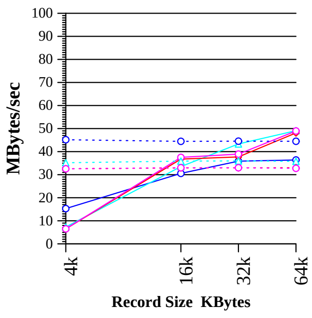


(b) Local confi guration over an *Orchestra* device with *Orchestra* confi gured in RAID-0 mode

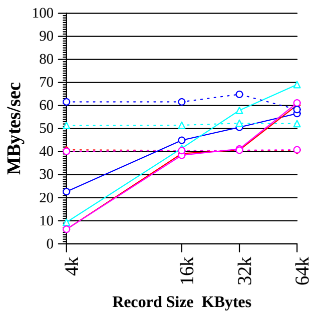
FIGURE 3.8: IOzone (*Orchestra*-fs) throughput for the local case using two devices combined in RAID-0 mode



(a) One storage server



(b) Two storage servers



(c) Three storage servers

FIGURE 3.9: IOzone (*Orchestra*-fs) throughput for the 1, 2 and, 3 storage servers.

servers between 1, 2, and 3. We name these setups 1x1, 2x1, and 3x1 respectively. For these setups we create a single volume over all available system storage, which is two disks per storage node. This volume uses the locking, allocation, and RAID0 modules. For every setup we use only one application node. This node uses the exported device to create a file system and then runs an instance of IOzone. In these setups we compare *Orchestra-fs* with ext2, a widely used filesystem. Figure 3.9 shows the performance of *IOzone* for these setups.

The results show that in the 1x1 configuration the performance of *Orchestra-fs* drops dramatically compared to the local case. For the local setup the write throughput is 60 MB/sec and the read throughput is 100 MB/sec while for the 1x1 the write throughput is 20 MB/sec and the read throughput is 30 MB/sec. We attribute this to the fact that *Orchestra-fs* issues requests with size

4KB even when the request size is larger. This means that for request sizes of 64 KB *Orchestra-fs* sends segments of 4 KB over the network. Since *Orchestra-fs* can issue only a single outstanding request, it is bound by the latency of the network, which becomes the bottleneck for the *IOzone* performance. The results also show that *Orchestra-fs* performs better than *ext2* for request sizes greater than 16KB for read, re-read, and re-write operations. This isn't the case for the write operation as *ext2* is slightly better than *Orchestra-fs* for all the request sizes. An important conclusion from our experiments is that when we vary the number of storage nodes the performance of *IOzone* scales as the number of storage nodes grows. For two storage servers the writes throughput is 35 MB/sec and the read throughput is 40 MB/sec, while for three storage servers the write throughput is 50 MB/sec and the read throughput is 50 MB/sec. This means that *Orchestra-fs* efficiently exploits the scalability that *Orchestra* provides in the case there are no sharing conflicts.

3.6 Related Work

The authors in [40] discuss how storage systems may be built out of commodity storage nodes and interconnects. However, their focus is mechanisms that improve system reliability for enterprise environments. They use a voting mechanism and erasure codes for dealing with failures as well as techniques for background recovery to avoid high overheads. We share similar objectives in enabling block-level storage technologies based on commodity components. However, our work examines how such, future block-level systems can be tailored to support changing application needs without compromising transparency and scale.

Our work bears similarity with the Petal-Frangipani approach [29, 47] in that all filesystem communication happens through a distributed volume layer, simplifying filesystem design and implementation. However, contrary to Frangipani which uses a separate lock server and allocates blocks through the FS, *Orchestra* performs locking as well as block allocation through the block layer. Moreover, *Orchestra* allows storage systems to provide varying functionality through virtual hierarchies as well provides flexibility and control in distributing many storage layers to a number of storage nodes.

Object-based storage devices (OSD) is a recent effort to improve the efficiency and scalability of storage systems. The OSD approach defines an object structure that is understood by the storage devices, and which may be implemented at various systems components, e.g. the storage controller or the disk itself. Our approach does not specify fixed groupings of blocks, e.g. to objects. Instead, it allows virtual modules to use metadata and define block groupings dynamically, based on the module semantics. Moreover, these groupings and associations may happen at any layer in the storage hierarchy. For instance, a versioning virtual device [19] that is inserted either at the application server or storage node hierarchy, specifies through its metadata which blocks form each version of a specific device. In both approaches, allocation happens closer to the blocks. In OSD objects are allocated by the storage device, whereas in *Orchestra* by a virtual module that is inserted in the storage hierarchy. In terms of locking, certain aspects are not finalized yet with OSD [52], however, object attributes may be used to implement mutual exclusion mechanisms. In *Orchestra*, similarly to allocation, locking is provided by a new virtual module that is inserted to the storage hierarchy on demand. OSD specifies that object devices perform protection checking, whereas in *Orchestra* we envision that, beyond traditional permissions in the filesystem, protection will also be provided by a virtual module at fine granularities. Finally, the OSD approach is expected to allow implementing advanced storage functionality, such as support for collections of objects, multi-object operations, snapshots, cloning, and space-management [24]. When implemented close to the object device, such functionality can distribute the work that today takes place in the centralized file-server, resulting in improved performance and simplified management [24]. *Orchestra* shares the same vision and provides a systematic framework for enabling such extensions at the block-level.

Although our approach shares similarities with work in modular and extensible filesystems [22, 41, 43, 56] and network protocol stacks [28, 34, 37, 50], existing techniques from these areas are not directly applicable to block-level storage virtualization. A fundamental difference from network stacks is that the latter are essentially stateless, except for configuration information, and packets are ephemeral, whereas storage blocks and their associated metadata need to persist. Compared to extensible filesystems, our work targets clustered storage systems and presents a single system image at the block-layer.

Distributed file systems such as the Global File System (GFS) [44] and the General Parallel File System (GPFS) [42] are used extensively today in medium and large scale storage systems. However, the complexity of the distributed file system prohibits any practical extensions to the underlying storage system and forces all applications to use a single, almost fixed, view of the available storage. In our work we examine how such issues can be addressed in future storage systems.

The most popular virtualization software is volume managers. The two most advanced open-source volume managers currently are EVMS and GEOM. EVMS [18], is a user-level distributed volume manager for Linux. It uses the MD [16] and device-mapper kernel modules to support user-level plugins called *features*. The most recent version does offer persistent metadata and block remapping primitives to these plugins. However, from the EVMS documentation of the feature API or metadata support we could not determine if EVMS can support generic extensions, such as versioning, as *Orchestra* does. GEOM [21] is a stackable BIO subsystem under development for FreeBSD. The concepts behind it GEOM are, to our knowledge, the closest to *Orchestra*. However, GEOM does not support persistent metadata which, combined with dynamic block mapping are necessary for advanced modules such as versioning [19]. LVM [46] and Vinum [30] are simpler versions of EVMS and GEOM. *Orchestra* has all the configuration and flexibility features of a volume manager coupled with the ability to write extension modules with arbitrary virtualization semantics.

Finally, besides open-source software, there exist numerous commercial virtualization solutions as well, such as HP OpenView Storage Node Manager [23], EMC Enginuity [17], Veritas Volume Manager [51] and Veritas File System. However, in all cases, the offered virtualization functions are predefined and they do not seem to support extensibility of the I/O stack with new features.

3.7 Limitations

Although *Orchestra-fs* provides most of the functionality a single filesystem should support, there are still functionalities that are not implemented. Such functionality is the behavior of *Orchestra-fs* in the presence of failures. Many modern filesystems deal with failures using a technique called “journaling”. According to this technique, the filesystem holds a log area into the storage media logging information for every operation it performs. Prior to every filesystem operation, the log is updated. When the filesystem is mounted it checks its state. In the case where its state is inconsistent, it consults the log in order to recover. Although this technique is really useful, it is beyond the scope of this work.

3.8 Summary

To provide access to *Orchestra* volumes through a file system API, we build *Orchestra*-fs a *stateless, pass-through* file system that is currently implemented as a dynamically linked user-level library. *Orchestra*-fs does not maintain internal state, instead it uses *Orchestra* facilities for locking and block-allocation, and does not require explicit communication among its instances.

We implement *Orchestra*-fs under Linux and evaluate it using various setups with single application node and multiple storage nodes. We run *IOzone* over various setups and we measure the base-line performance of *Orchestra-fs*. We find that in the local setup when we use two disks instead of one the performance of *IOzone* is approximately doubled for all operations and request sizes. From the results we can also conclude that *Orchestra* incurs very low overhead in the system. We also find that when we vary the number of storage nodes the performance of *IOzone* scales as the number of storage nodes grows. We also see that in the 1x1 setup the performance of *Orchestra-fs* is lower than the local setups, due to the latency of the network. Finally, we find that in the multiple storage servers setup the performance of *Orchestra-fs* and *ext2* are comparable.

Chapter 4

Conclusions

In this work, we evaluate new architectures made of large numbers of commodity storage "bricks", accessed in parallel by many (commodity) application servers. There are two main challenges to achieve this.

The first is how we can efficiently access stored data in networked storage systems. Usually, this happens at the block layer. One way to provide transparent access to data is by using the iSCSI protocol. However, it is not clear which is the impact of iSCSI on overall system and application performance. This lack of understanding prevents us from optimizing the performance of such systems. Thus, in this work we first accurately evaluate the costs of each layer in the I/O protocol stack. To do this we implement a framework of high-accuracy timers in kernel-space for measuring kernel code paths and we use it in order to get a detailed breakdown of the system time spent by applications (TPC-H, Postmark, SPEC-SFS) that run over iSCSI. Our examination of kernel-level overheads shows that improving I/O path performance requires dealing not only with TCP and interrupt processing costs, but buffer cache management as well.

We also examine the behavior of benchmarks (Postmark, TPC-H, Spec-SFS, IOmeter) that run on top of iSCSI at the initiator side. We experiment with three setups: Direct, iSCSI, and iSCSIx3. In summary, we see that using iSCSI without increasing system resources compared to a local configuration has a significant impact in all applications we examine. However, the impact of iSCSI differs in each case. Postmark is sensitive to increased I/O latency, TPC-H is affected by reduced I/O throughput and increased CPU cycles, and Spec-SFS by the sharing of one network between both client-server as well as iSCSI I/O traffic. iSCSIx3 is able to scale system resources and recover and in some cases improve system performance. Postmark benefits from the increased target buffer cache, TPC-H by the increased number of disks (and to a lesser extend by the increased buffer cache), whereas Spec-SFS remains limited by the available network bandwidth in the unified interconnect.

The second challenge is how can multiple applications share a distributed storage system. Traditionally, such sharing happens at the file system level. For this reason, today there is a significant interest in distributed file systems that scale to large numbers of nodes. In our work we propose using a *simple, stateless* file-system that relies on the advanced features of *Orchestra*. Thus, we design and implement a file system that sits above *Orchestra*, in order to provide sharing to the applications. We experiment with various setups: Local setup with one disk and two disks, and with multiple storage nodes and a single application node varying the number of storage nodes between one and three. We find that in the local setup the performance of *IOzone* scales for all operations and request sizes, when increasing the number of disks. We also find that when we vary the number of storage nodes the performance of *IOzone* scales as the number of storage nodes grows. However, in the 1x1 basic setup the performance of *Orchestra-fs* is lower than the local setups, due to the latency of the network. Finally, we find that in the multiple storage servers setup the performance of *Orchestra-fs* and *ext2* are comparable.

Overall, our results show that building next generation, network-based I/O architectures, requires optimizing I/O latency, reducing network and buffer cache related processing in the host CPU, and increasing the sheer network bandwidth to account for consolidation of different types of traffic. Finally, based on our results, we believe that offloading the functionality from the filesystem layer towards the block layer will both simplify the design of the filesystem as well as improve scalability.

Bibliography

[1] http:// www.linux.com / howtos / KernelAnalysis-HOWTO-6.shtml.

[2] Ext2 filesystem. e2fsprogs.sourceforge.net/ext2.html.

[3] Intel vtune performance analyzers. http:// www.intel.com/ software/ products/ vtune.

[4] Linux kernel sources version 2.4.23-pre5. http://www.kernel.org.

[5] Oprof: profiling system for linux 2.2/2.4/2.6. http:// oprofile.sourceforge.net.

[6] Papi: Performance application programming interface. http:// icl.cs.utk.edu/ papi/.

[7] Project: Intel iSCSI reference implementation. http://sourceforge.net/ projects/ intel-iscsi.

[8] Scheduling in unix and linux. http:// www.kernelnewbies.org/ documents / schedule /.

[9] Unix Top. http:// sourceforge.net/ projects/ unixtop.

[10] Stephen Aiken, Dirk Grunwald, and Jesse Willeke Andrew R. Pleszkun. A performance analysis of the iSCSI protocol. In *11th NASA Goddard, 20st IEEE Conference on Mass Storage Systems and Technologies (MSST2003)*, April 2003.

[11] ANSI. Fibre Channel Protocol (FCP), X3.269:1996. In *11 West 42nd Street, 13th Floor, New York, NY 10036*.

[12] ANSI. SCSI-3 Architecture Model (SAM), X3.270:1996.

[13] Michael F. Brown, Josh Hawkins, Mike Ostman, and William Moloney. UMass Lowell iSCSI Project. http://www.cs.uml.edu/ mbrown/iSCSI.

[14] Anshul Chadda, Ashish Palekar, Robert Russell, and Narendran Ganapathy. Design, implementation, and performance analysis of the iSCSI protocol for SCSI over TCP/IP. In *Internetworking 2003 International Conference*, June 2003.

[15] Ismail Dalgic, Kadir Ozdemir, Rajkumar Velpuri, and Umesh Kukreja. Comparative performance evaluation of iSCSI protocol over metro, local, and wide area networks. In *12th NASA Goddard & 21st IEEE Conference on Mass Storage Systems and Technologies (MSST2004)*, April 2004.

[16] Miguel de Icaza, Ingo Molnar, and Gadi Oxman. The linux raid-1,-4,-5 code. In *LinuxExpo*, April 1997.

[17] EMC. Enginuity(TM): The Storage Platform Operating Environment (White Paper). http://www.emc.com/pdf/techlib/c1033.pdf.

[18] Enterprise Volume Management System. evms.sourceforge.net.

[19] Michail D. Flouris and Angelos Bilas. Violin: A Framework for Extensible Block-level Storage. In *Proceedings of 13th IEEE/NASA Goddard (MSST2005) Conference on Mass Storage Systems and Technologies*, Monterey, CA, April 11–14 2005.

[20] Michail D. Flouris, Dimitrios Xinidis, Renaud Lachaize, and Anglelos Bilas. Orchestra: Scalable support for shared extensible virtual block devices. DRAFT: submitted for publication, October 2005.

[21] FreeBSD: GEOM Modular Disk I/O Request Transformation Framework. http://kerneltrap.org/node/view/454.

[22] J.S. Heidemann and G.J. Popek. File System Development with Stackable Layers. *ACM Transactions on Computer Systems*, 12(1):58–89, February 1994.

[23] HP. OpenView Storage Area Manager. http://h18006. www1.hp.com/ products/ storage/ software/ sam/ index.html.

[24] IBM. Object store project. http://www.haifa.il.ibm.com/projects/storage/objectstore/overview.html.

[25] Internet Engineering Task Force (IETF). iSCSI, version 08, Sept. 2001.

[26] Iometer team. Iometer: The I/O Performance Analysis Tool. http://www.iometer.org.

[27] Jeffrey Katcher. PostMark: A New File System Benchmark. http://www.netapp.com/tech_library/ 3022.html.

[28] Eddie Kohler, Robert Morris, Benjie Chen, John Jannotti, and M. Frans Kaashoek. The Click modular router. *ACM Transactions on Computer Systems*, 18(3):263–297, August 2000.

[29] Edward K. Lee and Chandramohan A. Thekkath. Petal: Distributed virtual disks. In *Proc. of The 7th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS7)*, pages 84–92, Cambridge, MA, October 1996.

[30] Greg Lehey. The Vinum Volume Manager. In *Proceedings of the FREENIX Track (FREENIX-99)*, pages 57–68, Berkeley, CA, June 6–11 1999. USENIX Association.

[31] Yingping Lu, Farrukh, Noman, and David H.C. Du. Simulation study of iSCSI-based storage system. In *12th NASA Goddard & 21st IEEE Conference on Mass Storage Systems and Technologies (MSST2004)*, April 2004.

[32] Microsoft.com. Monitoring and Tuning System Performance in Microsoft Windows XP. http:// support.microsoft.com/ kb/823887.

[33] Mike Brim and George Kola. An analysis of iSCSI for use in distributed file system design. http://www.cs.wisc.edu/m̃jbrim/uw/740/paper.pdf.

[34] David Mosberger and Larry L. Peterson. Making Paths Explicit in the Scout Operating System. In *Proc. of the 2nd USENIX Symposium on Operating Systems Design and Impl. (OSDI96)*, October 28–31 1996.

[35] Wee Teck Ng, Hao Sun, Bruce Hillyer, Elizabeth Shriver, Eran Gabber, and Banu Ozden. Obtaining high performance for storage outsourcing. In *Proc. of the 1st USENIX Conference on File and Storage Technologies (FAST02)*, pages 145–158, January 2002.

[36] William D. Norcott and Don Capps. IOzone Filesystem Benchmark. http://www.iozone.org.

[37] S. W. O'Malley and L. L. Peterson. A dynamic network architecture. *ACM Transactions on Computer Systems*, 10(2):110–143, May 1992.

[38] Ashish A. Palekar and Robert D. Russell. Design and implementation of a SCSI target for storage area networks. Technical Report TR 01-01, University of New Hampshire, May 2001.

[39] David A Patterson, Garth Gibson, and Randy H Katz. A Case for Redundant Arrays of Inexpensive Disks (RAID). ACM, 1988.

[40] Y. Saito, S. Frolund, A. Veitch, A. Merchant, and S. Spence. FAB: Enterprise storage systems on a shoestring. In *Proc. of the ASPLOS 2004*, October 2004.

[41] Paul W. Schermerhorn, Robert J. Minerick, Peter W. Rijks, and Vincent W. Freeh. User-level Extensibility in the Mona File System. In *Proc. of Freenix 2001*, pages 173–184, June 2001.

[42] Frank Schmuck and Roger Haskin. GPFS: A Shared-disk File System for Large Computing Centers. In *USENIX Conference on File and Storage Technologies*, pages 231–244, Monterey, CA, January 2002.

[43] Glenn C. Skinner and Thomas K. Wong. Stacking/ vnodes: A progress report. In *Proc. of the USENIX Summer 1993 Technical Conference*, pages 161–174, Berkeley, CA, USA, June 1993. USENIX Association.

[44] S. Soltis, G. Erickson, K. Preslan, M. O'Keefe, and T. Ruwart. The Global File System: A File System for Shared Disk Storage, October 1997.

[45] Standard Performance Evaluation Corporation (SPEC). SFS 3.0. http://www.spec.org/sfs97r1/docs/sfs-3.0.pdf, 2001.

[46] David Teigland and Heinz Mauelshagen. Volume managers in linux. In *Proceedings of USENIX 2001 Technical Conference*, June 2001.

[47] Chandramohan A. Thekkath, Timothy Mann, and Edward K. Lee. Frangipani: A Scalable Distributed File System. In *Proc. of the 16th Symposium on Operating Systems Principles (SOSP-97)*, pages 224–237, October 5–8 1997.

[48] Fujita Tomonori and Ogawara Masanori. Performance optimized software implementation of iSCSI. In *SNAPPI03*, September 2004.

[49] Transaction Processing Performance Council (TPC). *TPC BENCHMARK H, Standard Specification, Revision 2.1.0*. 777 N. First Street, Suite 600, San Jose, CA 95112-6311, USA, August 2003.

[50] Robbert van Renesse, Kenneth P. Birman, Roy Friedman, Mark Hayden, and David A. Karr. A framework for protocol composition in horus. In *Symposium on Principles of Distributed Computing*, pages 80–89, 1995.

[51] Veritas. Volume Manager(TM). http://www.veritas.com/vmguided.

[52] Ralph O. Weber (Editor). Information technology – scsi object-based storage device commands (OSD), revision 10. Technical Council Proposal Document T10/1355-D, Technical Committee T10, July 2004.

[53] Michael Widenius and David Axmark. *MySQL Reference Manual*. O'Reilly & Associates, Inc., June 2002.

[54] Dimitrios Xinidis, Michail D. Flouris, and Angelos Bilas. Performance Evaluation of Commodity iSCSI-based Storage Systems. In *13th NASA Goddard, IEEE Conference on Mass Storage Systems and Technologies (MSST2005)*, April 2005.

[55] Dimitrios Xinidis, Michail D. Flouris, and Angelos Bilas. Virtual Timers: Using Hardware Physical Timers for Profiling Kernel Code-Paths. In *Proc. of 8th Workshop on Computer Architecture Evaluation using Commercial Workloads (CAECW-8)*, February 2005.

[56] Erez Zadok and Jason Nieh. FiST: A Language for Stackable File Systems. In *Proceedings of the 2000 USENIX Annual Technical Conference (USENIX-00)*, pages 55–70, Berkeley, CA, June 18–23 2000. USENIX Association.