ANALYZING STORAGE SYSTEM WORKLOADS

A DISSERTATION
SUBMITTED TO THE DEPARTMENT OF COMPUTER SCIENCE,
FACULTY OF SCIENCE AT THE UNIVERSITY OF CAPE TOWN
IN FULFILLMENT OF THE REQUIREMENTS
FOR THE DEGREE OF
MASTER OF SCIENCE BY COURSEWORK AND DISSERTATION

by
Paul Sikalinda
January 9, 2006

Supervised by
Prof. Pieter Kritzinger

Abstract

Analysis of storage system workloads is important for a number of reasons. The analysis might be performed to understand the usage patterns of existing storage systems. It is very important for the architects to understand the usage patterns when designing and developing a new, or improving upon the existing design of a storage system. It is also important for a system administrator to understand the usage patterns when configuring and tuning a storage system. The analysis might also be performed to determine the relationship between any two given workloads. Before a decision is taken to pool storage resources to increase the throughput, there is need to establish whether the different workloads involved are correlated or not. Furthermore, the analysis of storage system workloads can be done to monitor the usage and to understand the storage requirements and behavior of system and application software. Another very important reason for analyzing storage system workloads, is the need to come up with correct workload models for storage system evaluation. For the evaluation, based on simulations or otherwise, to be reliable, one has to analyze, understand and correctly model the workloads.

In our work we have developed a general tool, called ESSWA (Enterprize Storage System Workload Analyzer) for analyzing storage system workloads, which has a number of advantages over other storage system workload analyzers described in literature. Given a storage system workload in the form of an I/O trace file containing data for the workload parameters, ESSWA gives statistics of the data. From the statistics one can derive mathematical models in the form of probability distribution functions for the workload parameters. The statistics and mathematical models describe *only* the particular workload for which they are produced. This is because storage system workload characteristics are sensitive to the file system and buffer pool design and implementation, so that the results of any analysis are less broadly applicable. We experimented with ESSWA by analyzing storage system workloads represented by three sets of I/O traces at our disposal.

Our results, among other things show that: I/O request sizes are influenced by the operating system in use; the start addresses of I/O requests are somewhat influenced by the application; and the *exponential* probability density function, which is often used in simulation of storage systems to generate inter-arrival times of I/O requests, is not the best model for that purpose in the workloads that we analyzed. We found the *Weibull*, *lognormal* and *beta* probability density functions to be better models.

Acknowledgement

First and foremost, I would like to thank my *Heavenly Father* who created the heavens and the earth for providing all the resources that I needed to undertake this study. I would also like to express my thanks and appreciation to the following people whose contribution to this work is invaluable.

- My wife, Miyoba Choonga, for her support, love and prayers for me, and also for being patient with me while pursuing my studies.
- My supervisor, Professor Pieter S. Kritzinger, for his guidance, supervision and for suggesting the topic which developed into this dissertation.
- Data Network Architectures (DNA) Research Group for taking me on and for the financial assistance rendered to me during my studies.
- Mr. L. Walters for his help in finding the right tools, such as the R programming language, and providing and explaining the code for calculating the λ^2 discrepancy statistic used in this work.
- Mr. J. Kalonga and Mr. Y. Yavwa for their unselfish continued support which made it possible for me to take study leave from my employer and undertake the studies part of which is this work.
- Sameshan, Alapan, Colette, Jesse, Ben, Nico, Reinhardt, Victor, Samuel, James and the rest of my fellow masters students who have been with me during my studies for their friendship which provided a fun and pleasant environment.
- System Administrators, S. Chetty and M. West, for continually keeping the computer systems running while I was doing this work.
- Evans, Emmanuel, David, Joe, my *other* family members and all my brothers and sisters in Christ for praying for me during the course of my studies.
- Godfrey and Sebastian for residing peacefully with me in the same room during the first and second years of my studies respectively.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

This dissertation is about analyzing storage system workloads in the form of I/O traces. In this work we developed a tool for carrying out the analysis and used it to analyze three workloads.

In Section 1.1 of this chapter, we state the main parameters that define storage system workloads. In Section 1.2 we give the objectives and motivation for our study by stating what we hoped to achieve and the reasons why we pursued this study. Before analyzing any workload it is obviously important to understand the system whose workload is being analyzed. Therefore, in Section 1.3 we describe high performance storage systems classified as enterprize storage systems (ESSs) which have become prevalent in recent years. In this section we also discuss the disk drive, which is an indispensable component of storage systems.

1.1 Storage System Workload Parameters

A storage system workload consists of I/O requests issued to a storage system over a given period of time and is mainly described by the following parameters for each I/O request:

- logical volume[1] number,
- start address,
- request size,
- operation type (i.e., read or write) and

[1] A logical volume is simply a group of information located on fixed-disk drives, called the physical volumes. Data on logical volume(s) appear to be contiguous to the user but can be discontiguous on the physical volumes.

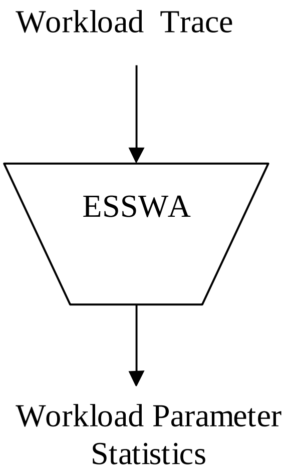


Figure 1.1: Storage system workload analyzer model

- timestamp.

These parameters are described fully in Section 2.1. We derived three more parameters from these basic parameters.

- *Inter-arrival time:* The inter-arrival time is the difference between any two consecutive timestamps.
- *Logical seek distance:* We defined the logical seek distance as the difference between any two consecutive start addresses.
- *Parallelism degree:* We defined the parallelism degree as the number of I/O requests with the same timestamp.

1.2 Objectives and Motivation

In this section we first state the objectives and then the motivation for our study.

1.2.1 Objectives

In our work we intended to:

- *develop* a general tool for analyzing storage system workloads that is better than the tools which are currently described in literature and
- *analyze* some storage system workloads.

We developed the tool and called it ESSWA (Enterprize Storage System Workload Analyzer). Given a storage system workload in the form of an I/O trace file containing data for the workload parameters listed in Section 1.1, ESSWA gives statistics of the data. From the statistics one can derive mathematical models, in the form of probability distribution functions, for the parameters. The statistics and mathematical models describe *only* the particular workload for which they are produced. This is because storage system workload characteristics are sensitive to the file system design and implementation, so that the results of any analysis are less broadly applicable[Hsco03]. ESSWA is also capable of displaying some of the results visually using Empiric Cumulative Distribution Functions and histograms. Chapter 5 describes ESSWA in detail. In summary, ESSWA is a machine that takes I/O traces as input and produces statistics for workload parameters as output as illustrated in Figure 1.1.

We analyzed storage system workloads represented by three sets of I/O traces, two of which are publicly available with the courtesy of the Storage Performance Council (SPC)[2]. The third set is from the HP I/O trace repository[3]. Some of the results we obtained are presented and discussed in Chapter 6.

1.2.2 Motivation

We pursued the work described in this dissertation because of the importance of analyzing storage system workloads and the need for a better storage system workload analysis tool as explained in the following sections.

A. Importance of Workload Analysis

A lot of research effort is being spent on the development of ESSs. This is because disk storage subsystems have not kept up the speed with processors. Processor performance has been increasing at a much higher rate than that of disk drives. Therefore, the I/O subsystem has become a bottleneck in today's computer systems. The situation is made worse with the proliferation of applications which involve large volume of data stored on disks. Such applications include data warehousing, image processing, digital video editing, transaction processing, decision support systems, scientific and engineering simulations, etc. Having realized this problem, the research community is looking into ways of improving the I/O subsystem. IBM and HP are among the organizations doing research and development of ESSs.

In the development of storage systems such as ESSs, workload analysis is important mainly because of two reasons.

[2] http://www.storageperformance.org/home.

[3] http://tesla.hpl.hp.com/public_software.

- **Understanding usage patterns:** The analysis might be performed to understand the usage patterns of existing storage systems. It is very important for the architects to understand the usage patterns when designing and developing a new, or improving upon the existing design of a storage system[Drac00]. Understanding the usage patterns will help in making certain decisions. For example, to come up with a reasonable size of the data access unit in a new storage system design it is very important to know the sizes of I/O requests in the existing systems.

- **Modelling storage system workloads:** Part of the effort in the development of high performance storage systems goes into the evaluation of these systems in terms of design, correctness and performance. For the evaluation, based on simulations or otherwise, to be reliable, one has to *analyze*, understand and correctly model the workloads. Currently there is a need to come up with correct workload models for storage system evaluation. Ganger[Gang95] found that the *commonly used workload models (e.g., uniform distribution function for start addresses, exponential distribution function for inter-arrival times) are inappropriate and can produce dramatically incorrect results.* For inter-arrival times, Hsu *et al*[Hsco03] agrees with Ganger that the exponential probability density function is not always the proper model. Hsu *et al* found the lognormal probability density function to be a better model for the inter-arrival times in the workloads they analyzed.

  In our findings, we discovered that no one has attempted to analyze *access patterns* in storage system workloads with the aim of finding better models for the following parameters:

  - Logical volume number,
  - Request size,
  - Start addresses and
  - I/O parallelism degree (i.e., the number of I/O requests issued at the same time).

  In our work we made attempts to find better models for the parameters which describe both the arrival and access patterns, some of which are not currently modelled correctly.

- **Other reasons:** There are other reasons for analyzing storage system workloads. A system administrator can perform the analysis to understand the usage patterns. It is important for him to understand the usage patterns when configuring and tuning a storage system. The analysis might also be performed to determine the relationship between any two given workloads. Before a decision is taken to pool storage resources to increase the throughput, there is need to establish whether the different workloads involved are correlated or not. Furthermore, analysis of storage system workloads can be done to monitor the usage and to understand the storage requirements and behavior of system and application software. The reasons for performing storage system workload analysis in general are discussed in depth in Section 2.3.2.

B. Need for a Better Workload Analyzer

A number of software tools have been developed and described in literature for storage system workload analysis. These tools read sequences of I/O trace records, perform some analysis on them, and output the results. However, each one of these tools has one or more drawbacks. Hence the need for a better tool. Some of these tools:

- are not user-friendly,
- are not flexible (i.e., configurable),
- are not extendable,
- have limited reporting formats and
- have many separate analysis and many separate I/O trace manipulation programs which are difficult to maintain.

These drawbacks are discussed further in Section 5.1.1.

1.3 Storage Systems

Before one can analyze and model the workload of a system, one has to know what it is and how it functions. The aim of the following sections is to help the reader understand what storage systems are and how they function. For now we can say that a storage system is a system that stores data on one or more disks which are accessed by the operating system.

In this section we explain the various components that make up storage systems that fall in the category of ESSs. But before we do this, we first describe how the application software accesses the data stored in a disk system and then explain what makes ESSs different from other disk systems.

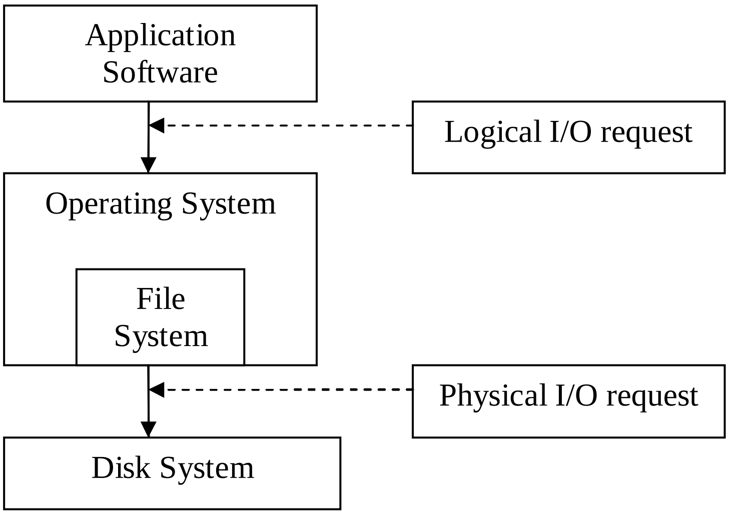


Figure 1.2: The routing of an I/O request from the application to the storage system

1.3.1 I/O Request Servicing

In a computer system, software applications and the hardware are interfaced by a large, relatively complex, low-level piece of software called the *operating system.* Therefore, a software application issues I/O requests to the operating system to access the data in a disk system. When the operating system receives the request, it uses its *file system* to access the data on the disk. The file system is a system that an operating system uses to access, organize and keep track of files on disk[Dpjh98].

A file system sends I/O requests to the disk system to retrieve or store data. Each I/O request to the disk system consists of at least the following parameters: logical volume number, logical block[4] address (i.e., start address), request size and operation type. When the disk system receives the request it maps the logical volume number and the logical block address into the physical disk drive and physical block address respectively to service the request. Next it writes or reads the data and sends the acknowledgement that the data have been written or the data being requested on the reverse path. The process of how an I/O request is routed from an application to the disk system is shown in Figure 1.2[Dpjh98].

1.3.2 Enterprize Storage Systems

ESSs are storage systems that can be defined as powerful disk storage systems with capabilities that allow them to meet the most demanding requirements of performance,

[4] A block is typically 512 bytes.

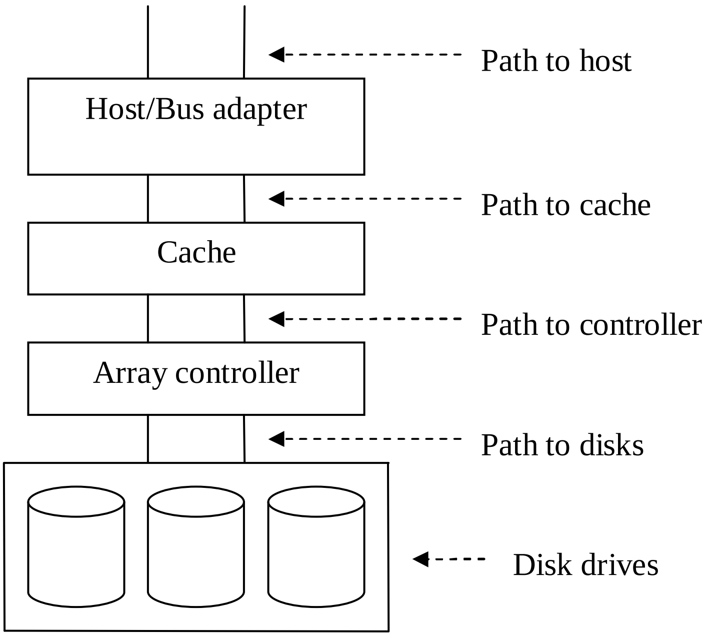


Figure 1.3: ESS components

capacity and availability that a computing business may require. An ESS comprises many disk drives defining huge amounts of storage. In addition, protection against physical drive failure can be provided through the provision of one of the many Redundant Array of Independent Disks (RAID) methods. Typically an ESS is made up of the following components (see Figure 1.3)[Sack03]:

- **Host/bus adapters:** These are interfaces used when connecting a peripheral, in this case a storage system, to a computer that does not have native support for that peripheral's interface.

- **Array controller:** The array controller consists of the hardware and the software that manages one or more arrays of disks. It is also called the disk command module, and this is where the RAID controller resides.

- **Cache:** Cache is the faster memory than disk placed between the host and the disk system which holds data that have recently been read and written and, in some cases, adjacent data blocks that are likely to be accessed next. The cache can be placed either in the array controller or disk drive.

- **Paths:** This refers to the media connecting the various components. The connections may be fibre or wires. Examples are the *Fibre Channel* and *Small Computer*

System Interface (SCSI) cables. The connections could also be a network, e.g., a storage area network (SAN).

- **Disk drives:** A disk drive is simply a machine that reads data from and writes data onto a magnetic disk. A disk drive rotates the disk very fast and has one or more heads that read and write data. Section 1.3.3 below discusses disk drives comprehensively.

An I/O request is sent from the host's file system to an *array controller* which in turn sends the request to the disk drive. The disk drive translates the logical block address into physical block address and reads the requested data. However, if the data requested are cached, then the request is served using the cached data. This improves performance because reading data from disk is slower than reading from the cache[Sack03].

1.3.3 Disk drives

A disk drive contains a disk which in turn consists of a collection of platters (1-15) called a stack illustrated in Figure 1.4[5]. Each platter has two recordable surfaces. The stack of platters is rotated at about 10,000 to 15,000 revolutions per minute and has a diameter from just over an inch to just over 8 inches. Each disk surface is divided into concentric circles, called *tracks*. There are typically 1,000 to 5,000 tracks per surface. Each track is in turn divided into *sectors* that contain the data. Each track may have 64 to 200 sectors. With the introduction of *Logical Block Access* (LBA), disk drives became addressed by blocks. All tracks aligned vertically form a *cylinder*[Dpjh98].

To access data, the first step is to position the read/write head over the proper track. This is called a *seek*, and the time to move the head to the desired track is called the *seek time* while the distance moved is called the *physical seek distance*. Average seek times are usually advertised as 8ms to 12ms, but, depending on the application and scheduling of disk requests the actual average time may be only 25 or 33% of the advertised time, because of *locality of disk references*. This locality arises both because of successive access to the same file and because the operating system tries to schedule such accesses together. Once the head has reached the correct track, we must wait for the desired sector to rotate under the read/write head. This is called the *rotational latency* or *rotational delay*. Assuming LBA's are uniformly distributed, the average latency to the desired data is half-way around the disk. The last component of the disk access, *transfer time*, is the time to transfer a block of bits. This transfer time

[5] This figure is taken from "The Linux System Administrator's Guide: Version 0.7" - http://www.faqs.org/docs/linux_admin/x1001.html.

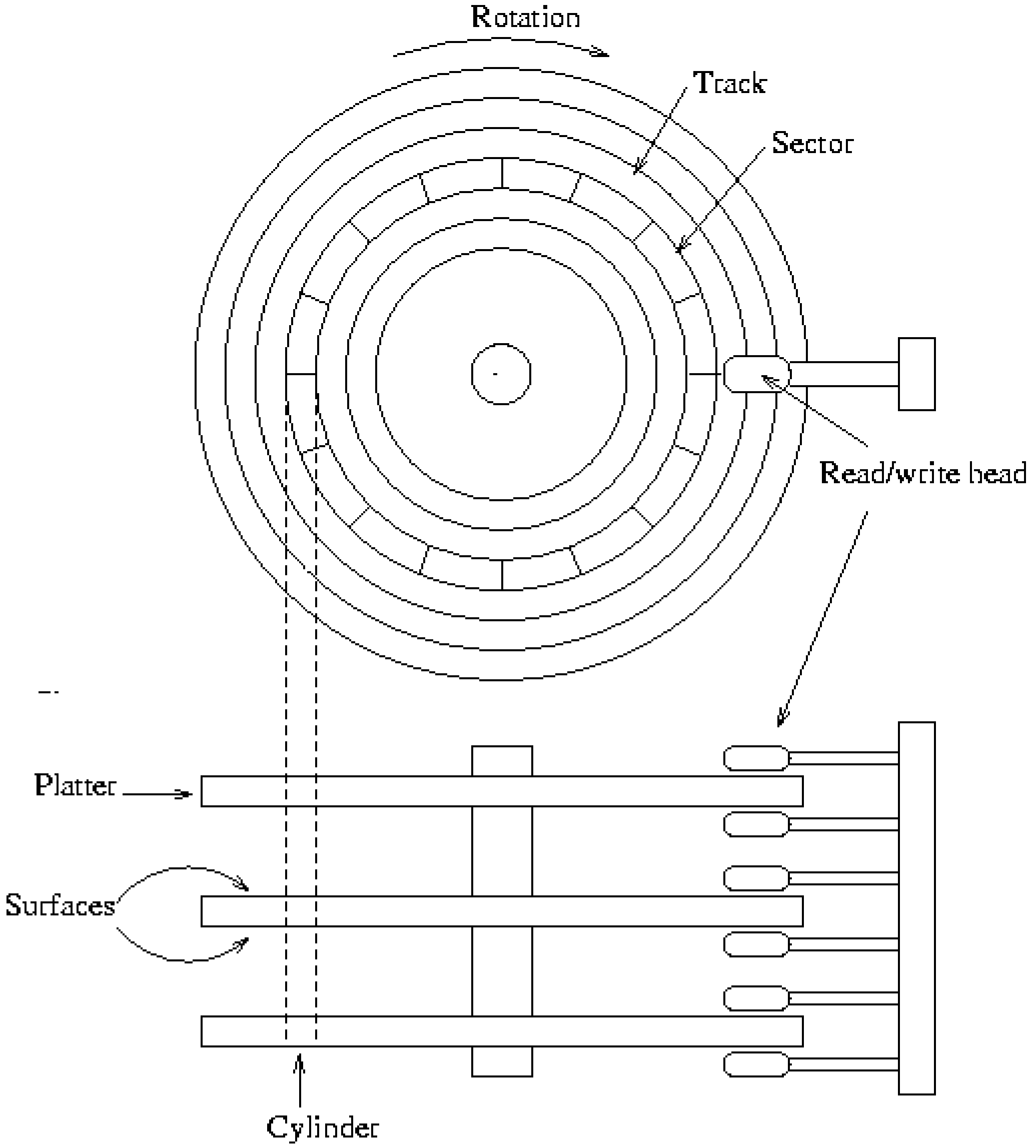


Figure 1.4: A schematic picture of a hard disk drive

is a function of sector size, the rotation speeds and the recording density of the track. The detailed control of the disk and the transfer between the disk and the memory is usually handled by a disk controller[Dpjh98].

Disk drives have been improved in terms of data density, i.e., the amount of data stored per unit area on the disk surface. The bandwidth of the path connecting the disk drives and processors has also been improved. But data access speed, which depends on the seek time, rotational latency and transfer time, has not been improved to satisfactory levels as compared to the processor speed. Therefore, the I/O subsystem has become a bottleneck in today's high performance computing systems[Dpjh98].

1.4 Dissertation Outline

The layout of the rest of this dissertation is as follows:

Chapter 2 - Storage System Workloads. This chapter explains storage system workload parameters, discusses issues concerning storage workload trace collection and lastly discusses the uses of storage system workload traces.

Chapter 3 - Related Work on Workload Analysis. In this chapter, previous studies on storage workload analysis are classified, and then two specific studies are discussed in detail. For each study, we first give the aim, then describe the I/O workload traces used and finally present some results.

Chapter 4 - Statistical Methodology. This chapter looks at the statistical techniques and tools that we implemented in ESSWA for analyzing and modelling storage system workloads. These include visual techniques (i.e., histograms and Empiric Cumulative Distribution Functions) and computations of key data statistics.

Chapter 5 - Storage System Workload Analyzers. This chapter describes the tool we developed for storage system workloads analysis. It begins with a description of a similar tool encountered in literature. At the end the two tools are compared and contrasted.

Chapter 6 - Results. This chapter discusses the results we obtained from analyzing three storage system workloads.

Chapter 7 - Conclusion and Future Work. This chapter presents some of our conclusions and suggestions for future work.

Chapter 2

Storage System Workloads

This chapter discusses storage system workloads in the form of traces. Section 2.1 lists the major parameters that describe any storage system workload and defines a typical format for trace files that contain values for these parameters. Section 2.2 discusses issues concerning storage workload trace collection and Section 2.3 discusses the uses of storage system workload traces.

2.1 Storage System Workload Parameters

As mentioned in Chapter 1, a storage system workload is typically described by the following parameters: logical volume number, start address, request size, operation type and the timestamp for each I/O request issued to the storage system over a given period of time[Stfs92]. Values for these parameters are recorded in a trace file. Therefore, for each parameter, there is a corresponding field in the trace file.

Before analyzing a storage system workload, records about individual I/O requests issued by the host processor(s) must be collected first. The relevant information about I/O requests is collected while the system is handling the workload of interest. The collection of this information is called a *trace* and the file that contains this information is called a *trace file*. Each record in the trace file represents one I/O request. The process of collecting the traces is called *tracing the system* and is usually accomplished by using hardware probes or by adding instrumentation to the system software[Wwha04]. The following sections define the fields in a trace file according to the SPC trace file format specification document[Stfs92].

2.1.1 Logical Volume

The first field in a trace file is the logical volume number and can take on positive integer values. If there are a total of n logical volumes described in the complete trace file, then the trace file must contain at least one record for each of volumes 0 through $n-1$.

2.1.2 Start Address

The second field in a trace file is the start address. The start address or the logical block address (LBA) is a positive integer that describes the logical volume block offset of the data to be transferred. The values for this field may range from 0 to $n-1$, where n is the capacity in blocks of the logical volume. There is no limit on this field, other than the restriction that the sum of the start address and the request size must be less than or equal to the capacity of the logical volume.

2.1.3 Request Size

The third field is the request size. The request size is a positive integer that describes the amount of data in bytes transferred. There is also no upper limit for this field other than the restriction that the sum of the start address and the request size must be less or equal to the capacity of the logical volume.

2.1.4 Operation Type

The fourth field is the operation type. The operation type is a flag that indicates whether the I/O request represented by a particular record in the trace file is for a read or write operation. In other words, it defines the direction of the transfer, either the data transfer is from the operating system to the storage system or vice versa. This field takes on a single, case sensitive character and the two possible values are:

- ”R”(or ”r”) to indicate a read operation.
- ”W”(or ”w”) to indicate a write operation.

2.1.5 Timestamp

The fifth field in a trace file is for the arrival times of I/O requests in the form of timestamps. The timestamp is usually a positive real number representing the offset in seconds for an I/O request recorded in the trace file from the start of the trace. The format of the field is ”s.d”, where ”s” represents the integer portion, and ”d” represents the fractional portion of the timestamp. Both the integer and fractional parts of the field are mandatory. The value of this field in a particular record must be greater than or equal to all values for the preceding records, and less than or equal to all values for succeeding records.

2.1.6 Other Fields

It is important to note that the five fields above are usually mandatory, and that there are other optional fields, which can be included in a trace file. The optional fields include: I/O request sequence number, a flag to indicate whether the request is synchronous or asynchronous, partition number, and file object pointer. Although these fields are optional, provision is made for them because the information provided by them can be worthy of a detailed analysis[Stfs92]. For the purpose of our study only the five fields discussed above were considered.

2.2 Storage Workload Trace Collection

The task of collecting storage workload traces is not simple. A number of researchers have collected traces some of which are publicly available. For example, Hsu *et al*[Hsco03] collected storage system traces from a number of personal computer (PC) and server environments, Roselli *et al*[Drac00] collected four sets of file system traces from different environments, and Ramakrishnan *et al*[Kkpr92] collected eight sets of file system traces from several production VAX/VMS computer systems. More information about some of these traces is given in Chapter 3.

There are a number of issues which one has to take into consideration when collecting storage workload traces. The following sections discuss some of the issues which we think are very important.

2.2.1 Level of Trace Collection

It is important to consider the level at which the traces are collected when performing storage workload analysis. Storage workload traces can be collected at the *logical level* where the application makes I/O requests to the file system or at the *physical level* where the file system makes I/O requests to the storage system as illustrated in Figure 2.1.

The traces collected at the logical level represent the *file system* workloads and are used for studying the activities of file systems. On the other hand, the traces collected at the physical level represent the *storage system* workloads and are used to study activities of storage systems. The latter are the traces of interest in our study.

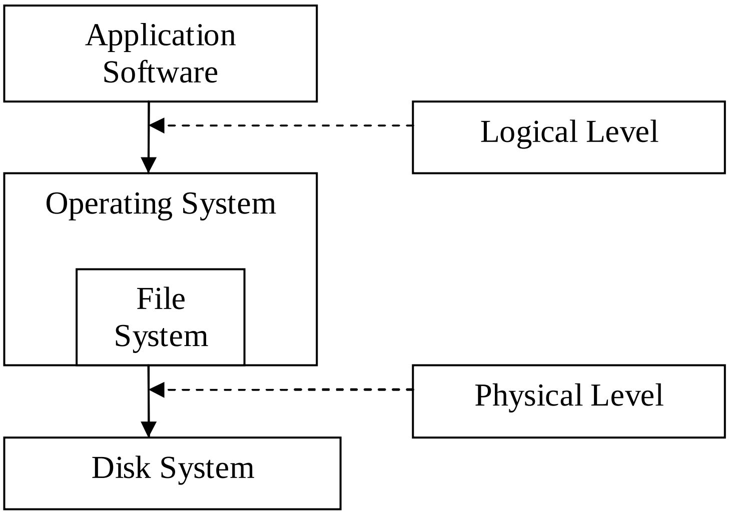


Figure 2.1: Levels of I/O trace collection

2.2.2 Trace File Type

Trace files can be binary or simple text files. In the SPC trace format specification document, it is recommended that the trace file be composed of variable length ASCII records, rather than binary data. Despite the fact that the ASCII format is somewhat wasteful of storage space and places higher CPU demands on analysis programs, it offers many advantages from a legibility and portability standpoint[Stfs92].

2.2.3 Type of Workload Traces

There are two types of traces: *Dynamic* and *Static* traces. Some studies, such as [Douc99, Sien94, Zhou93], concentrate on static data (static traces) which are collected by examining file system meta-data at one or several frozen instants in time, to minimize the complexity. These studies of static traces are useful for studying distributions of file attributes commonly stored in meta-data, such as file sizes, file names and last access times. Dynamic traces contain continuous file or block access patterns yielding more detailed information about storage usage[Drac00].

However, dynamic traces are more difficult to collect. They require huge amounts of space to store and usually involve modifying the operating system to collect them. Furthermore there is a performance impact on the CPU when collecting this kind of traces. In our work we are dealing with dynamic traces.

2.2.4 Tracing Methods

There are different methods of collecting storage workload traces. One way of collecting traces involves modifying the operating system thereby making the whole process difficult. Once this has been done there is also a task of convincing users to use the operating system which has been modified. Another method of collecting traces is to use an auditing system already built into the operating system to log all system calls relevant to both the file system and the storage system[Kppo98].

2.2.5 Trace Details

Most of the traces which have been collected by different researchers for various studies contain different details. The details contained in the traces usually depend on a number of factors including the following:

- The goal of the workload analysis. For instance, if one is interested in knowing how I/O request sizes vary overtime, then the timestamps and the sizes of I/O request are the details which should be included in the trace file.

- The level at which the traces are collected. The pieces of information about I/O requests that can be captured at the logical level are not the same as those that can be captured at the physical level. For example, logical block addresses can not be recorded at the logical level because they are not included in the application I/O requests. Instead file names, file descriptors or, in case of a database system, database object names (e.g., table, index and view names) are included.

- The method of trace collection. For example, using the auditing subsystem of the operating system will limit the details of the trace to what the subsystem is able to collect.

2.2.6 Tracing Period

When tracing a system in a given environment, one has to collect traces for a period of time long enough to capture all the characteristics of the workload. Characterization of workloads through traces does not only depend on the variety of the environments being traced but also on the length of the tracing period. The tracing period should be such that the trace data will reflect both the high activity periods (i.e., the *peak periods*) and the low activity periods[Drac00].

It is said that no single trace analysis project has the scope to analyze all the relevant features of all relevant workloads. Instead, each study lets us understand a part of the bigger picture[Drac00]. As a result most of the workload analysis studies that have been done so far differ in many ways in terms of the foregoing issues. Before deciding on these issues, researchers first define the goals of their workload analysis studies and then, depending on the goals, make certain decisions.

2.3 Uses of Storage System Traces

Storage system traces can be used in trace-driven simulations and to analyze storage system workloads.

2.3.1 Trace-driven Simulation

One of the approaches used by storage subsystem researchers employs a record of real system's storage activity in the form of disk request traces to run a simulation[Gang95]. Such a simulation is called a *trace-driven simulation*. It is a form of event-driven simulation in which events are taken from a real system that is operating under conditions similar to the one being simulated[Hsco01]. In [Gang95, Hsco01, Whsu01, Oust85, Hsco03, Wwha04] this approach was used.

One disadvantage of this approach is that timing effects are difficult to model realistically, specifically to account for events that occur faster or slower in the simulated system than in the original system. This problem comes from the fact that information about how the arrival of subsequent I/O requests depend upon the completion of previous requests cannot be easily extracted from a system and recorded in the traces[Wwha04]. This is one of the reasons why traces are not usually used to drive simulations of storage systems. The other reasons are[Gang95]:

- For non-technical reasons, it can be extremely difficult to convince systems administrators to allow tracing.

- Traces, which tend to be very large, must be stored online when they are to be used for experiments.

- Each trace represents a single measure of behavior, making it difficult to establish statistical confidence in results.

- It is very difficult to isolate and/or modify specific characteristics (e.g., arrival rate) of a trace.

- Traces do not support studies of expected future workloads, since one cannot trace what does not yet exist.

However, trace-driven simulation is still used in practice, probably because synthetic traces[1] are believed to be less representative of actual storage system workload as explained in Section 2.3.2(B). For example, Hsu *et al*[Wwha04] used trace-driven simulation with a new method they devised for replaying I/O traces that more accurately models the timing of I/O request arrivals and allows the I/O request rate to be more realistically scaled than previous practice.

2.3.2 Storage System Workload Analysis

Apart from driving trace-driven simulations, I/O traces are used to analyze the storage system workloads. The importance of storage system workload analysis can not be over-emphasized. The results of the workload analysis are usually used in various ways to improve the storage systems in terms of many different aspects. This section looks at some of the ways in which workload analysis results are used to improve storage system design, performance and correctness and to increase the storage system throughput.

A. Design of Storage Systems

As already mentioned in Section 1.2.2(A), when making changes to an existing storage system design or designing a new storage system, it is important to understand the current usage patterns[Kkpr92]. For example, if the workload analysis results show that large blocks of data are accessed frequently, the design should be such that the new storage system will be able to effectively support access to these large blocks of data[Drac00].

B. Modelling Storage System Workloads

In the design, performance and correctness evaluation of storage systems using software tools such as RAIDframe[2][Zksr96] and RAIDsim[3][Zksr90] synthetic traces are used. Synthetic traces are usually used in experiments because of the following reasons[Gang95]:

- Synthetic traces can be manufactured on the fly rather than stored.
- One can generate many synthetic traces with the same characteristics and thereby achieve some statistical confidence.
- Changes to the synthetic traces can be made much more readily.

[1] A synthetic trace is one whose values for the workload parameters, such as inter-arrival time of I/O requests, are generated using some models such as analytic and empiric (measured) distributions.

[2] RAIDframe is used in the design and development of RAID storage systems.

[3] RAIDsim is used for performance and correctness modelling of RAID storage systems.

- Synthetic traces representing expected future environments can be generated.

Note that if access and arrival patterns of I/O requests are not well reflected in a synthetic workload, then it is not a good representation of the actual storage system workload and the storage system therefore cannot be evaluated effectively using this synthetic workload. For the evaluation to be trustworthy, say for testing hardware designs of storage systems using simulations, the synthetic workloads should be representative of the actual workload[Gang95]. For this to be achieved, storage system workloads should be analyzed to come up with proper models for the workload parameters that can be used to generate the synthetic workloads. Currently, designers of disk arrays have few means of validating their design decisions with realistic workloads because workloads are poorly modelled[Vktr03].

C. Understanding Application I/O Behavior and Requirements

One reason for analyzing and characterizing storage workloads is to understand the I/O behavior and requirements of modern applications. Knowing these requirements and behavior can help buyers of large, enterprize-scale applications in deciding what type of storage systems to acquire. The buyers often have very little idea of the storage requirements of these applications, even though storage is a major, and increasing, percentage of the total system cost. Buying decisions are often based on simplistic metrics such as capacity. The vendors of storage systems also need to have knowledge of the behavior of their systems under a variety of workloads. I/O behavior and requirements may vary extensively depending on the application (e.g. transaction processing vs. decision support system) or the configuration of other system components, such as the amount of available memory[Vktr03].

D. Storage System Configuring

Given a storage system, such as a large disk array, it is very difficult to determine how to configure that system for better performance and higher throughput without accurate knowledge of the workload. For example, without information as to the sequentiality and mix of read and write operations, etc., present in a workload, it is hard to determine which RAID levels or stripe sizes to use[Dpjh98]. Therefore some workload analysis should be performed before the attempt to configure a storage system.

E. Storage System Monitoring and Tuning

Another reason for performing workload analysis is to monitor the storage system performance and the workload behavior. Once a storage system has been installed, it is necessary to monitor it to ensure that it is meeting its performance requirements, or to determine if the workload requirements have changed. If they have, then this information can be used in future tuning or reconfiguration of the system[Vktr03]. For example, if workload analysis results show that the load is not balanced among the logical volumes, then reconfiguring the storage system may be necessary.

F. Design of Optimization Techniques

One of the keys to overcoming the I/O bottleneck is to understand how storage is actually used, so that the storage system can be optimized by architects for the usage patterns, or if need be, new optimization techniques can be designed[Hsco03]. These techniques include: read caching, pre-fetching, write buffering, I/O scheduling and parallel I/O.

Some of the assumptions that are made by designers of storage systems about the workload are not true. For example, increasing cache sizes will not necessarily help improve read response time in all workloads[Vktr03]. This contradicts the assumption made by some of the storage system designers. Hence the need to analyze workloads from different environments and design optimization techniques accordingly.

The following sections discuss the optimization techniques mentioned above in detail.

- **Read caching.** *Caching* is a general technique for improving performance by temporarily holding in a faster memory those data items that are likely to be used again. How well the cache absorbs read requests is a very important factor. An effective cache replacement policy will bring about a high hit ratio and reduce seek times thus improving the overall performance of the storage system[Wwha04].

  Storage system workloads must be analyzed, to come up with effective cache replacement policies for the cache. For example, the *least recently used* (LRU) replacement policy will work well if the I/O workload is such that once a data item is read, it will be read many times in a given period of time. Further, to set the block size for the cache, we need to know the typical I/O request size because, managing the cache at a small granularity is very inefficient especially when most I/O transfers are much larger than the cache block size. The inefficiency is due to the large data structures needed for cache management[Wwha04].

- **Pre-fetching.** *Pre-fetching* is defined as a technique of predicting data blocks that are likely to be used in the future and fetching them before they are actually needed[Wwha04]. Since pre-fetching strategy simply pre-fetch blocks that are being accessed sequentially, understanding the access patterns of the workload is very important to decide whether this strategy will be effective for a given workload. This strategy will be effective in storage systems whose workloads consist of large data blocks accessed sequentially[Drac00].

- **Write buffering.** This term, *write buffering*, refers to the technique of temporarily holding written data in fast memory before destaging the data to the permanent storage. The time written blocks of data are kept in the write buffer before they are written back to the disk is called *delay time.*

  A write operation is usually reported as completed once its data have been accepted into the buffer. The write buffer helps to better regulate the flow of data to permanent storage especially for the fact that writes tend to come in bursts[Hsco03]. The write buffer is usually implemented with some form of non-volatile storage (NVS). This is to prevent any loss of data if the system fails before the buffered data are written to permanent storage. In some environments, (e.g., UNIX file system), a less expensive approach of periodically (usually every 30 seconds) flushing the buffer contents to disk is considered sufficient.

  By delaying the time at which the written data are destaged to permanent storage, write buffering makes it possible to combine multiple writes to the same location into a single physical write, resulting in a reduced number of physical writes that have to be performed by the system. Write buffering can also increase the efficiency of writes by allowing multiple consecutive writes to be merged into a single big-block I/O request. Furthermore, more sophisticated techniques can be used to schedule the writes to take advantage of the characteristics and the state of the storage devices.

  Among other things, one may have to analyze workloads to come up with reasonable values for write caching parameters, such as the delay time. By analyzing a workload one can tell whether write buffering will be effective for that workload. If the analysis shows that the workload exhibits locality of reference in the write operations, then write buffering will be an effective technique in the storage system handling that workload.

- **Request scheduling.** The time required to satisfy a request depends on the state of the disk, specifically, whether the requested data are present in the cache and where the disk head is relative to the requested data. *Request scheduling* is a technique in which the order in which requests are handled is optimized to improve performance. One has to analyze the workload to determine the effectiveness of request scheduling, say in terms of reducing write service time, in the storage system handling that workload. The effectiveness of this technique generally increases with the number of requests that are available to be scheduled at any given instant[Wwha04].

- **Parallel I/O.** A widely used technique for improving storage system performance is *parallel I/O*. In this technique, data are distributed among several disks so that multiple requests can be serviced by the different disks concurrently[Wwha04]. Besides, a single request that spans multiple disks can be speeded up if it is serviced by the disks in parallel. The latter tends to make more sense for workloads dominated by very large transfers such as in scientific workloads. For most other workloads, where requests are small and plentiful, the ability to handle many of them concurrently is usually more important. Therefore, it is also important to analyze and understand the workload behavior, before implementing this technique[Wwha04].

G. Other Uses of Workload Analysis Results

There are other purposes for which storage system workloads are analyzed. We give two examples here. Firstly, to determine whether two or more workloads are correlated. In [Hsco03], it is stated that storage managed by various entities in many big organizations would be consolidated through the use of storage utilities such as SAN or storage service providers (SSPs). Whether such pooling of resources is more efficient depends on the I/O characteristics of the workloads, and in particular, on whether the workloads' I/O events are independent. If two workloads' I/O events are not correlated, then the workloads can be handled by a single storage system because the resultant workload will be relatively smooth with minimal negative performance impact while the throughput is increased. Otherwise if the two workloads' I/O events are correlated, then they can not be efficiently handled by a single storage system.

Secondly, to determine whether a given workload is suitable for an intelligent storage system. Here understanding I/O workload characteristics is also important. Although the growth of processing power available in the storage systems makes it possible to build intelligent storage systems that can dynamically optimize themselves for the actual workload, we need to know how much idle time the workload allows in the storage system for running background functions that perform optimizations, say reallocating data blocks on the disks to balance the workload[Hsco03].

Chapter 3

Related Work on Workload Analysis

Our findings show that there are two categories of research concerning storage workload analysis.

- The first category involves analyzing storage workload traces with the aim of modelling the workload and discovering some usage patterns. For example [Hsco03, Hsco01, Drac00, Kkpr92, Fewa04] deal with characterization and modelling of different storage workloads from different environments including database, online transaction processing, office application, program development and scientific systems.

- The second category involves developing software tools to perform storage workload analysis given some workload traces. For example, Alistair and Kim of Hewlett Packard Laboratories, have developed a general tool, called Rubicon[Vktr03], for the characterization of storage workloads.

In our work we did not only perform some storage system workload analysis but also developed a software tool for performing the analysis. In this chapter, however, we only discuss previous studies related to storage workload analysis. We discuss work related to the development of workload analyzers in Chapter 5.

3.1 Storage Workload Classifications

Let us first recap the classifications of storage workloads explained in Section 2.2 to put things in context before we look at individual studies carried out by some researchers on storage workload analysis. These classes are:

- Storage system workloads. These are represented by traces collected at the physical level (see Figure 2.1). These workloads are also known as I/O traffic or physical storage workloads[Hsco03].

- File system workloads (or logical storage workloads). These are represented by traces collected at the logical level.

3.2 Previous Studies

File system workloads have been characterized in detail. Several studies of the logical workload characteristics of database and scientific systems have been conducted. Compared to the analysis of workload behavior at the logical level, storage system behavior has received much less attention[Drac00]. Hsu *et al*[Hsco03] pointed out that part of the reason is that storage system level characteristics are influenced by the file system design and configuration. Therefore the results of any analysis are less widely applicable. Hence the need to analyze the storage system workloads for many different environments.

In the following sections, we look at some specific studies done on analyzing storage workloads. First, we look at the analysis of storage system workload done by Hsu *et al*[Hsco03] and, second, analysis of file system workloads done by Roselli *et al*[Drac00]. We picked on these two studies because we found them to be more comprehensive than other storage and file system workload analysis studies that we came across in literature, respectively. For each study, we first give the aim, then describe the workload traces used and finally present some results. As we discuss each study, we also mention some of the comments and conclusions made by the respective researchers.

3.2.1 Analyzing Storage System Workloads by *Hsu et al*

A. Aim:

Hsu *et al*[Hsco03] collected traces to empirically examine the storage system workloads of a wide range of server and personal computer (PC) environments, focussing on how these workloads would be affected by developments in storage systems.

B. Trace Description:

Hsu *et al* collected traces at the physical level. They argued that, to study storage systems, analyzing traces collected at the physical level is generally more practical and realistic. This is in comparison with collecting logical level traces first, then filtering them by simulation to get traces representing the storage system workload. Traces collected at logical level have to be filtered away by simulating not only the buffer cache and pre-fetcher but also how the data are laid out and how the meta-data are referenced. This is a difficult task especially because these components in today's well-tuned systems are complex. Secondly, traces collected at the logical level do not include records about I/O requests that bypass the file system interface (e.g. raw I/O, virtual memory paging, and memory-mapped I/O)[Hsco03].

The traces collected were from both server and PC systems running real user workloads on three different platforms: Windows NT, IBM AIX and Hewlett-Packard HP-UX. On Windows they used a trace facility called *VTrace*. VTrace is a software tracing tool for Intel x86 PCs running Windows NT and Windows 2000. Traces on IBM AIX and Hewlett-Packard HP-UX were collected using kernel-level trace facilities built into the respective operating systems. A total of 14 PCs running Windows NT were traced over a period ranging from about a month to well over nine months. But they only utilized 45 days of the traces. The servers included two file servers, a time-sharing system and a database server that was being used by an enterprize resource planning (ERP) system[Hsco03].

C. Results:

- **Intensity of I/O:** As regards to I/O intensity, Hsu *et al* considered the significance of storage system workload component in the overall workload of the computer system taking into account the rate at which I/O requests were generated.

  - **Overall significance of storage system workload:** The PC storage system workload traces contained data for the periods during which user input activity occurred at least once every ten minutes. They found that the processor was, on average, busy only for about 10% of the time, while the storage system was busy only for about 2.5% of the time. They concluded that there was substantial idle time in the PC storage systems that could be used for performing background tasks. Due to the limited information in the server traces, the percentage of time the disk and processor were busy could only be calculated for the PC workloads[Hsco03].
  - **Storage system workload intensity:** From the analysis results, Hsu *et al* observed that server workloads were more I/O intensive than PC workloads and projected that the rates of I/O activity in servers would increase. They found that on average, PC workloads had 65,000 read and write operations per day while server workloads had 522,000 read and write operations

per day. These figures were taken over days when there was some activity recorded in the traces[Hsco03].

- **Request arrival rate:** Hsu *et al* generated data sets of I/O request inter-arrival times from the traces and fitted standard probability distributions to them. They discovered that the commonly used *exponential* distribution function to model inter-arrival time was a poor fit in all the workloads. Instead, they found the *lognormal* distribution function to be a reasonably good fit.

  They also explored the possibility of request scheduling by looking at the distribution of the *queue depth*, which they defined as the length of the request queue as seen by an arriving request. They found out that the average number of reads outstanding was only about 0.2 for all workloads while that of writes outstanding was about 0.3 for PC workloads and 5 for server workloads. For all the workloads they discovered that the maximum queue depth could be more than 90. *This they attributed to the fact that I/O requests seldom occurred singly but tended to arrive in groups because, if there were long intervals with no arrival, there were intervals that had far more arrivals than their even share.* This characteristic is called **burstiness**[Hsco03].

- **Dependence among workloads:** When storage system workload is smooth and uniform over time, system resources can be very efficiently utilized. However, if it is bursty resources have to be provisioned to handle the bursts so that during the periods when the system is relatively idle, these resources will not be wasted. Hsu *et al* established two ways of smoothing the load. The first one is to aggregate multiple workloads in the hope that the peak and idle periods in different workloads cancel one another out. This is possible only if the workloads are not correlated. The second approach to smoothing the traffic is shifting the load temporally, for example, deferring or offloading some work from the busy periods (e.g., write buffering) to the relatively idle periods or by eagerly or speculatively performing some work in the hope that such work will help improve performance during the next busy period (e.g., pre-fetching and re-organizing data layout based on access patterns)[Hsco03].

  Hsu *et al*'s workload analysis results show that there was little cross-correlation among the server workloads, suggesting that aggregating them would most likely help to smooth out the traffic and enable more efficient utilization of resources. On the other hand, the cross-correlation among PC workloads, except at small time intervals, was significant[Hsco03].

- **Self-similarity in storage system workloads:** Hsu *et al* also analyzed the storage system workloads to find out whether the workload arrival events were self-similar. Self-similarity, defined in simple terms, is the phenomenon which describes how a property of an object is preserved while scaling in space and/or in time[Hsco03].

Formal definition of self-similarity[Hsco03, Kppo98]: Let $\gamma = (Y_1,\ Y_2,\ Y_2,\ ...)$ be a stochastic process and $\chi = (X_1,\ X_2,\ X_2,\ ...)$ an incremental process of γ such that $X(i) = Y(i+1) - Y(i)$. In this context γ counts the number of I/O arrivals and $X(i)$ is the number of I/O arrivals during the ith time interval. γ is said to be self-similar with parameter H, $0 \leq H \leq 1$, if for all integers m,

$$\chi = m^{(1-H)}\chi^{(m)} \tag{3.1}$$

where

$$\chi^{(m)} = (X_k^{(m)}\ :\ k = 1,\ 2,\ 3,\ ...), \quad m = 1,\ 2,\ 3,\ ... \tag{3.2}$$

is a new aggregated time series obtained by dividing the original series into blocks of size m and averaging over each block as:

$$X_k^{(m)} = \frac{(X_{km-m+1} + ... + X_{km})}{m},\ k \geq 1 \tag{3.3}$$

and k is an index over the sequence of blocks. The single parameter H expresses the degree of self-similarity and is known as the Hurst parameter. For self-similar series H is between 0.5 and 1 and as $H \rightarrow 1$ the degree of self-similarity increases.

They found I/O traffic in all the workloads to be self-similar. Average H for PC workloads was found to be 0.81 and 0.9 for server workloads. They concluded that storage system workloads being self-similar implied that burstiness existed over a wide range of time scales and that attempts at smoothing the traffic temporally would tend to not remove all the variability.

- **Idle periods:** When the storage workload is not constant but varies overtime, there may be opportunities to use the relatively idle periods to do some useful work. Hsu *et al* considered intervals of I/O requests to be idle if the average number of I/O requests per second during the interval was less than 40 for the database server workload and 20 for all the other workloads. Based on this definition with an interval duration of 10 seconds, they found that for PC workloads more than 99% of the intervals were idle and the corresponding figure for the server workloads is more than 93%. Their conclusion was that there were resources in the storage systems that were significantly underutilized and that could be put to good use[Hsco03].

- **Interaction of reads and writes:** The interaction between reads and writes complicates a computer system and throttles its performance. Static data can simply be replicated to improve not only performance of the system but also its scalability and durability. However, if the data are being updated, the system has to ensure that the writes occur in the correct order and results of each write are propagated to all possible replicated copies or have these copies invalidated. The former makes sense if the updated data are unlikely to be updated again but likely to be read. The latter is useful when it is highly likely that data will be updated many times before the data are read. In cases where data are being both updated and read, replication may not be useful. Therefore, the read-write composition of the traffic with the flow of data from writes to reads is an extremely important workload characteristic[Hsco03].

  - **Read/write ratio:** Hsu *et al* observed that the ratio of the number of read requests to the number of write requests ranged from 0.71 to 0.82 for all the workloads. This means that writes accounted for about 60% of the requests.
  - **Working set:** Hsu *et al*[Hsco03] defined the working set W (t,τ) as the set of blocks referenced within a period of τ units of time with t as the ending time point. With $\tau = day$ and $t = midnight$, their results show that the daily working set for the various workloads ranged from just over 4% (PC workloads) to about 7% (file server) of the storage used. Since a small fraction of the data stored was in active use, it was probably a good idea to identify the blocks that were in use and to optimize their layout.
  - **Read/write dependencies:** Hsu *et al* classified dependencies into three categories: true dependence (read after write or RAW), output dependence (write-after-write or WAW) and anti-dependence (write-after-read or WAR). A RAW is said to exist between two operations if the first operation writes a

block that is later read by the second operation and there is no intervening operation on the block in a given number of references. WAW and WAR are defined similarly. Using 1,000 references as a window size, Hsu *et al* found that 25% of the reads fell into the WAR category for all the workloads. This meant that blocks that were read tended not to be updated, so that if disk blocks were replicated or re-organized based on their read access patterns, write performance would not be affected significantly. Hsu *et al* also found that all the workloads contained more WAW than RAW. This implies that updated blocks were more likely to be updated again than to be read, suggesting that if the blocks were replicated, only one of the copies had to be updated and the rest invalidated, rather than having all the copies updated[Hsco03].

Table 3.1 summarizes some of the characteristics of storage system workloads based on the work of Hsu *et al*[Hsco03].

Characteristic	**PC Workloads**	**Server Workloads**
I/O Intensity	high	high
Request Arrival Rate	bursty	bursty
Cross Correlation	high	low
Self-similar	Yes	Yes
Storage System Idle Time	high	high
Read/Write Ratio	0.71 to 0.82	0.71 to 0.82
Read/Write Dependencies	less WAR, high RAW and WAW	less WAR, high RAW and WAW
Block Working Set (%)	4	7

Table 3.1: Storage system workload characteristics

3.2.2 Analyzing File System Workloads by Roselli *et al*

A. Aim:

The aim of Roselli *et al*[Drac00] in their analysis of file system workloads was to understand how modern workloads affect the ability of file systems to provide high performance to users. They investigated the data lifetime, efficiency of cache, file sizes and access patterns.

B. Trace Description:

Roselli *et al* collected and analyzed file system traces from four different environments, including both UNIX and Windows NT systems, clients and servers, and instructional and production systems. The first three of these environments had Hewlett-Packard series 700 workstations running HP-UX 9.05. The first set of traces was from twenty workstations located in laboratories for undergraduate classes. They referred to this workload as the Instructional workload (INS). The second set of traces was from 13 machines for university graduate students, faculty and administrative staff of their research project. They referred to this workload as the Research workload (RES). The third set of traces was from a web server for an online library project. This server is said to have been receiving 2,300 accesses daily during the period of the trace. They referred to this workload as the Web workload (WEB). The fourth set of traces was from eight desktop machines running Windows NT 4.0. These machines were used for a variety of purposes including time management, personnel management, accounting, procurement, mail, office suite applications, web browsing, groupware applications, firewall applications. They referred to this workload as the NT workload (NT). We refer to these workloads as INS, RES, WEB and NT respectively.

C. Results:

- **Data lifetime:** Roselli *et al* found the write delay of 30 seconds being used as a standard in many file systems to be less than the lifetime of most blocks in the workloads they analyzed. Their results show that most blocks in INS had a lifetime uniformly distributed between 1 second and 1 hour and in WEB between 1 second and 5 minutes. In RES they found that most blocks had a lifetime slightly above 5 minutes. In NT they discovered a bimodal distribution pattern - nearly all blocks either died within a second or lived longer than a day.

  An important discovery of their analysis is the fact that a large portion of blocks, in all the four workloads, died due to being overwritten and a closer examination of the data showed a high degree of lifetime locality in overwritten files. For INS, 3% of all files created during the tracing period were responsible for all overwrites. These files were overwritten an average of 15 times each. For RES, 2% of created files were overwritten, with each file overwritten an average of 160 times. For WEB, 5% of created files were overwritten, and the average number of overwrites for these files was 6,300 times. For NT, 2% of created files were overwritten, these files were overwritten an average of 251 times each. In general, a relatively small set of files was repeatedly overwritten, causing many new writes and deletions.

Characteristic	**INS Workload**	**RES Workload**	**WEB Workload**	**NT Workload**
Data Lifetime	1 sec - 1 hour	over 5 min	1 sec - 5 min	0 - 1 sec and over 1 day
Optimal Cache Size (MB)	16	16	64	16
Largest File Size (MB)	419	244-419	244	419
Access patterns: File sizes				
< 20KM	Entire runs	Entire runs	Entire runs	Entire runs
> 100KM	Random runs	Both entire and random runs	Random runs	Random runs

Table 3.2: File system workload characteristics

They concluded that since newly written blocks often had longer than 30 seconds, increasing the write delay would reduce write traffic. However they were also quick to mention that the write delay was limited by the amount of data allowed to be cached by the operating system. They also performed some simulations to find the optimal write buffer size and found that a small size of about 16MB would be sufficient even for write delays of up to a day.

- **Cache efficiency:** Roselli *et al* found that relatively small caches absorbed most read traffic and that there were diminishing returns to using large caches. This is in contrast to the claim made by Rosenblum and Ousterhout[Rose92] in 1992 that large caches would avert most disk reads. They found that there was little benefit in increasing the cache size beyond 64MB for WEB workload and 16MB for the other workloads.

  They also pointed out that the effect of memory-mapped file should be considered when performing workload analysis because it had become a common method to access files. Their results show that the average numbers of files mapped were 43, 18 and 7 for INS, RES and WEB workloads respectively during peak periods.

- **File sizes:** Knowing the distribution of file sizes is important for designing metadata structures that efficiently support the range of sizes commonly in use. Therefore Roselli *et al* analyzed the workloads in terms of files sizes. From the results, they noticed that the file sizes in all the workloads were larger, on average, than those in the workloads analyzed in earlier studies. For example, the largest file accessed in the traces collected from the Sprite file system[Bake91] in 1991 was 38MB; the largest files in the workloads Roselli *et al* analyzed ranged from 244MB (WEB) to 419MB (INS and NT)[Drac00].

- **Access patterns:** When optimizing file system performance, knowing the access patterns is crucial. With this in mind, Roselli *et al* considered the access patterns in the workloads. Their results show that small files of less than 20KM were read in their entirety whereas large files over 100KB were accessed randomly for all the workloads except the RES workload where both entire runs and random runs were well-represented.

Table 3.2 summarizes some of the characteristics of file system workloads based on the work of Roselli *et al*[Drac00].

Chapter 4

Statistical Methodology

In this chapter, we discuss the statistical tools and techniques that we implemented in ESSWA for analyzing storage system workloads. The statistical approach we took is similar to the one that Lourens Walters used to analyze and model *web traffic* for simulations[Lowa04]. Our aim in this chapter is to give an overview and not a detailed discussion of these tools and techniques. For a detailed study we refer the reader to any typical statistics and probability theory text book such as [Dwsc92, Tjst03, Fdal95, Reki99].

The rest of this chapter is as follows: Section 4.1 introduces the concept of probability distribution functions; Section 4.2 describes some visual techniques that are used to explore data sets[1]; Section 4.3 discusses the key data statistics such as the mean, variance, coefficient of kurtosis and tail index; Section 4.4 looks at some of the attributes[2] of probability distribution functions; and Section 4.5 explains how models for the storage system workload parameters can be derived using a statistical methodology.

4.1 Probability Distribution Functions

Before we describe the various statistical tools and techniques that we implemented in ESSWA, we first introduce the concept of probability distribution functions of random variables. We start by defining random variables in Section 4.1.1 and cumulative distribution functions in Section 4.1.2.

[1] A data set in this writing refers to a collection of values.

[2] In this study we refer to a parameter defined in statistical science as a numerical constant that is unknown but whose estimation from data will in some sense characterize a given data set for a particular class of probability distribution functions as an *attribute*.

4.1.1 Random Variables

A *random variable* is a function that associates a unique numerical value with every outcome of an experiment. The value of the random variable will vary from trial to trial as the experiment is repeated. For example in our study, typical random variables would be the request size and the logical volume number of an I/O request. Random variables can either be *continuous* or *discrete.* Continuous random variables have uncountably many possible values, and take each with probability 0; these quantities usually represent lengths, weights, etc., and need not be integers. Discrete random variables can only take a finite or countable number of values, and have a positive probability of taking each one; typically these are integer-valued quantities obtained by counting[Dwsc92].

4.1.2 Cumulative Distribution Functions

All random variables (discrete and continuous) have a cumulative distribution function. A cumulative distribution function, $F(x)$, is a function giving the probability that the random variable X is less than or equal to x, for every value x. Formally, the cumulative distribution function $F(x)$ is defined as[Dwsc92]:

$$F(x) = P[X \leq x] \;\; for \;\; -\infty < x < \infty \tag{4.1}$$

4.1.3 Probability Distribution Functions

A. Probability Mass Functions

The probability distribution function of a discrete random variable, usually called the *probability mass function*, is a list of probabilities associated with each of its possible values, say $x_1,\ x_2,\ x_3\ ...,\ x_n$. More formally, the probability mass function of a discrete random variable X is a function which gives the probability $p(x_i)$ that the random variable equals x_i, for each value x_i[Dwsc92]:

$$p(x_i) = P[X = x_i] \;\; for \;\; i = 1,\ 2,\ 3,\ ...,\ n \tag{4.2}$$

It satisfies the following conditions:

- $0 \leq p(x_i) \leq 1 \;\; for \;\; i = 1,\ 2,\ 3,\ ...,\ n$
- $\sum_i^n p(x_i) = 1$

The cumulative distribution function, $F(x)$, of a discrete random variable X is found by summing the probabilities $P[X \leq x]$.

B. Probability Density Functions

The probability distribution function of a continuous random variable is a function which can be integrated to obtain the probability that the random variable takes a value in a given interval and is called the *probability density function.* More formally, the probability density function, $f(x)$, of a continuous random variable X is the derivative of the cumulative distribution function $F(x)$[Dwsc92]:

$$f(x) = \frac{d}{dx}F(x) \tag{4.3}$$

It follows that:

$$\int_a^b f(x)dx = F(b) - F(a) = P[a < X < b] \tag{4.4}$$

If $f(x)$ is a probability density function then it must obey two conditions:

- that the total probability for all possible values of the continuous random variable X is 1:

  $\int_{-\infty}^{\infty} f(x)dx = 1$

- that the probability density function can never be negative: $f(x) > 0$ for all x.

4.2 Visual Techniques

4.2.1 Histogram

A histogram is simply a graph of grouped (binned) data in which the number of values in each bin is represented by the area of a rectangular box. Histograms are used to identify the shape, location and scale of the distribution of data sets. Histograms show the presence of symmetry, peakedness, outliers and heavy tails. A histogram requires a *bin width.* The bin width is the range of values in each group or bin of a histogram. A poor choice of bin size results in loss of information or over-sensitivity to small changes in the data distribution. There are three commonly used rules for bin width calculation: the Surges rule, the Friedman/Diaconis rule and the Scott rule[Lowa04].

4.2.2 Empirical Cumulative Distribution Function (ECDF)

An ECDF, for each potential value x in a given data set, is equal to the probability that observations are less than or equal to x. Note that an ECDF is the same as the cumulative distribution function defined in Section 4.1.2 except that it is based on a data set. An ECDF is considered to be a visual tool because it is usually in the form of a graph.

4.3 Data Statistics

Having mentioned visual displays as a way of gaining an intuitive idea of the data being analyzed, in this section we discuss the second way. This approach involves computing a few "key" numbers which summarize a given data set. The aim is to reduce a large batch of data to just a few numbers which can be grasped simultaneously, and help in the understanding of the important features of the data set[Dwsc92].

4.3.1 Ratio and Frequency Table

A. Ratio

A *ratio* is a comparison of two numbers. It is the relationship between two groups or values, which expresses how much bigger one is than the other. For example, in this study we are interested in the ratio of the number of read operations to the number of write operations in the storage system workloads.

B. Frequency Table

A *frequency table* is a table which is constructed by dividing a data set, say *inter-arrival times* for I/O requests, into intervals or bins, and counting the number of values in each interval. The actual number of values as well as the percentage of values in each interval are shown. A frequency table can also be a data listing that lists the frequency of each value in a data set. The frequency table actually provides most of the information shown in a histogram[Dwsc92].

4.3.2 Five Number Summary

The five number summary is based on the idea of *ranking* the quantitative items in a *sorted* data set. Let's say we have n items in a data set (x_1, x_2, x_3, ..., x_n). The first number is said to have *rank 1*, the second smallest *rank 2*, ..., the largest *rank n.* The smallest item x_1 is called the *minimum* and the largest, x_n, is called the *maximum.* The number whose rank is $(n+1)/2$ is called the *median* and denoted as x_m. If n is odd, the median is the middle number. If n is even, then the median is the average of "two middle numbers" in the data set. The number whose rank is $l = (\lceil m \rceil + 1)/2$ is called the *lower quartile* and the number whose rank is $u = n - l + 1$ is called the *upper quartile.* These five-numbers (minimum, lower quartile, median, upper quartile, maximum), called the *five-number summary* and usually written from smallest to largest as (x_1, x_l, x_m, x_u, x_n), provide useful summary of a data set[Dwsc92].

For example if we have (512, 1,025, 8,192, 25,400, 32,832) as a five-number summary of a data set of I/O request sizes in bytes. This tells us that:

- Half the I/O request sizes are less 8,192 bytes and half are greater than 8,192 bytes, because 8,192 is the median.

- Half the I/O request sizes are between 1,025 and 25,400 bytes, because these two I/O request sizes are the lower and upper quartiles.

Besides the five-number summary there are two important types of data items that need to be considered, the *Outliers* and *Strays*. These are basically extreme values in the data set. The *Outliers* are defined as those observations which are greater than[Dwsc92]

$$x_m + 6(x_u - x_m) \tag{4.5}$$

or less than

$$x_m - 6(x_u - x_m) \tag{4.6}$$

Less outlying values called the *Strays* are those observations which are not outliers but are greater than[Dwsc92]

$$x_m + 3(x_u - x_m) \tag{4.7}$$

or less than

$$x_m - 3(x_u - x_m) \tag{4.8}$$

4.3.3 Measures of Location and Spread

A *measure of location* is a statistic that locates the ”middle”, in some sense, of a data set[Dwsc92]. For example, given a set of I/O request inter-arrival times, we should use the measure of location to answer the question: *What is the typical inter-arrival time of I/O requests*? Two most important measures of location are the median and the mean[Dwsc92]. A *measure of spread* is a statistic that gives the extent of variability in the data set. In the following sections we discuss the mean and then some measures of spread.

A. Mean

The *sample mean*, which is considered to the most important measure of location, is found by adding together all the values in the data set, and dividing the total by n, the number of items in the data set. The mean is defined to be[Dwsc92]:

$$\bar{x} = \frac{1}{n}\sum_{i=1}^{n} x_i \tag{4.9}$$

B. Range and Inter-quartile Range

Two measures of spread can be defined from the five-number summary. They are the *range* R, defined as:

$$R = x_n - x_1 \tag{4.10}$$

and the *inter-quartile range* I, defined as:

$$I = x_u - x_l \tag{4.11}$$

The range is unreliable as a measure of spread because it depends only on the smallest and the largest values in the data set, and thus is sensitive to the outlying values, unlike the inter-quartile range[Dwsc92].

C. Sample Variance and Sample Standard Deviation

Two important measures of spread are the *sample variance and the sample standard deviation.* The variance, denoted by, s^2, is defined as:

$$s^2 = \frac{1}{n-1}\sum_{i=1}^{n}(x_i - \bar{x})^2 \tag{4.12}$$

In other words, it is the sum of the squared differences between each data value and the mean divided by the $n-1$, where n is the sample size. The standard deviation, denoted by s, is the square root of the variance. The sample standard deviation is sometimes expressed as a fraction of the sample mean, in which case it is known as a coefficient of variation. Coefficient of variation (CV) is defined as:

$$CV = \frac{s}{\bar{x}} \tag{4.13}$$

4.3.4 Sample Skewness and Kurtosis

A. Sample skewness

Sample Skewness is a measure of symmetry, or more precisely, the lack of symmetry. A distribution, or data set, is symmetric if it looks the same to the left and right of the center point. Sample skewness of a distribution is defined as[Fdal95]:

$$\frac{n}{(n-1)(n-2)}\sum_{i=1}^{n}(\frac{x_i-\bar{x}}{s})^3 \tag{4.14}$$

where $n > 2$ is the number of non-missing values for a variable, x_i is the i^{th} value of the variable, $\bar{x}$ is the sample mean, and s is the sample standard deviation. If the coefficient of skew is greater than 0 then, we term the distribution to be positively skewed or vise versa[Fdal95].

B. Sample Kurtosis

Sample kurtosis is a measure of whether the data are peaked or flat relative to a normal distribution. That is, data sets with high kurtosis tend to have a distinct peak near the mean, decline rather rapidly, and have heavy tails. Data sets with low kurtosis tend to have a flat top near the mean rather than a sharp peak.

Sample kurtosis of a distribution is defined as[Reki99]:

$$\frac{n(n+1)}{(n-1)(n-2)(n-2)}\sum_{i=1}^{n}(\frac{x_i-\bar{x}}{s})^4 - \frac{3(n-1)^2}{(n-2)(n-2)} \tag{4.15}$$

where $n > 3$.

For the normal distribution, the coefficient of kurtosis is always 3. Therefore an alternative definition for coefficient of kurtosis is subtracting 3 from the value obtained from Equation 4.15. This implies that the coefficient of kurtosis in effect measures departure from normality, i.e., negative values corresponding to flatter than normal distribution, and positive values to distributions which are more peaked and heavy tailed than the normal[Tjst03]. Note that the histogram is an effective graphical technique for showing both the skewness and kurtosis of data sets.

4.3.5 Tail Index

The *tail index* is a statistic that is used to determine how heavy tailed a distribution is. A heavy tailed distribution is a distribution for which the upper part or "tail" of the distribution declines according to a power rate rather than an exponential rate.

Distributions commonly used to model network characteristics such as the exponential and normal distributions have tails which decline exponentially or faster. Heavy tailed distributions have tails that decline slower than these distributions which result in greater degree of variability in the size of observations. In heavy tailed distributions, the probability of larger observations occurring is relatively high. We say that a random variable X follows a heavy-tailed distribution with tail index α if

$$P[X > x] \sim cx^{-\alpha}, \;\; as\; x \rightarrow \infty, \; 0 < \alpha < 2, \tag{4.16}$$

where c is a positive constant, and where $\sim$ means the ratio of the two sides tends to 1 as $x \rightarrow \infty$. This distribution has infinite variance, and if $\alpha \leq 1$ it has an infinite mean. Smaller values of α imply heavier tails. Heavy tailed behavior can also be detected in a data set by inspecting the data set's complementary cumulative distribution function which is defined as[Lowa04]:

$$P[X > x] = \bar{F}(x) = 1 - F(x), \tag{4.17}$$

where $F(x)$ is the cumulative distribution function $F(x) = P[X \leq x]$.

4.3.6 Correlation Coefficient and Autocorrelation Function

Quite typically, a single observation of the real world will result in a collection of measurements, which can be expressed as a vector of observations describing the outcome of the experiment. For example, in our study we have measurements like request sizes, inter-arrival times and logical seek distances of I/O requests.

It is usually necessary to determine the relationships among the different parameters in an experiment. The *correlation coefficient* is a statistic used to test for independence between any two given data sets. The correlation coefficient is a quantity which is denoted by r. It takes values between -1 and +1. The sign of r indicates whether the relationship between two variables involved is positive or negative. The absolute value of r, ignoring the sign, gives a measure of the strength of the association between the variables. The further $|r|$ is from zero, the stronger the relationship. The correlation coefficient takes the value 0 only if the two random variables are unrelated[Fdal95, Dwsc92].

The formula for calculating the correlation coefficient r from two random variables (x_1, y_1), (x_2, y_2), ..., (x_n, y_n), where the means of the x-values and the y-values are $\bar{x}$ and $\bar{y}$ and their standard deviations are s_x and s_y, respectively, is[Fdal95]:

$$r = \frac{1}{n-1}\sum_{i=1}^{n}(\frac{x_i - \bar{x}}{s_x})(\frac{y_i - \bar{y}}{s_y}) \tag{4.18}$$

where n is the size of the data set.

The autocorrelation of a random variable X is simply the correlation of the process against a time-shifted version of itself. The magnitude of the shift is called the *lag.* The *autocorrelation function* is used to test for independence between time instances of a random variable and is defined as[Reki99]:

$$r(m) = \frac{\sum_{i=1}^{n-m}(x_i - \bar{x})(x_{i+m} - \bar{x})}{\sum_{i=1}^{n}(x_i - \bar{x})^2},\ m < n \tag{4.19}$$

where m is the lag and n is the size of the data set.

The autocorrelation value, like the coefficient of correlation, will vary between -1 and +1, with values near 1 and -1 indicating stronger correlation. For many random variables the value $r(1)$ is particularly significant. If a random variable is autocorrelated the autocorrelation is often greatest at a lag of 1. Thus ESSWA calculates $r(1)$ for any given data set. For a data set that is not autocorrelated, $|r(1)| < 2/\sqrt{n}$ at 95% confidence level[Lowa04].

4.3.7 Goodness-of-fit Statistic

A goodness-of-fit statistic is used to determine the conformity of the observed data's empirical distribution function with a theoretical distribution function. In other words it is a measure of how well observed data conform to a specified, expected, or theoretical probability distribution function. A number of goodness-of-fit tests exist. These include the chi-square test, Anderson Darling test and the λ^2 Discrepancy measure. In this section we give more details about the λ^2 discrepancy statistic which we implemented in ESSWA for finding probability distribution functions fitting some of the workload parameter data sets. Pederson and Johnson defined the λ^2 discrepancy statistic in the paper "Estimating Model Discrepancy" as[Lowa04]:

$$\widehat{\lambda}^2 = \frac{X^2 - K - df}{N-1} \tag{4.20}$$

where N is the sample size and *df* is defined as $n - r - 1$, where n is the number of bins and r the number of model constants estimated from the data. Assuming that $Y = (Y_1,\ Y_2,\ Y_3,\ ...,\ Y_n)$ is a multinomial random variable with $p = (p_1,\ p_2,\ p_2,\ ...,\ p_n)$

denoting a hypothesized vector for Y, then[Lowa04]

$$K = \sum_{i=1}^{n} \frac{(Y_i - Np_i)}{Np_i} \tag{4.21}$$

and

$$X^2 = \sum_{i=1}^{n} \frac{(Y_i - Np_i)^2}{Np_i} \tag{4.22}$$

As can be seen from Equation 4.20, the λ^2 statistic is based on the χ^2 goodness-of-fit statistic. The λ^2 and χ^2 statistics are both based on binning techniques, and measure the magnitude of departure of empirical data from a distributional model.

The λ^2 discrepancy measure has a number of advantages over the other goodness-of-fit tests. It has been found that for smaller data sets the λ^2 statistic is less biased and has smaller variance than the χ^2 statistic[Sppm90]. Other than that, the λ^2 statistic can be used when dealing with large data sets whereas other statistics e.g. the Anderson Darling and the χ^2 statistics, cannot. Another advantage of the λ^2 statistic is that it can be used to compare the goodness-of-fit tests performed on data sets with different sample sizes. This is not possible with the χ^2 statistic and the Anderson Darling statistic. The λ^2 statistic can be used to compare results from tests performed on data sets of different sizes because the sample size and number of bins are taken into account in the calculation of the statistic[Lowa04].

The λ^2 statistic can take on any real value. On one hand, negative values indicate a perfect fit between the given probability distribution function and the data set. On the other hand, the closer the positive value is to zero the better the fit. Therefore positive values close to zero indicate a better fit while large positive values indicate a lack of fit. *Note that the conclusions drawn based on this statistic are not always right* because this test is not infallible. Therefore further analysis of the data should be done to ascertain the match[Lowa04]. ESSWA, to accomplish this, is able to generate a data set of random values following the best fitting probability distribution function using attribute values, such as the mean value, estimated from the data set being analyzed and then draw the cumulative distribution functions for the two data sets. This allows the user to see how well the probability function said to be the best fit, matches the real data set.

4.4 Theoretical Distribution Attributes

Section 4.3 investigates measures of location and spread for samples of data. There are equivalent concepts for probability distribution functions of random variables. The most important attributes of location and spread for random variables are given the same names, the *mean* also known as the *expected value*, and the *variance.* The mean and variance of a random variable are theoretical because they are calculated from the probability functions and not from a data set. The formulae defining the mean and the variance of a random variable are different from the formula which defined the sample mean, denoted as $\bar{x}$ and sample variance, denoted as s^2[Dwsc92].

4.4.1 Theoretical Mean

Like the sample mean, the *expected value* (or mean) of a random variable indicates its average or central value. It is a useful summary value of the variable's distribution. Stating the expected value gives a general impression of the behavior of some random variable without giving full details of its probability mass function (if it is discrete) or its probability density function (if it is continuous).

Two random variables with the same expected value can have very different distributions. There are other useful descriptive measures which affect the shape of the distribution, for example the variance. The expected value of a random variable X is symbolized by $E(X)$ or μ.

If X is a discrete random variable with possible values $x_1,\ x_2,\ x_3,\ ...,\ x_n$, and $p(x_i)$ denotes $P[X = x_i]$, then the expected value of X is defined as:

$$\mu = E(X) = \sum_{i=1}^{n} x_i p(x_i) \tag{4.23}$$

where the elements are summed over all values of the random variable X.

If X is a continuous random variable with probability density function $f(x)$, then the expected value of X is defined as:

$$\mu = E(X) = \int_{-\infty}^{\infty} x f(x) dx \tag{4.24}$$

4.4.2 Theoretical Variance

Like the sample variance, the *theoretical variance* of a random variable is a non-negative number which gives an idea of how widely spread the values of the random variable are likely to be; the larger the variance, the more scattered the observations on average. The variance gives an impression of how closely concentrated around the expected value the distribution is; it is a measure of the 'spread' of a distribution about its average value. Variance is symbolized by $V(X)$ or $Var(X)$ or σ^2. The variance of the random variable X is defined to be:

$$\sigma^2 = Var(X) = E(X - E(X))^2 = E(X^2) - E(X)^2 \tag{4.25}$$

where $E(X)$ is the expected value of the random variable X.

4.5 Modelling Methodology

A number of probability distribution functions have proved useful as models for a large variety of practical problems in science, engineering and elsewhere. In this study we are interested in deriving models, in form of probability distribution functions, for the storage system workload parameters. ESSWA provides statistics, given the workload parameter data sets, which can be used to formulate the models for the workload parameters. These models can in turn be used to generate synthetic workloads for storage system evaluation using simulations.

4.5.1 Modelling Continuous Parameters

Probability density functions and mass functions are used to model parameters which can be defined as continuous and discrete random variables respectively. Among all the workload parameters that we considered, only the inter-arrival time of I/O requests can take on continuous values. ESSWA computes the λ^2 discrepancy statistic for use in finding the probability density function which fits a given data set of inter-arrival times. The probability distribution functions it tests against are the *normal, lognormal, exponential, gamma, beta, Weibull* and *Pareto* probability density functions. We only considered these statistical mathematical families because, with the exception of the exponential and normal probability density functions, they can model heavy tailed empirical distributions. Recent studies, such as [Hsco03], has shown that storage system traffic can be heavy tailed, bursty and, like web server and network traffic, appears to exhibit self-similar characteristics. It is also important to note that these probability density functions are usually used as workload models especially that, together they represent a wide variety of possible shapes[Lowa04]. Table 4.1 (taken from [Lowa04]) presents functions and attributes associated with these seven probability distributions.

Lourens[Lowa04] implemented a *function* in R programming language for calculating the λ^2 discrepancy statistic in his study which involved finding mathematical models for web traffic parameters, such as inter-session time and web user request inter-arrival time, for simulation. As already mentioned, the λ^2 discrepancy measure is

Distribution Family	**Distribution Attributes**	**General Density Function**
Exponential	$\mu,\ \alpha$	$\frac{1}{\alpha}e^{-(x-\mu)}\ \ x > \mu;\ \ \alpha > 0$
Weibull	$\mu,\ \gamma,\ \alpha$	$\frac{\gamma}{\alpha}(\frac{x-\mu}{\alpha})^{\gamma-1}e^{(\frac{-(x-\mu)}{\alpha})^{\gamma}}$ $x > \mu;\ \ \gamma > 0;\ \ \alpha > 0$
Lognormal	$\theta,\ \zeta,\ \sigma$	$\frac{e^{-(\frac{(log(x-\theta)-\varsigma)^2}{2\sigma^2})}}{(x-\theta)\sigma\sqrt{2\pi}}\ \ x \geq \theta;\ \ \zeta > 0;\ \ \sigma > 0$
Normal	$\mu,\ \sigma$	$\frac{e^{-(\frac{(x-\mu)^2)}{2\sigma^2})}}{\sigma\sqrt{2\pi}}\ \ \ \sigma > 0$
Beta	$\alpha,\ \beta,\ a,\ b$	$1\frac{(x-a)^{\alpha-1}(b-x)^{\beta-1}}{B(\alpha,\beta)(b-a)^{\alpha+\beta-1}}$ $a < x < b;\ \ \alpha > 0;\ \ \beta > 0$
Gamma	$\mu,\ \gamma,\ \alpha$	$2\frac{1}{\alpha\Gamma(\gamma)}(\frac{x-\mu}{\alpha})^{\gamma-1}e^{(\frac{x-\mu}{\alpha})}$ $x > \mu;\ \ \alpha > 0;\ \ \gamma > 0$
Pareto	$\alpha,\ \beta$	$\beta\alpha^{\beta}x^{-\beta-1}\ \ x \geq \alpha;\ \ \alpha > 0$

Table 4.1: Probability density functions used in this study and their attributes

a statistic which relies on the *binning of data.* Since the size chosen for the bin affects the accuracy of the statistic, the function uses well known techniques to accurately calculate bin widths. These techniques calculate bins for skewed distributions differently from symmetric distributions. The function also uses the method of maximum likelihood estimation to obtain likelihood estimates for probability density function attributes[Lowa04] (see Table 4.2).

We incorporated this function in ESSWA. ESSWA, given a data set of inter-arrival times, uses it to compute the λ^2 discrepancy statistic for each of the seven probability density functions and then picks the one with the smallest value as the best match.

4.5.2 Modelling Discrete Parameters

For workload parameters that can be defined as discrete, ESSWA provides statistics that can be used to formulate probability mass functions. These parameters include the logical volume number, request size, logical seek distance, operation type and parallelism degree. Probability mass functions require that we state the probability of having a particular discrete value. The following section describes how we determined the probabilities while the next section explains the *binomial* distribution function. We used this function to model one of the storage system workload parameters.

A. Determining Probability

Suppose an identical experiment is conducted N times, and let M be the number of times that the event E is observed to occur. We can define the probability of E

Probability Density Function	**Maximum Likelihood Estimator**
Exponential	$\hat{\alpha} = \frac{1}{n}\sum_{1=1}^{n} x_i = \bar{x}$
Weibull	$\hat{\gamma} = [(\sum_{i=1}^{n} x_i^{\hat{\gamma}} log x_i)(\sum_{i=1}^{n} x_i^{\hat{\gamma}})^{-1} -$ $n^{-1}\sum_{i=1}^{n} log x_i]^{-1}$ $\hat{\alpha} = [n^{-1}\sum_{i=1}^{n} x_i^{\hat{\gamma}}]^{\frac{1}{\hat{\gamma}}}$ for fixed $\mu = 0$
Lognormal	For $z_i = log(x_i - \theta)$: $\zeta = \frac{1}{n}\sum_{i=1}^{n} z_i = \bar{z}$ $\hat{\sigma} = [(n-1)^{-1}\sum_{j=1}^{n}(z_j - \bar{z})^2]^{\frac{1}{2}}$
Normal	$\hat{\mu} = \frac{1}{n}\sum_{i=1}^{n} x_i = \bar{x}$ $\hat{\sigma} = [(n-1)^{-1}\sum_{j=1}^{n}(x_j - \bar{x})^2]^{\frac{1}{2}}$
Beta	$\psi(\hat{\alpha}) - \psi(\hat{\alpha} + \hat{\beta}) = \frac{1}{n}\sum_{i=1}^{n} log(\frac{Y_i - a}{b-a})$ $\psi(\hat{\beta}) - \psi(\hat{\alpha} + \hat{\beta}) = \frac{1}{n}\sum_{i=1}^{n} log(\frac{b - Y_i}{b-a})$
Gamma	$\frac{1}{n}\sum_{i=1}^{n} x_i = \hat{\alpha}\hat{\lambda}$ $\frac{1}{n}\sum_{i=1}^{n} ln x_i = ln\hat{\alpha} + \frac{\Gamma'(x)}{\Gamma(x)}$ for fixed $\mu = 0$
Pareto	$\hat{\alpha} = min(x_i)$ $\hat{\beta} = n[\sum_{i=1}^{n} log(\frac{x_i}{\hat{\alpha}})]^{-1}$

Table 4.2: Maximum likelihood estimators

by[Tjst03]:

$$P[E] = lim_{N\to\infty}(\frac{M}{N}) \tag{4.26}$$

B. Binomial Distribution Function

The binomial distribution may be used as a probability model in situations in which the following conditions are satisfied[Dwsc92]:

- We have a random experiment with exactly two outcomes, one of which we can label ”success”, and the other ”failure”.

  e.g., An I/O request can either be a read (i.e., taken as a ”success”) or write (taken as a ”failure”) operation.

- The random experiment is repeated n times, $n \geq 1$. The outcome on any one repetition is not influenced by the outcome on any other repetition. e.g., if the operating system makes $n = 1,000$ I/O requests to the storage system, the operation type (i.e., either a read or write) of the requests must not be autocorrelated.

- The probability of success remains constant from trail to trial.

If we have n independent trials, each trial has two outcomes, success or failure, and P[success] $= p$ for all trials. Let the random variable X be the number of successes in

n trials. Then X has the binomial distribution, and $p(x) = P[X = x]$ is given by the probability mass function[Dwsc92]:

$$\begin{aligned} p(x) &= (^{n}_{x})p^x(1-p)^{n-x} \ \ for \ \ x = 0,\ 1,\ 2,\ ...,\ n \ \ and \\ &= 0 \quad otherwise \end{aligned} \tag{4.27}$$

In the above example, X is the number of read operations and because there are 1,000 I/O requests, x can assume one of the values 0, 1, 2, 3, ..., 1,000, and X is therefore an example of a discrete random variable.

The binomial process occurs in many contexts. From an industrial or commercial perspective, one of the most binomial processes occurs in the field of quality. In particular, the binomial probability distribution provides probability models for deciding whether or not a consignment of goods meets the desired specifications[Dwsc92]. In our work we used the binomial distribution to model I/O operation type which can either be *read or write.*

Chapter 5

Storage System Workload Analyzers

There are a number of studies involving storage workload analysis and characterization using file system and storage system workload traces. These efforts have used custom built programs for the tasks at hand. Each study has developed individual tools for various analysis tasks, such as appending and filtering traces, but none has integrated these tasks into a single analysis system.

Despite the fact that the majority of storage workload analysis and characterization studies use ad hoc solutions for trace analysis, a number of trace analysis tools have been described in literature including [Vktr03, Grim95, IBM99, Toua91]. These tools read, filter and analyze trace data, and provide facilities for data visualization.

In this chapter, we discuss ESSWA, the tool we developed for analyzing storage system workloads. We start off by discussing a similar tool, called *Rubicon*[Vktr03] in Section 5.1, which is a general tool for workload analysis and then discuss ESSWA from the programmer's point of view in Section 5.2 and from a user point of view in Section 5.3. Finally we compare and contrast Rubicon with ESSWA in Section 5.4.

5.1 Rubicon

Rubicon was developed by Alistair and Kim of Hewlett Packard Laboratories for the characterization of I/O workloads. In general, Rubicon reads a sequence of I/O trace records, performs some analysis on them, and outputs the result of the analysis. The results include data statistics such as the average values for workload parameter data sets. This tool provides a rich set of operations on I/O traces, and was designed to be easily extended through the addition of new analysis functions and reporting methods.

5.1.1 Motivation

The developers of Rubicon identified a number of drawbacks with other workload analysis tools. This is because these tools do not provide the following features and capabilities[Vktr03]:

A. Partitioning of the Trace

Alistair and Kim observed that many analysis tools do not effectively partition traces into separate parts for processing. This can be necessary for many reasons, including studying the workload on each logical volume, examining various time-slices of the trace, or singling out the I/O requests caused by a single operation type. This leads to complicated analysis code, leading to longer analysis process and multiple files that need to be organized, or poorly structured code that is often difficult to maintain or modify for a related, but different, task[Vktr03].

B. Extendability

The second issue is how to incorporate new types of analysis. In this case, there are two options; the new functionality must either be incorporated into the original program, or a new program developed. Again at some later point in time if another type of analysis is desired, the same decision must be made. Alistair and Kim noted that over time, this results in unnecessarily large and complex programs, or in a multitude of smaller programs, each of which contains some functionality identical to, but separated from, its parent. In either case, the maintenance and development problems associated with the code base are made worse[Vktr03].

C. Multiple Types of Output

Another problem noted is that most of the other analysis tools provide only one type of output. But some users may have a need for the same analysis results to be presented and used in different ways. Most systems use separate programs for the conversion of results, when what is really required is a tool that can, when necessary, simultaneously generate different output formats from the same set of internal results. Again, this raises the issue of one program against many[Vktr03].

D. Results Staging

Fourthly, one stage of analysis or processing might depend on a second. Once again, in most tools this is typically accomplished through the use of several different programs.

Although this can be an effective technique, Alistair and Kim observed that it tends to complicate the analysis environment[Vktr03].

E. Configurability of the Analysis Tool

The last issue pointed out by Alistair and Kim, is that of incorporating information about the details of the storage configuration for the machine on which the trace is gathered. These details, say about the logical and physical storage devices in the system, are vital for the correct interpretation of the trace records. They suggested that any analysis tool must be easily configurable for many different system layouts. Without this capability, the user is faced with the development of customized code for every system on which the workload analysis must be performed[Vktr03].

5.1.2 Requirements for Workload Analysis Tools

Based on the above mentioned issues, Alistair and Kim came up with some requirements for a generic workload characterization system. Thus they developed Rubicon with these requirements in mind. These requirements are[Vktr03]:

- **Single image:** there must be a single program and source base.
- **Trace manipulation:** there must be built-in primitives for manipulating I/O traces, particularly for filtering.
- **Configurability:** the system components must be easily and dynamically (re)configured for different target systems or analysis.
- **Multiple report formats:** the system must have the ability to report results in several different formats, independent of the analysis performed.
- **Extensibility:** the system must allow new analysis algorithms and reporting formats to be easily incorporated.
- **Staged analysis:** the system must allow the results of one analysis stage to be fed into another.
- **Efficiency:** the final system must not be significantly more expensive in execution time than the comparable special-purpose program for the same analysis task.

5.1.3 Design of Rubicon

Rubicon was developed in $C++$ and is made up of objects that can be combined in different ways to perform various analysis tasks, and which together satisfy the requirements specified in Section 5.1.2. The following sections describe these objects.

A. Analysis Objects

Each of the *analysis objects* in Rubicon performs a separate function on the trace records. These objects are connected in a directed acyclic graph, with the trace records entering at a designated root, flowing through each of the nodes, and terminating in analysis objects. There are four of these base object types[Vktr03]:

- **Analyzers:** Analyzers do most of the *work* in Rubicon, given a stream of records, they analyze those records and store their results for future use. The current version of Rubicon has approximately fifty different types of analyzers, that perform many different types of analysis ranging from simple rate measurement (number of I/O requests per second) to correlations between I/O request streams, spatial and temporal locality measures and self-similarity properties[Vktr03].

- **Multiplexors:** Multiplexors read trace records and multiplex these to several outputs. Multiplexors are used to create multiple flows of records, so that different types of analysis can be done on each record stream. As such, they essentially serve as connectors between the components of the other types.

- **Filters:** Filters read a stream of trace records, and output selected records, based on a filter specification. Filters are used to select out subsets of the full trace for analysis. A typical use is to combine multiplexors and filters to generate a number of logical trace record streams, say one for each disk on a machine.

- **Transformers:** Transformers perform simple transformations on trace records. The most frequently used transformer in the current Rubicon system transforms the logical volume offset specified in the trace to a physical storage device offset.

B. Result Manipulation Objects

The objects described above handle the flow and analysis of records. The following set of objects are used to store and manipulate the results[Vktr03]:

- **Attributes:** Attributes store the results calculated by analyzers. Typically, though not necessarily, such results are named sets or tables of numeric results.

- **Flows:** Flows store the attributes calculated by a set of related analyzers. In the most common Rubicon configurations, each flow typically represents the analysis for a different storage device. Flows provide the interface through which attributes can be communicated to different parts of the system.

- **Reporters:** Reporters transform the results stored in attributes into the desired output format(s). Given the raw data and structuring information stored in an attribute, reporters generate output in the desired format. They are reporters that generate flat data files, *gnuplot* input files (for graph generation) and spreadsheets.

5.1.4 Trace Collection

Rubicon mainly uses I/O traces gathered from systems running HP-UX. The HP-UX kernel contains a number of internal instrumentation points. Through a specialized interface, it is possible to obtain complete information on all system calls and on a number of important events in the internal kernel interfaces. One of these trace points is located at the internal block I/O interface. Alistair and Kim developed a tool to record information at this interface on all I/O requests that are issued and store this information in a trace file. Each record in the trace file contains values for the various parameters measured[Vktr03].

5.2 ESSWA

In our study, our aim was to develop a storage system workload analyzer that does not only meet the requirements outlined in Section 5.1.2 upon which Rubicon is based but which is also more user-friendly, and uses a proven and stable statistical environment. A tool that is better than all the tools we encountered in literature.

In this section we describe ESSWA from a programmer's perspective. Since we implemented the functions of ESSWA to perform various analysis tasks in **R** language and environment, we give a brief introduction to R before talking about ESSWA in detail.

5.2.1 R language and Environment

R is a language and environment for statistical computing and graphics. It is a GNU project and is similar to another language and environment called **S** which was developed at Bell Laboratories by John Chambers *et al*[Wnve04].

R provides a wide variety of statistical techniques including linear and nonlinear modelling, classical statistical tests, time-series analysis, classification, clustering, and graphical techniques, and is highly extensible. R provides an *Open Source* route to participation in research involving statistical methodology. The S language is often the vehicle of choice in such research but is not free.

Advantages of Using R

Using R offers some advantages some of which are not offered by other programming languages like C, C++, Java and S, and statistical software like S and Statistica[Wnve04]. Some of the advantages of R are that it[Wnve04]:

- is available as free software under the terms of the Free Software Foundation's GNU General Public License.
- compiles and runs on a wide variety of UNIX platforms and similar systems (including FreeBSD and Linux), Windows and MacOS.
- allows users to easily produce well-designed publication-quality plots, including mathematical symbols and formulae where needed.
- is an integrated suite of software facilities for data manipulation, calculation and graphical display. It is a fully planned and coherent system, rather than an incremental buildup of very specific and inflexible tools, as is frequently the case with other data analysis software. It includes:
  - an effective data handling and storage facility,
  - a suite of operators for calculations on arrays, in particular matrices,
  - a large, coherent, integrated collection of intermediate tools for data analysis,
  - graphical facilities for data analysis and display either on-screen or on hard copy, and
  - a well-developed, simple and effective programming language which includes conditionals, loops, user-defined recursive functions and input and output facilities.
- is designed around a true computer language and allows users to add functionality by defining new functions. For computationally-intensive tasks, C, C++ and Fortran code can be linked and called at run time. Furthermore, advanced users can write C code to manipulate R objects directly.

- can also be extended (easily) via packages. There are about eight packages supplied with the R distribution and many more are available through the CRAN family of Internet sites[1] covering a very wide range of modern statistics.

- has its own LaTeX-like documentation format, which is used to supply comprehensive documentation, both on-line in a number of formats and in hard copy.

Because of these benefits, we decided to use R to implement the analysis tasks for ESSWA.

5.2.2 ESSWA Design

ESSWA is made up of two major components which communicate via an interface as shown in Figure 5.1. The first component is a front-end GUI implemented using Visual Basic .NET. We chose this integrated development environment because of its features that enable one to easily develop robust and acceptable GUIs.

The second component is a collection of functions (i.e., the back-end) implemented in R to perform various analysis tasks. The front-end basically calls the functions in the back-end to calculate various statistics for the workload parameter data sets. However, to visually display the data sets in form of histograms or cumulative distribution functions, the front-end directly calls R generic functions.

Next we discuss what constitutes the ESSWA back-end, the interface between the ESSWA front- and back-end, and the ESSWA front-end.

A. Back-end

The ESSWA back-end is a single program which consists of a number of functions listed in Appendix A. These include functions, given a data set, that calculate the following statistics:

- five number summary (minimum value, lower quartile, median, upper quartile, maximum value),

- coefficient of variation,

- coefficient of kurtosis,

- coefficient of skew,

- upper outlier limit,

[1] http://cran.r-project.org/

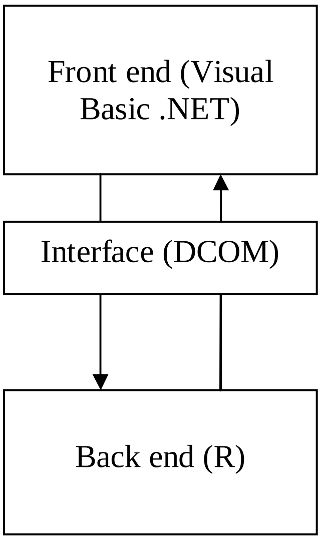


Figure 5.1: Interaction between ESSWA front-end and back-end

- lower outlier limit,
- coefficient of correlation for different data sets,
- tail index,
- goodness-of-fit statistic and
- autocorrelation value.

B. Interface between Front- and Back-end

For the interface between R and Visual Basic .NET, we used the Distributed Component Object Model (DCOM) software. DCOM is a standard *Windows* mechanism used for communication between Windows applications on the same machine or different machines. One application is run as DCOM back-end which offers services to the front-end calling application. The services are described in a *Type Library* and are (more or less) language-independent. Therefore the calling application can be written in C, C++, Visual Basic, Perl, Python, etc. The basic R distribution is not a DCOM back-end, so we used some currently available add-ons that interface directly with R or R programs and provide a DCOM back-end. The two packages we used are called DCOMServer and RDCOM and are described in [Rdct04].

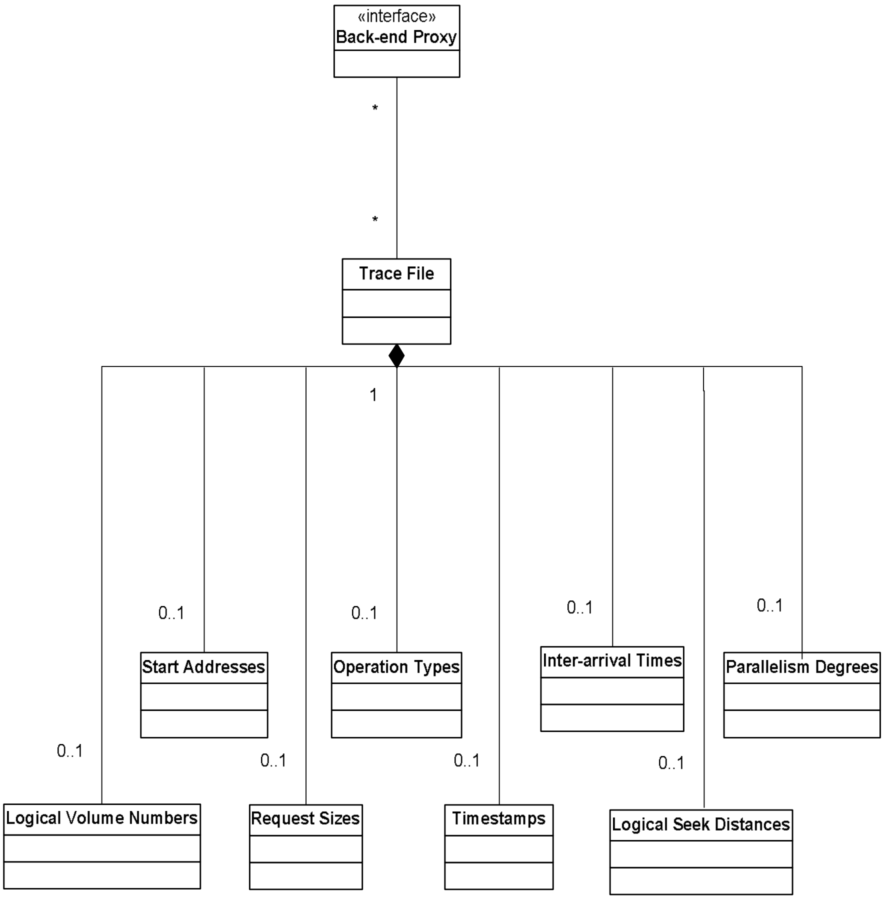


Figure 5.2: ESSWA Front-end object classes

C. Front-end

ESSWA front-end is made up of a number of objects belonging to a number of classes. Figure 5.2 shows the Unified Modelling Language (UML) class diagrams of these classes and their relationships. From the diagram we can see that there are ten classes and that an object belonging to the *trace file* class is made up of eight objects of the classes: *logical volume numbers*, *inter-arrival times*, *request sizes*, *operation types*, *logical seek distances*, *parallelism degrees*, *timestamps* and *start addresses.* We can also see that the class *Back-end proxy* is an interface class.

The eight classes which comprise the *trace file* class have member variables to store *corresponding* data sets from trace files. For example, the *inter-arrival times* class has a member variable (i.e., an array of real numbers) to store inter-arrival times from a trace file. Each of the eight classes implements a number of methods which in turn simply call functions in the back-end to manipulate the data sets stored in their member variables. The statistical tools and techniques used in the computations are discussed in Chapter 4.

The *trace file* class implements five methods. When invoked, the first method reads a trace file and populates the eight objects which together make up a trace file object. For each unique timestamp in a trace file this method counts the number of I/O requests with that particular timestamp. The collection of these numbers are stored in the parallelism degree object. This method also generates inter-arrival times, by calculating the differences between consecutive timestamps, and logical seek distances, by calculating the differences between consecutive start addresses. The second method appends data from a trace file to the trace file object which is already populated. The third method calculates the tracing period using the values in timestamps object. The fourth method calculates the coefficient of correlation between any two data sets for different parameters. The fifth and last method filters a trace file retaining only data for the specified operation type and/or logical volume number.

An object of type *back-end proxy* acts as an interface between the front-end and the back-end. This object handles the communication between the front- and back-end using the DCOM software. Implementing the communication in this way helped us to eliminate the need and task of writing the code to handle this communication in every method in the front-end which intends to communicate with the back-end. Therefore instead of calling the back-end functions directly, all methods wanting to interact with the back-end do this indirectly through the back-end proxy object. This object is instantiated the first time a trace file is read.

5.3 Using ESSWA

This section describes ESSWA from the user viewpoint. It explains how to install, run and interact with ESSWA to analyze storage system workload traces. It also discusses ESSWA's robustness and resource requirements in terms of memory and CPU time.

5.3.1 Installing ESSWA

The following steps must be taken to install ESSWA:

- Install the R interpreter. The installation file for R can be downloaded through the CRAN family of Internet sites.

- Install the *DCOMServer* and *RDCOM* packages for the DCOM interface. These packages are also available for downloading through the CRAN family of Internet sites.

- Run the R analysis program listed in Appendix A on the R console to create the ESSWA back-end.

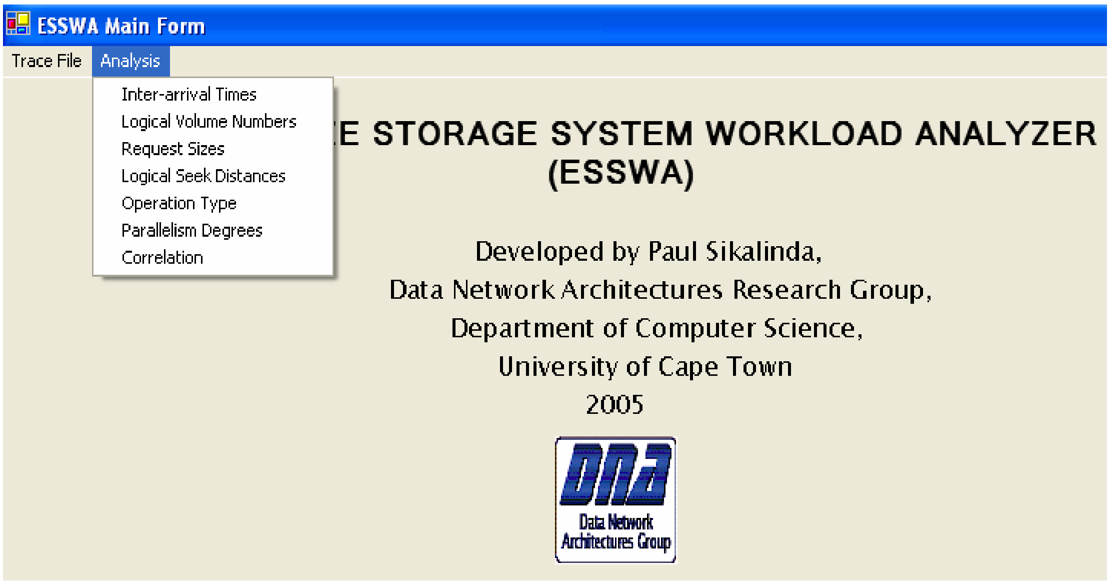
- Install the .NET engine which can be downloaded through the Microsoft website[2]. This engine is needed to run the ESSWA front-end GUI.

- Install the ESSWA front-end GUI. We have created a *Windows* setup file for the GUI. This setup file should be double clicked to begin the installation. When the installation begins, the person performing the installation will be prompted to specify the folder in which to install the front-end. Once this folder is specified, the installation process proceeds to completion. When the installation is completed, an icon for the GUI is created in the installation folder.

5.3.2 Running ESSWA

The user must click on the GUI icon created in the installation folder to run the GUI. When the GUI starts, the *main window* appears with two menus; *Trace File* and *Analysis* as shown in Figure 5.3. Each of them has in turn a number of menu items which the user can use to read, append and filter trace files, and to carry out workload analysis tasks as described below.

[2] http://www.microsoft.com/downloads/details.aspx?FamilyId=262D25E3-F589-4842-8157-034D1E7CF3A3&displaylang=en



E STORAGE SYSTEM WORKLOAD ANALYZER
(ESSWA)

Developed by Paul Sikalinda,
Data Network Architectures Research Group,
Department of Computer Science,
University of Cape Town
2005

Figure 5.3: ESSWA main window

5.3.3 ESSWA Tasks

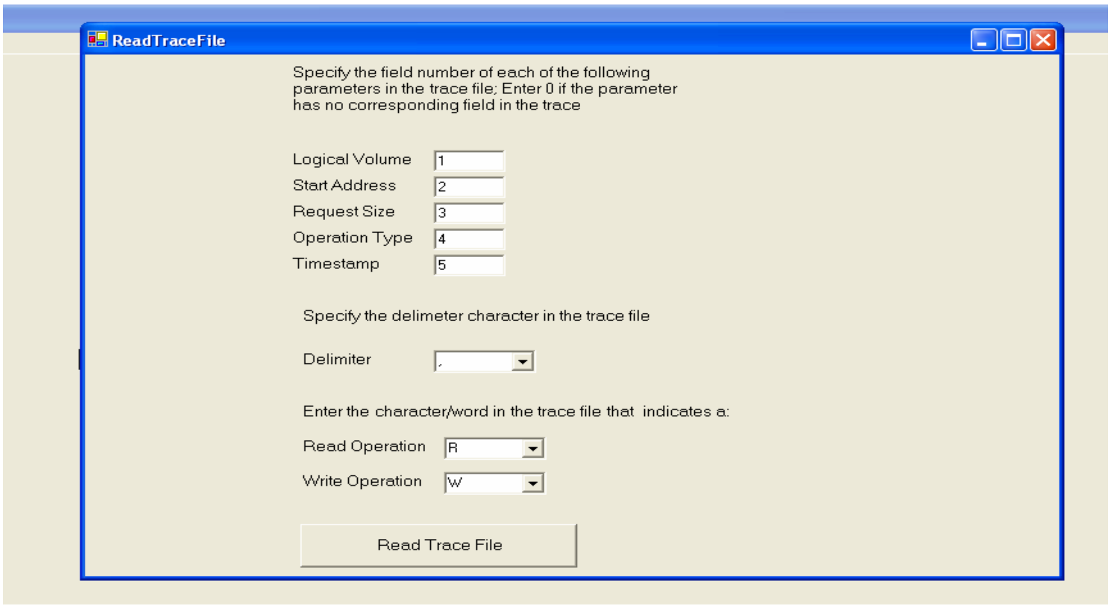
A. Input Tasks

The tasks of reading, appending and filtering trace files are performed using the *Trace File* menu. Under this menu, there are three menu items; *Read Trace File*, *Append Trace File* and *Filter Trace File* used for the following tasks, respectively:

- **Reading a trace file.** When the user clicks on the *Read Trace File* menu item, the *parameter window* shown in Figure 5.4 appears. On this window, the user must specify the following before the system reads the trace file:

  - for each of the workload parameter shown in the window, the position of the corresponding field in the trace file. A value of zero should be entered against the parameter whose corresponding field is missing in the trace file.
  - the delimiting character of the fields in the trace file. The character can either be entered or selected from a list provided.
  - the words (e.g., ”write” and ”read”) used to indicate whether a given record in the trace file is for a read or write operation. The words can either be entered or selected from a list provided.

  After entering the above details, the user should click on a button at the bottom of the window to read the trace file. When this button is clicked, the system



Specify the field number of each of the following parameters in the trace file; Enter 0 if the parameter has no corresponding field in the trace

Logical Volume 1
Start Address 2
Request Size 3
Operation Type 4
Timestamp 5

Specify the delimeter character in the trace file

Delimiter ,

Enter the character/word in the trace file that indicates a:

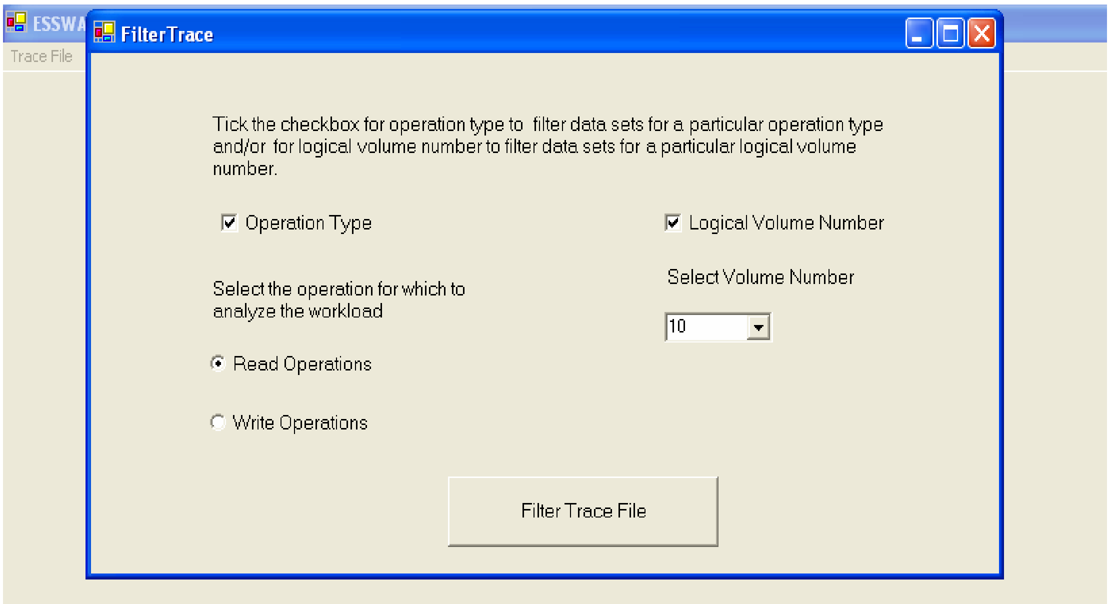
Read Operation R
Write Operation W

Figure 5.4: Read trace file parameter window

presents a dialog box which allows the user to browse the file directory and select the trace file. After the file is selected, the system will then attempt to read it and inform the user whether it has been read successfully or not. *Note that when the user reads a trace file, its data replace the data of any previously read trace file.*

- **Appending a trace file.** The process of appending a trace file is the same as that of reading a trace file *except* that the data of any previously read trace file is not replaced, i.e., the data of the newly read trace file is appended to the data of any previously read trace file.

- **Filtering a trace file.** When the user clicks on the *Filter Trace File*, the window shown in Figure 5.5 appears. There are two check boxes on this window. One labelled *Operation Type* while the other *Logical Volume Number*. The user should tick either of them or both, depending on whether he wants to filter the *read* trace file in terms of operation type or logical volume number or both. If the user ticks the *Operation Type* check box then he must specify the operation type of the records to retain in the trace file by ticking one of the two radio buttons shown on the figure at the bottom left. Similarly, if the user ticks the *Logical Volume Number* check box, he must specify the logical volume number of the records to retain in the trace file by selecting the volume number in the combo box provided on the window. Finally, the user must click on the button labelled *Filter Trace*



Tick the checkbox for operation type to filter data sets for a particular operation type and/or for logical volume number to filter data sets for a particular logical volume number.

☑ Operation Type

☑ Logical Volume Number

Select the operation for which to analyze the workload

● Read Operations

○ Write Operations

Select Volume Number

10

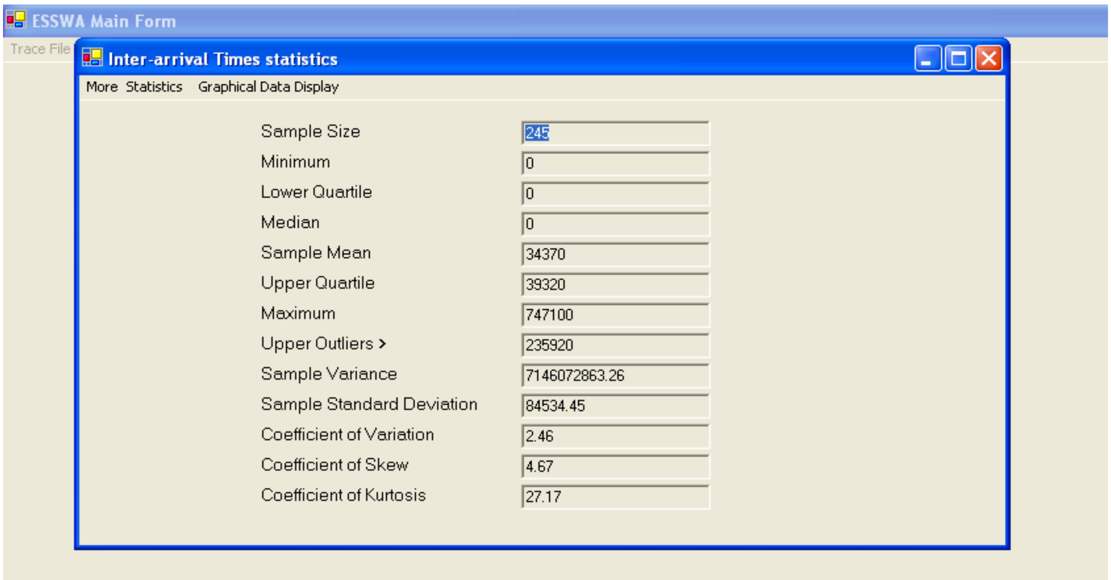
Figure 5.5: Filter trace file window

File to complete the task.

B. Analysis Tasks

The user must use the *Analysis* menu on the front-end's *main window* to analyze the workload represented by the trace file which has been read. This menu has seven menu items; *Inter-arrival Times*, *Logical Volume Numbers*, *Request Sizes*, *Logical seek Distances*, *Operation Types*, *Parallelism Degrees* and *Correlation*. The user must click on one of the first six menu items to carry out analysis of the workload in terms of the respective parameter and the *Correlation* menu item to determine the correlation between any two workload parameters.

- **Inter-arrival time.** When the user clicks on the *Inter-arrival Times* menu item, a window is presented with a table of the key data statistics for the inter-arrival time data set except the λ^2 discrepancy values, the tail index and the autocorrelation value as shown in Figure 5.6. This window also provides menu items which can be used to display the data sets for inter-arrival times in form of histograms and ECDFs as illustrated in Figure 5.7. From this window the user can move on, using a menu item, to the next (see Figure 5.8) which presents the other three statistics. On this window, the best three probability density functions fitting the inter-arrival time data set are displayed together with the estimated attribute values and the corresponding λ^2 discrepancy values. This



Sample Size	245
Minimum	0
Lower Quartile	0
Median	0
Sample Mean	34370
Upper Quartile	39320
Maximum	747100
Upper Outliers >	235920
Sample Variance	7146072863.26
Sample Standard Deviation	84534.45
Coefficient of Variation	2.46
Coefficient of Skew	4.67
Coefficient of Kurtosis	27.17

Figure 5.6: Key data statistics window

last window also presents a button which is labelled *ECDFs.* When this button is clicked, three data sets following the best three probability density functions are generated (i.e., each one of the three data sets following one of the three functions) and their ECDFs are drawn on a graph together with that of the inter-arrival time data set from the trace file. This allows the user to visually confirm the matches.

- **Logical volume number.** When the user clicks on the *Logical Volume Number* menu item, the window illustrated in Figure 5.9 is presented with a frequency table of the logical volume numbers. This window also provides controls for obtaining the autocorrelation values and the menu items for drawing the histograms and ECDFs for logical volume number data sets.

- **Request size.** The *Request Size* menu item is used to analyze the request sizes of a workload. The first window presented when the user clicks on this menu item provides the same statistics for request sizes as those for inter-arrival times shown in Figure 5.6. From this window the user can move on to the next which presents a frequency table and provides controls for obtaining the tail index and autocorrelation value for the request size data sets.

- **Logical seek distance.** The user must click on the *Logical Seek Distances* menu item to analyze the workload in terms of the logical seek distance workload parameter. The window that appears provides the same statistics for logical seek

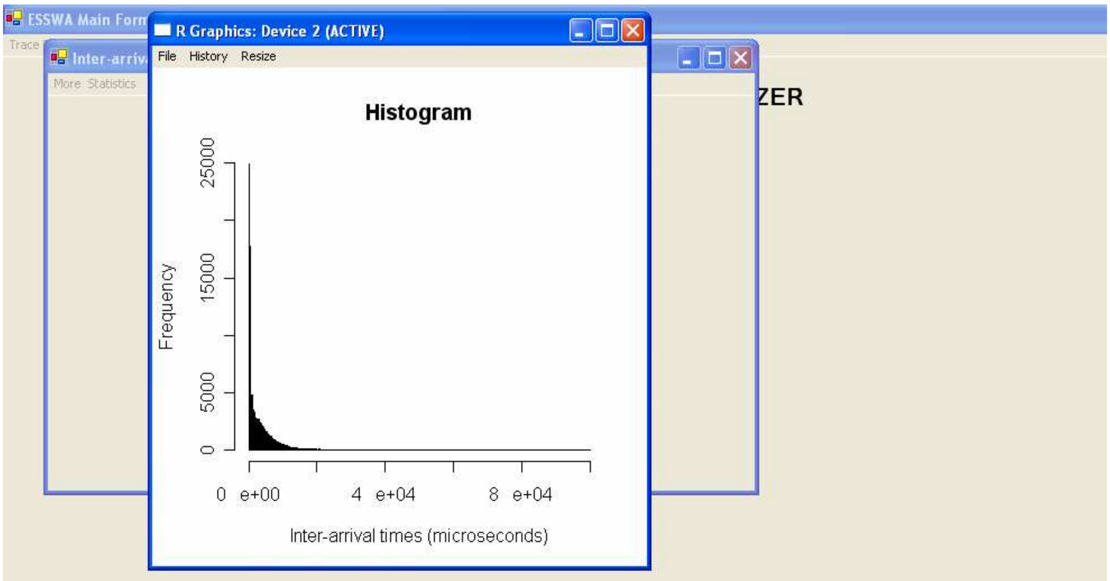
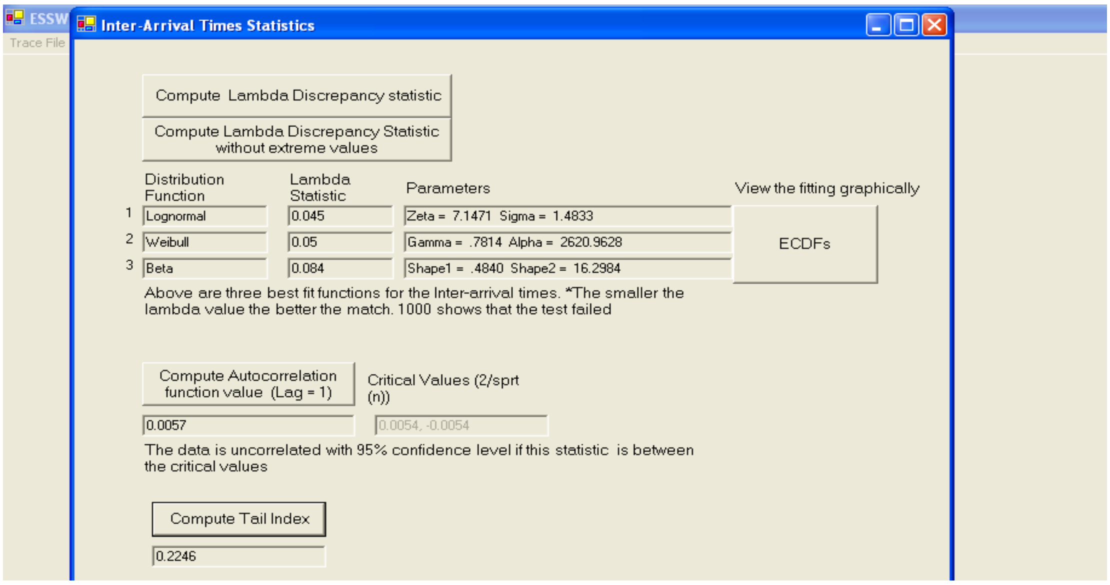


Figure 5.7: Histogram


ESSW
Trace File
Inter-Arrival Times Statistics
Compute Lambda Discrepancy statistic
Compute Lambda Discrepancy Statistic without extreme values
View the fitting graphically
ECDFs
Compute Autocorrelation function value (Lag = 1)
Compute Tail Index

	Distribution Function	Lambda Statistic	Parameters
1	Lognormal	0.045	Zeta = 7.1471 Sigma = 1.4833
2	Weibull	0.05	Gamma = .7814 Alpha = 2620.9628
3	Beta	0.084	Shape1 = .4840 Shape2 = 16.2984

Above are three best fit functions for the Inter-arrival times. *The smaller the lambda value the better the match. 1000 shows that the test failed

Critical Values (2/sprt (n))

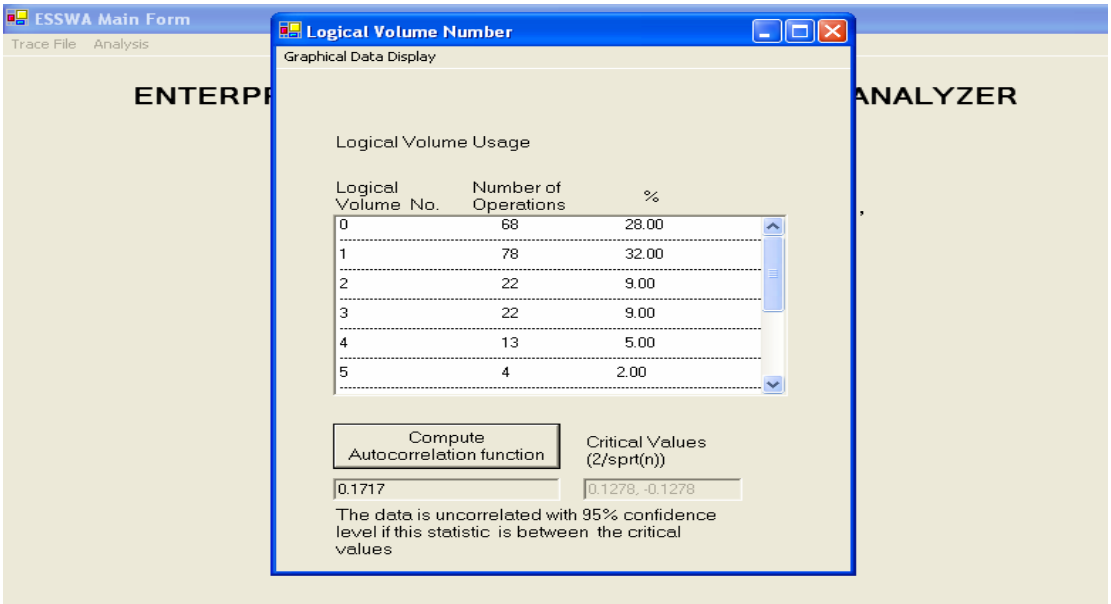
0.0057

0.0054, -0.0054

The data is uncorrelated with 95% confidence level if this statistic is between the critical values

0.2246

Figure 5.8: Lambda statistics, autocorrelation function and tail index window



Logical Volume Usage

Logical Volume No.	Number of Operations	%
0	68	28.00
1	78	32.00
2	22	9.00
3	22	9.00
4	13	5.00
5	4	2.00

Critical Values (2/sprt(n))

0.1717

0.1278, -0.1278

The data is uncorrelated with 95% confidence level if this statistic is between the critical values

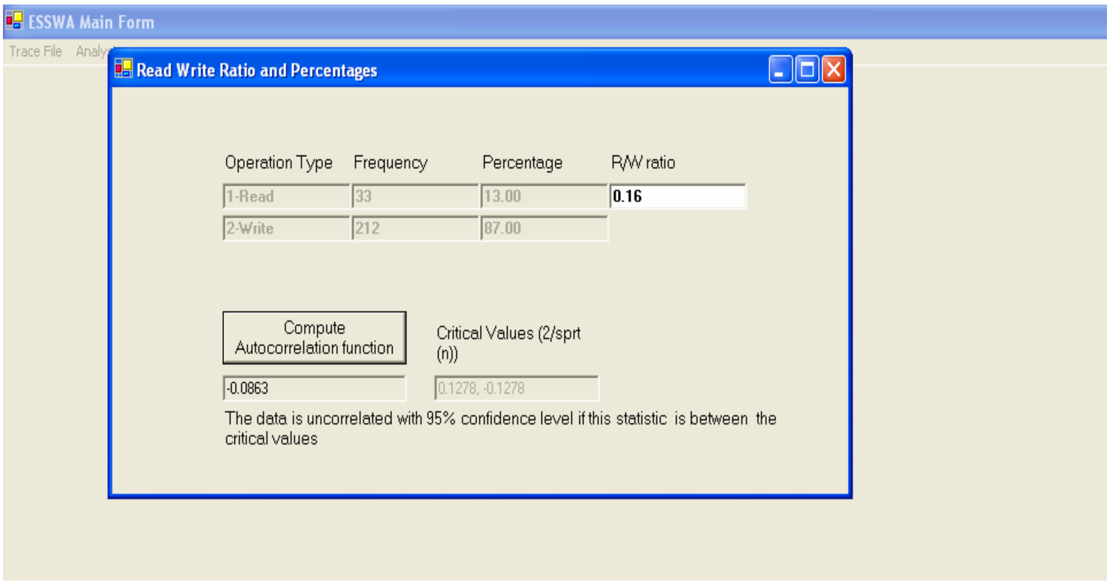
Figure 5.9: Logical volume number frequency table and autocorrelation function window

distances as those for inter-arrival times shown in Figure 5.6. Using the menu presented on this window, the user can move to the second window which provides the same statistics for logical seek distances as those for inter-arrival times shown in Figure 5.8. The user can further move on to the third window which provides a frequency table of absolute logical seek distances.

- **Operation type.** When the user clicks on the *Operation Types* menu item, the window shown in Figure 5.10 is presented with a frequency table of the operation types and the read/write ratio. This window also provides controls for obtaining the autocorrelation value for the operation type data sets.

- **Parallelism degree.** The work flow of analyzing parallelism degrees is similar to request sizes. The first window presents key data statistics for parallelism degrees, except the tail index and the autocorrelation value, and the menu items for displaying the parallelism degrees visually. The second window provides a frequency table, the tail index and the autocorrelation value for the parallelism degrees.

- **Correlation.** The last menu item under the *Analysis* menu takes the user to the window shown in Figure 5.11 for calculating the coefficient of correlation between any two workload parameters. On this window there are two groups of radio


ESSWA Main Form
Trace File
Analy
Read Write Ratio and Percentages
Compute Autocorrelation function

Operation Type	Frequency	Percentage	R/W ratio
1-Read	33	13.00	0.16
2-Write	212	87.00	

	Critical Values (2/sprt (n))
-0.0863	0.1278, -0.1278

The data is uncorrelated with 95% confidence level if this statistic is between the critical values

Figure 5.10: Operation type frequency table and autocorrelation function window

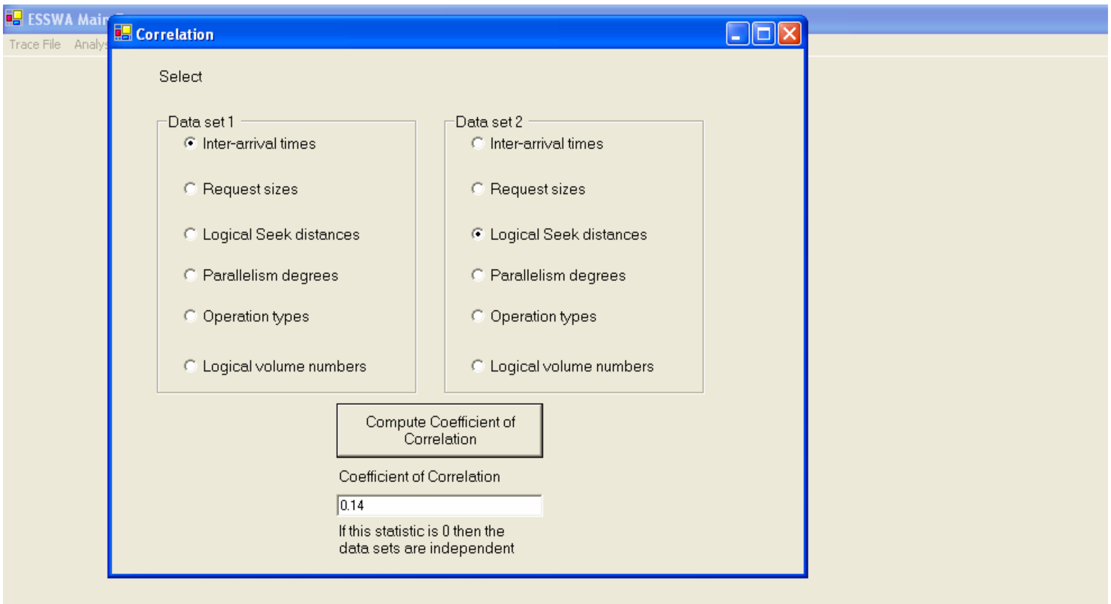
buttons. Each parameter under consideration is presented on this window with a radio button in both groups. The user selects the two parameters, one from each group, for which to calculate the coefficient of correlation by ticking the respective radio buttons. After selecting the parameters, the user must click on the button labelled *Compute Coefficient of Correlation* to obtain the coefficient of correlation.

5.3.4 More Information about ESSWA.

A. Robustness.

ESSWA is a robust system in that it is able to prevent some erroneous conditions from occurring and handle erroneous conditions that it can not prevent. The system performs some validations on the data entered by users to achieve the first point. For example, the system does not accept a letter on the field where an integer, say the logical volume number in Figure 5.5, should be entered. The system catches exceptions and provides meaningful error messages to the user to achieve the second point. For example, if the system fails to read a trace file because the user specified a wrong field delimiter, the user will be notified accordingly.

Furthermore, ESSWA provides messages to keep the user informed about the status of the computations. For example, when the system is computing the tail index, a



Correlation

Select

Data set 1
● Inter-arrival times
○ Request sizes
○ Logical Seek distances
○ Parallelism degrees
○ Operation types
○ Logical volume numbers

Data set 2
○ Inter-arrival times
○ Request sizes
● Logical Seek distances
○ Parallelism degrees
○ Operation types
○ Logical volume numbers

Coefficient of Correlation
0.14
If this statistic is 0 then the data sets are independent

Figure 5.11: Correlation window

message is displayed saying that the computation is in progress. When it is finished, the user is informed accordingly and the result is displayed.

B. Resource Requirements

- **Memory.** As already mentioned in Chapter 1, trace files need huge amounts of disk space to store and RAM to process. The sizes of I/O trace files that we had ranged from 15MB to 1.5GB with a total volume of over 10GB. About 50 percent of the files had sizes less than 50MB. On average, a trace file of size 50MB represented a workload with about 2 million I/O requests recorded over a period of about 3 to 4 hours. The machine on which we were running ESSWA had enough disk space to store the traces. The CPU in this machine had 512MB of RAM and its speed was 1.7GHz. With this memory, the system could read trace files whose sizes were equal or less than 50MB without running out of RAM. The time ESSWA took to read a trace file ranged from 7 minutes (15MB) to 1 hour 30 minutes (50MB).

- **CPU time.** Some of the computations performed by ESSWA are CPU intensive. Particularly, the tail index and the λ^2 discrepancy statistics require more CPU time to compute than any other statistic. For example, given a data set of 1,055,448 inter-arrival times, ESSWA, running on our machine, took about 20 minutes to calculate the λ^2 discrepancy statistics for the seven probability density functions under consideration and 10 minutes to calculate the tail index. The total time taken to compute all the other statistics for the same data set was about 12 minutes.

5.4 Comparisons between ESSWA and Rubicon

Both ESSWA and Rubicon basically read I/O traces describing the I/O requests and produce output in form of data statistics. However, each one of them offers different additional functionalities. For example Rubicon has a feature for determining the self-similarity of I/O events in a given workload while this functionality is lacking in ESSWA. On the other hand, ESSWA has a feature for finding a mathematical model for inter-arrival times in a given workload which is not included in Rubicon. There are other advantages which ESSWA and Rubicon have over each other. In the following sections we present some of these advantages.

5.4.1 Advantages of ESSWA over Rubicon

ESSWA has a number of advantages over Rubicon.

- It takes advantage of some of the statistical and graphical techniques of R language, which is a free, powerful and proven environment.

- It is GUI driven. This means that it is easier to use.

- Rubicon is more complicated than ESSWA and we think that its advantages do not make up for this complexity. While ESSWA is simple, it meets most of the requirements which Rubicon addresses. It is equally extendible. For example, we were able to add a function to filter traces for a particular logical volume number easily. It provides some functionality for trace manipulation, such as filtering a trace for a given logical volume and appending a trace to one which has been read already. It also provides multiple reporting formats (i.e., statistics and graphs) and the back-end which contains the code for analysis tasks is a single program.

- It is based on the client/server architecture. This means that the ESSWA front- and back-end can run on separate machines and that more than one front-end can use a single back-end.

- Unlike Rubicon which is mainly meant for workload traces collected from systems running on HP-UX[Vktr03], ESSWA is able to analyze storage system workloads from any system.

5.4.2 Advantages of Rubicon over ESSWA

Rubicon has more features for filtering traces than ESSWA. Other than that, it has certain features and capabilities such as for performing staged analysis and capturing information about the system from which the traces were collected which ESSWA does not have at the moment. Rubicon is also a more general tool in that it can be used to analyze network traffic, file system and storage system workloads. ESSWA is only meant for analyzing storage system workloads.

Chapter 6

Results

This chapter discusses some of the results we obtained from the analysis of three different storage system workloads using ESSWA. For each of the workload parameter data sets from all the three workloads, we present some key data statistics and visual displays and suggest a mathematical model in the form of a probability distribution function. Note that the *logical seek distance* analysis results presented are for a single logical volume, whereas the analysis results for all the other parameters are for all the logical volumes together, in a given workload. The last section of this chapter presents the results on auto- and cross-correlation of the workload parameters for all the three workloads.

The first storage system workload that we analyzed is represented by a trace from SPC. The trace was collected from an online transaction processing (OLTP) system. It contains records for 549,320 I/O requests. The second workload consists of 535,233 I/O requests also in the form of a trace from SPC. This trace was collected from a system running a web search engine.

The third workload consists of 228,460 I/O requests in the form of a trace from the HP trace repository. The set of traces from which we got this trace was collected from an HP-UX time-sharing server attached to several disk arrays which was instrumented to record all I/O system calls. We could not analyze this workload in terms of the *logical volume number* and *start address* (i.e., logical seek distance) workload parameters because there are no data for these parameters in the traces.

In all the three workloads, the timestamps at which the system calls were made to the storage systems were recorded at micro-second accuracy. We subsequently refer to these workloads as the *OLTP*, *Web* and *HP* workloads, respectively.

Logical Volume Number	Count	Frequency (%)
0	11,417	2.08
1	29,097	5.3
2	25,438	4.63
3	28,622	5.21
4	29,322	5.34
5	17,019	3.10
6	25,201	4.59
7	5,288	0.96
8	60,252	10.97
9	57,771	10.52
10	73,440	13.37
11	2,491	0.45
12	8,011	1.46
13	18,935	3.45
14	16,315	2.97
15	4,015	0.73
16	16,168	2.94
17	245	0.04
18	5,399	0.98
19	16,368	2.98
20	94,514	17.21
21	3,200	0.58
22	410	0.07
23	382	0.07

Table 6.1: Frequency table of OLTP logical volume numbers

6.1 Logical Volume Number

6.1.1 OLTP Logical Volume Number

In the OLTP workload we analyzed the results show that there are 24 logical volumes represented by numbers 0, 1, 2, ..., 23. Table 6.1 is the frequency table of the OLTP logical volume numbers. From the table we can see that the load is not uniform over the logical volumes. The workload is high in logical volumes 8, 9, 10 and 20. Each of these logical volume numbers has a frequency that is more than 10%. Logical volumes 1, 2, 3, 4, 5, 6, 12, 13, 14, 16, and 19 have between 1 and 5% of accesses each. The rest have less than 1% each. Figure 6.1 shows the same information in the form of a histogram.

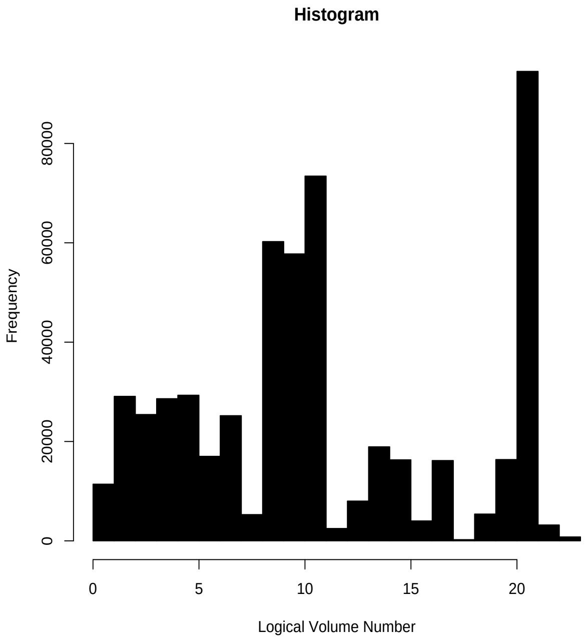


Figure 6.1: Histogram of OLTP logical volume numbers

6.1.2 Web Logical Volume Number

Table 6.2 shows the frequency table of Web logical volume number data set. There are 6 logical volumes involved in this workload. Like the OLTP, the Web workload is not equally shared among the 6 logical volumes. See also Figure 6.2. The frequency of logical volume number 0 is 33.76%, 1 is 33.12% and 2 is 32.05%. The other three logical volume numbers has a total frequency equal to 0.6% (i.e., 0.2% each).

Logical Volume Number	**Count**	**Frequency (%)**
0	180,693	33.76
1	177,287	33.12
2	176,917	32.05
3	107	0.02
4	117	0.02
5	112	0.02

Table 6.2: Frequency table of Web logical volume numbers

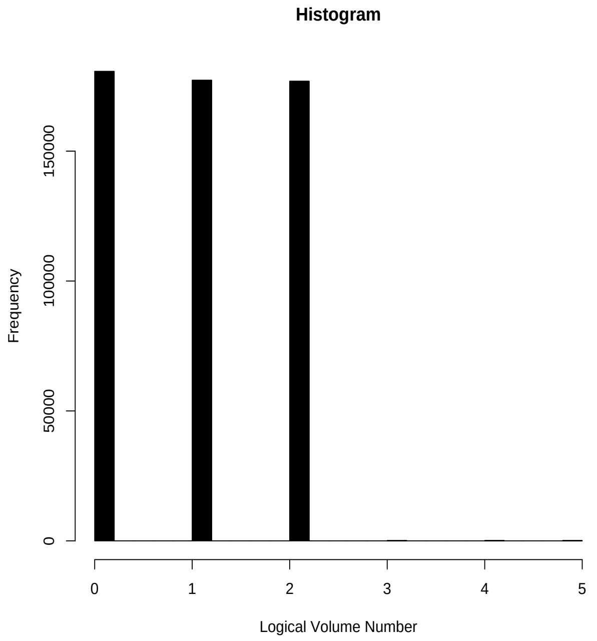


Figure 6.2: Histogram of Web logical volume numbers

6.1.3 Modelling Logical Volume Number

We can define a random variable X as the Logical volume number of an I/O request to model the logical volume number parameter for a storage system workload, say the Web workload. Using Table 6.2, the probability mass function of X can be defined as:

$$\begin{aligned} p(x) &= 0.3376 \;\; for \;\; x = 0, \\ p(x) &= 0.3312 \;\; for \;\; x = 1, \\ p(x) &= 0.3205 \;\; for \;\; x = 2, \\ p(x) &= 0.0002 \;\; for \;\; x = 3, \\ p(x) &= 0.0002 \;\; for \;\; x = 4, \\ p(x) &= 0.0002 \;\; for \;\; x = 5, \\ &= 0 \quad otherwise \end{aligned} \tag{6.1}$$

This probability mass function can be used to generate values for logical volume numbers, representative of those of the Web workload, for a synthetic workload.

6.2 Inter-arrival Time

6.2.1 OLTP Inter-arrival Time

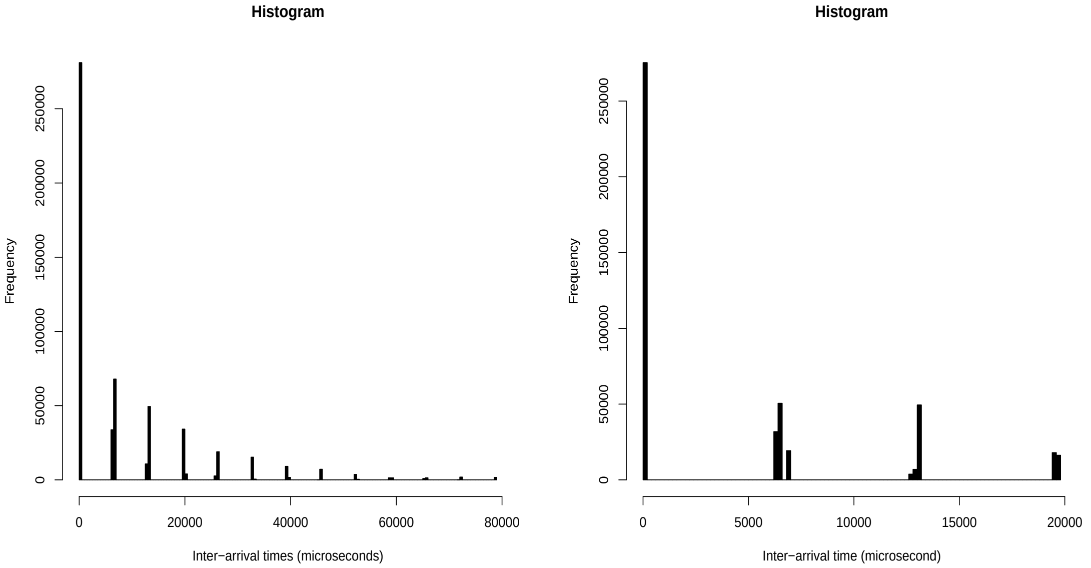
The second storage system workload parameter that we analyzed is the inter-arrival time. Table 6.3 gives the key statistics in microseconds for the OLTP inter-arrival time data set.

Statistic	**Value**
Sample Size	549,319
Minimum	0
Lower Quartile	0
Median	0
Sample Mean	10,530
Upper Quartile	13,180
Maximum	4,017,000
Upper Outliers >	79,080
Sample Variance	929,506,412.86
Sample Standard Deviation	30,487.81
Coefficient of Variation	2.89
Coefficient of Skew	38.03
Coefficient of Kurtosis	2,934.74
Tail index	0.55

Table 6.3: OLTP inter-arrival time statistics

From the table we can see that the minimum value of the OLTP inter-arrival time data set is $0\mu s$. The fact that the minimum inter-arrival time is $0\mu s$ implies that there are some I/O requests issued at the same time, i.e., I/O requests issued in parallel. The analysis of the degree of I/O parallelism is discussed separately in Section 6.6. From the table we can see that the median is also $0\mu s$. This means that at least 50% of the inter-arrival times have value zero. We can conclude from this that there is a lot of I/O parallelism in this workload. The mean value is 10,530μs which is much higher than the median. This is because a portion (25%) of inter-arrival times contains relatively very large values (between 13,180 and 4,017,000μs) pulling up the mean.

The coefficient of skew for OLTP inter-arrival times is 38.03. This indicates that the distribution of this data set is not symmetric but positively skewed. The coefficient of kurtosis which is 2,934.74 indicates that the distribution of OLTP inter-arrival times is heavy tailed. This is confirmed by the tail index which is 0.5544. The fact that the distribution of inter-arrival times is heavy tailed means that the probability of having a relatively long period between I/O requests (i.e., idle time) is quite high. This idle time can be utilized by the storage system handling this workload to perform background



(a) (b)

Figure 6.3: (a) Histogram of OLTP inter-arrival times without outliers (b) Histogram of OLTP inter-arrival times less than 20,000

tasks. The peakedness and unsymmetrical characteristics of the distribution of OLTP inter-arrival times can also be detected using a histogram. Figure 6.3(a) shows the histogram of OLTP inter-arrival times without outliers. From this histogram one can see that OLTP inter-arrival times are restricted to certain ranges of values. We looked at this behavior closely by drawing a histogram of OLTP inter-arrival times less than 20,000μs shown in Figure 6.3(b). Further investigations showed that they are certain values in the data set with frequencies 1, 2 and 3%, apart from zero whose frequency is 50%. We found this to be a bit strange because inter-arrival time is a continuous parameter and we expected the frequency (i.e., the probability) of each value in the data set to be very close to zero. However, we have no explanation for this behavior.

6.2.2 Web Inter-arrival Time

Table 6.4 presents some of the key statistics of the Web inter-arrival time data set in microseconds. The minimum inter-arrival time is 126μs. This means that in the Web workload there is no I/O parallelism. Figure 6.4 shows the histogram of the Web inter-arrival times without outliers.

The Web inter-arrival time data set is distributed over a very large range in the

Statistic	**Value**
Sample Size	535,232
Minimum	126
Lower Quartile	242
Median	1,697
Sample Mean	2,992
Upper Quartile	4,489
Maximum	100,100
Upper Outliers >	18,449
Sample Variance	12,640,115.12
Sample Standard Deviation	3,555.29
Coefficient of Variation	1.19
Coefficient of Skew	2.21
Coefficient of Kurtosis	10.17
Tail index	0.21

Table 6.4: Web inter-arrival time statistics

interval (126, 100,100) but less than that of the OLTP inter-arrival times. According to the value of the upper quartile, the bulk of values are in the interval (126, 4,489). The five number summary indicates a severe skew to the right in the distribution of the data. The range between the minimum and the median is 1,571 compared to that of the median and the maximum, which is 98,403. The values for the tail index, coefficient of kurtosis and coefficient of variation confirm the large skew to the right and indicate highly variable data just like the OLTP inter-arrival times. However, the histograms show that Web inter-arrival time distribution is less peaked than that of OLTP. This is confirmed by the fact that the coefficient of kurtosis for the latter is greater.

6.2.3 HP Inter-arrival Time

Table 6.5 shows the key data statistics of the HP inter-arrival times. Like in the OLTP workload, the minimum inter-arrival time is $0\mu s$. The maximum inter-arrival time is 601,300,000μs and is far greater than the maximum inter-arrival times in both the OLTP and Web workloads. Figure 6.5 shows the histogram of HP inter-arrival times without outliers.

6.2.4 Modelling Inter-arrival Time

Since storage system workload inter-arrival time is a continuous parameter, we used the goodness-of-fit statistic to find the best probability density functions which match the inter-arrival time data sets. Table 6.6 gives the three probability density functions

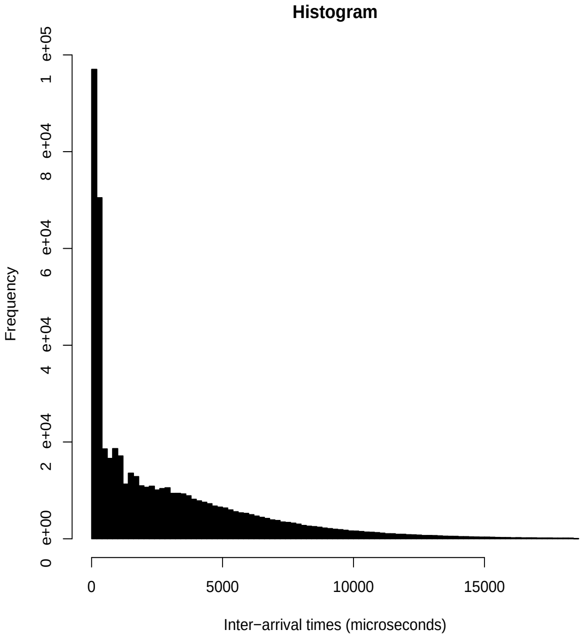


Figure 6.4: Histogram of Web inter-arrival times without outliers

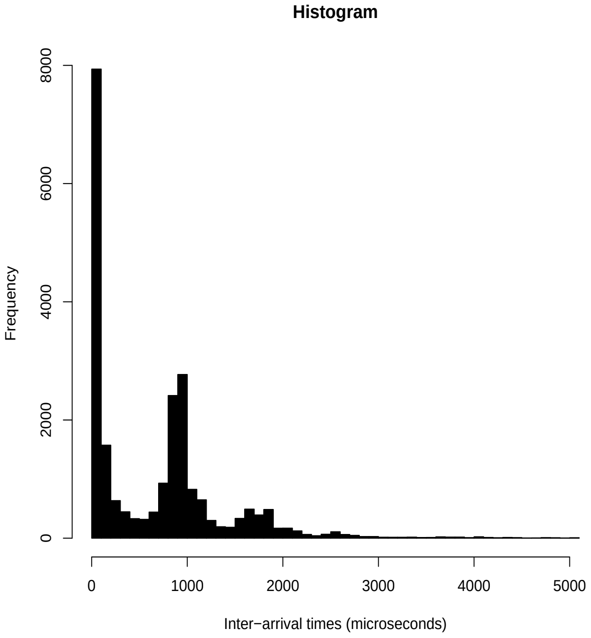


Figure 6.5: Histogram of HP inter-arrival times without outliers.

Statistic	**Value**
Sample Size	228,459
Minimum	0
Lower Quartile	132
Median	888
Sample Mean	32,940
Upper Quartile	1,592
Maximum	601,300,000
Upper Outliers >	5,112
Sample Variance	15,829,789,332,189.10
Sample Standard Deviation	3,978,666.78
Coefficient of Variation	120.79
Coefficient of Skew	151.11
Coefficient of Kurtosis	22,837.58
Tail index	1.94

Table 6.5: HP inter-arrival time statistics

(i.e., out of the seven that we considered) that best fit the OLTP inter-arrival time data set. Among the three functions in the table, the first function is the best fit followed by the second and then the third. The table also includes the estimated value(s) for the attribute(s) of each of the three functions and the value of the corresponding λ^2 discrepancy statistic used to determine how well the match is.

As already mentioned in Section 4.3.7, the λ^2 discrepancy test is not infallible. We generated[1] three data sets following the three probability density functions in the table using the corresponding value(s) for the attribute(s), i.e., each one of the three data sets following one of the three functions. Then we drew the ECDFs of these data sets together with the OLTP inter-arrival time data set to ascertain the matches. See Figure 6.6. The graphs confirm that the Weibull probability function is the best fit. This probability density function together with the estimated values for its attributes can be used to reproduce OLTP inter-arrival times for a synthetic workload.

Function Name	λ^2	**Attribute Values**
Weibull	0.730	Gamma = 0.293, Alpha = 5,162.097
Gamma	1.046	Shape = 0.208, Rate = 0
Exponential	1.188	Beta = 18,170.293

Table 6.6: The three best probability distribution functions matching the OLTP inter-arrival times

[1] R has functions, which ESSWA uses, that can generate values following the probability density functions that we used in our study.

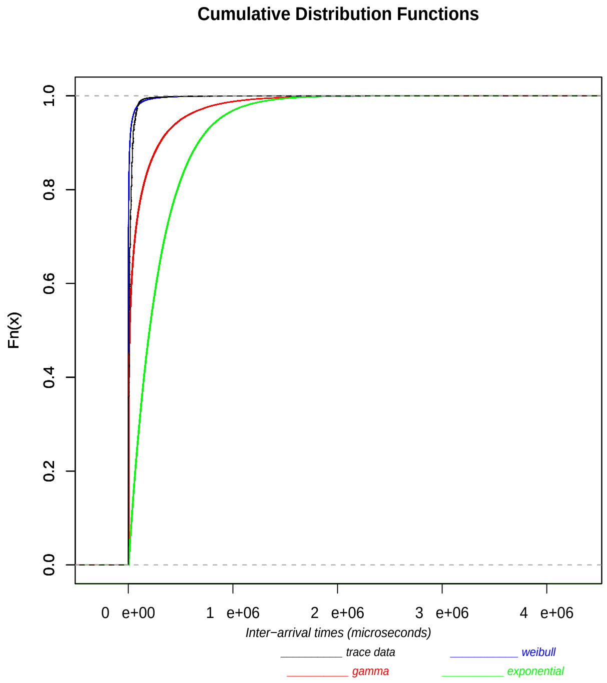


Figure 6.6: ECDF's of OLTP inter-arrival times and generated data sets

Table 6.7 contains the best three probability density functions fitting the Web inter-arrival times based on the λ^2 discrepancy statistic. These are the lognormal, Weibull and beta probability density functions. Of the three, the lognormal probability density function is the best fit followed by the Weibull and then, of course, the beta. We also confirmed this using ECDFs. See Figure 6.7. Note that Hsu *et al*[Hsco03] also found the lognormal distribution to be a better model for the inter-arrival time in the workloads they analyzed.

Function Name	λ^2	**Attribute Values**
Lognormal	0.045	Zeta = 6.978, Sigma = 1.488
Weibull	0.050	Gamma = 0.758, Alpha = 2,241.749
Beta	0.087	Shape1 = 0.456, Shape2 = 17.635

Table 6.7: The three best probability distribution functions matching the Web inter-arrival times

Based on the λ^2 discrepancy statistics, the best three probability density functions that fit the HP inter-arrival times are beta, normal and exponential probability density functions (see Table 6.8). However, from Figure 6.5, it is clear that none of the usual probability distribution functions would fit the HP inter-arrival times reasonably well. This is confirmed by the ECDFs in Figure 6.8. The figure shows that the exponential

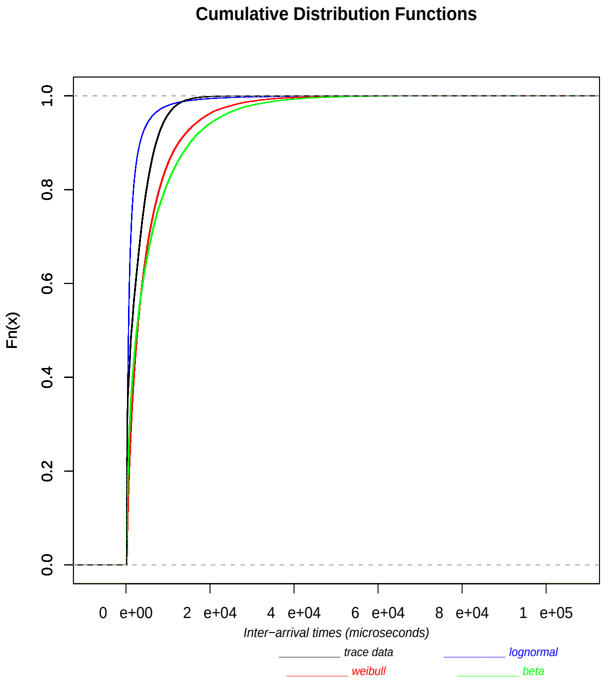


Figure 6.7: ECDF's of Web inter-arrival times and generated data sets

distribution matches the distribution of the HP inter-arrival times better than the normal distribution contrary to what the λ^2 discrepancy statistics indicate.

Function Name	λ^2	**Attribute Values**
Beta	0.149	Shape1 = 0.134 Shape2 = 342.332
Normal	18.916	Mean = 32,937.849, Standard Deviation = 3,978,579.702
Exponential	19.98	Beta = 1,000

Table 6.8: The three best probability distribution functions matching the HP inter-arrival times

6.3 Request Size

The third storage system workload parameter which we analyzed is the request size. This parameter, combined with the inter-arrival time constitutes what is called the *intensity distribution* of an I/O workload as opposed to the *locality distribution* defined by logical seek distances of I/O requests[Pgsk05].

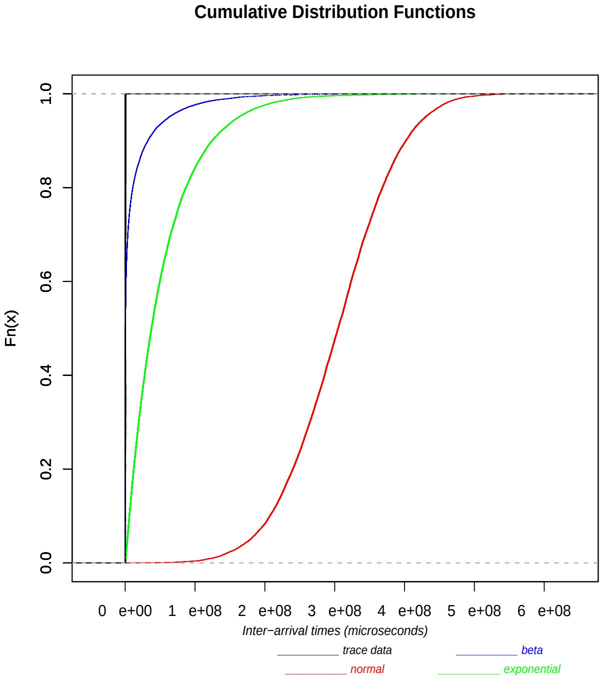


Figure 6.8: ECDF's of HP inter-arrival times and generated data sets

6.3.1 OLTP Request Size

Table 6.9 shows the key statistics of the OLTP request sizes in bytes. In the table we can see that the sample mean is greater than the upper quartile. This means that although at least 75% of the request sizes in the workload are less than 3,584, the mean is increased by values between the upper quartile and the maximum, (i.e., between 6,144 and 3,224,000) which forms only 25% of the data set.

The distribution of OLTP request sizes is multimodal (i.e., there are a number of peaks). After removing the few outlying values (i.e., values larger than 6,144), the histogram of the remainder of the data contains ten peaks at values 512, 1,024, 1,536, 2,048, 3,072, 3,584, 4,096, 4,608, 7,680 and 8,192 such that 91% of the I/O requests had one of these sizes (see Figure 6.9 and Table 6.10). The reason for this is not difficult to understand. In operating systems such as UNIX, both *read()* and *write()* system calls typically make use of the concept of a *Filesystem Block*. A Filesystem Block is a unit of data that an operating system typically uses when reading from or writing data to physical devices. It is configurable and most operating systems use a setting of 8,192 bytes. Disk reads and writes are typically not multiples of 8,192 bytes, and disk space is therefore wasted when only a few bytes are written to a Filesystem Block of 8,192 bytes. Therefore, most operating systems allow reads and writes of certain sizes smaller than 8,192 bytes, such as 512, 1,024, and so on, to be performed in order to

Statistic	**Value**
Sample Size	549,320
Minimum	512
Lower Quartile	1,024
Median	3,072
Sample Mean	5,175
Upper Quartile	3,584
Maximum	3,224,000
Upper Outliers >	6,144
Sample Variance	168,016,914.38
Sample Standard Deviation	12,962.13
Coefficient of Variation	2.50
Coefficient of Skew	66.40
Coefficient of Kurtosis	13,929.30
Tail index	1.70

Table 6.9: OLTP request size statistics

conserve disk space[Pgsk05].

Size	**Count**	**Frequency (%)**
512	102,534	19
1,024	58,262	11
1,536	57,090	10
2,048	23,894	4
3,072	100,511	18
3,584	65,852	12
4,096	29,928	5
4,608	9,975	2
7,680	14,705	3
8,192	39,761	7

Table 6.10: Frequency table of OLTP request sizes

6.3.2 Web Request Size

Table 6.11 contains the key statistics of the Web request size data set in bytes. From the table, we noted that the maximum request size in this data set is less than that of the OLTP request sizes. The maximum request size is 1,138,000 bytes while in the OLTP workload it is 3,224,000 bytes. The minimum sizes in both workloads are 512 bytes and it follows that the range of values of request sizes in the OLTP workload is greater (i.e., OLTP request sizes are more widely spread). This can also be seen in the difference between the standard deviations of the two data sets. On the other hand, the

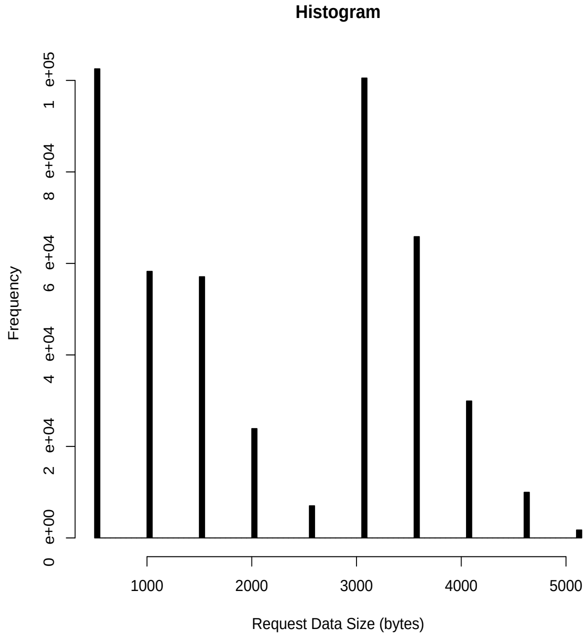


Figure 6.9: Histogram of OLTP peak request sizes

mean and median of Web request sizes are greater. Another notable difference between the two workloads in terms of request sizes is that the distribution of the OLTP request sizes after removing outliers has ten peaks while that of the Web request sizes has only four peaks at values 8,192, 16,384, 24,576 and 32,768 bytes which are all multiples of 8,192. These peaks are illustrated in Figure 6.10 and Table 6.12.

6.3.3 HP Request Size

Table 6.13 and Table 6.14 show the key data statistics and the frequency table of the HP request sizes respectively. Both the minimum (4 bytes) and the maximum (262,100 bytes) request sizes for the HP workload are less than the corresponding statistics in the other two workloads. From the frequency table we can see that the frequency of 8,192 (41%) is much higher than of any other value and that at least 62% of the sizes are between 4 and 32,768 bytes. Figure 6.11(a) shows the histogram of the HP workload request sizes. This figure shows that most of the sizes are less than or equal to 32,768 bytes. About 91% of the values in the data set are between 4 and 32,768. The histogram of values in this range is shown in Figure 6.11(b). We also noticed from the histogram (i.e., in Figure 6.11(a)) and the statistics that there are fewer sizes between 32,768 and 200,000 bytes than between 225,000 and 262,100 bytes.

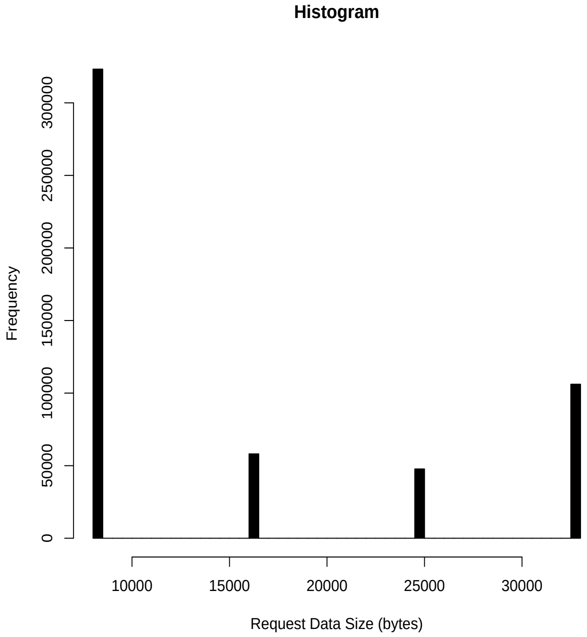
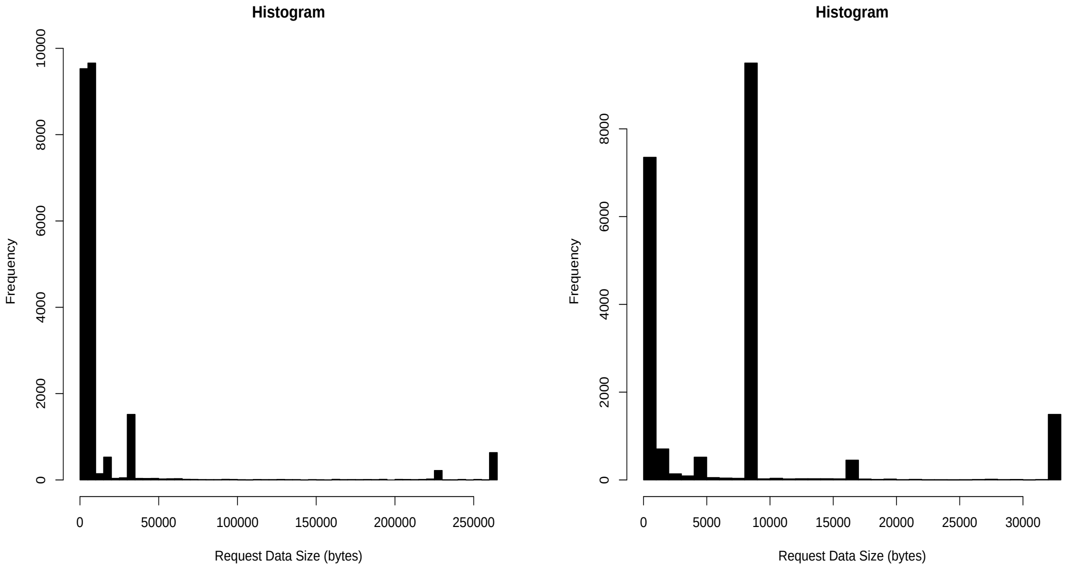


Figure 6.10: Histogram of Web request sizes



(a) **(b)**

Figure 6.11: (a) Histogram of HP request sizes (b) Histogram of HP request sizes between 4 and 32,768

Statistic	**Value**
Sample Size	535,233
Minimum	512
Lower Quartile	8,192
Median	8,192
Sample Mean	15,420
Upper Quartile	24,580
Maximum	1,138,000
Upper Outliers >	106,520
Sample Variance	103,512,492.07
Sample Standard Deviation	10,174.11
Coefficient of Variation	0.66
Coefficient of Skew	5.85
Coefficient of Kurtosis	552.06
Tail index	1.18

Table 6.11: Web request size statistics

Size	**Count**	**Frequency (%)**
8,192	323,209	60
16,384	58,136	11
24,576	47,743	9
32,768	106,119	20

Table 6.12: Frequency table of Web request sizes

6.3.4 Modelling Request Size

The best way to model this parameter would be to simply use a probability mass function, whose random variable X is the size of an I/O request. For example, using the probabilities in Table 6.10, the probability mass function for the OLTP request sizes, can be defined as:

$$\begin{aligned}
p(x) &= 0.19 \;\; for \;\; x = 512, \\
p(x) &= 0.11 \;\; for \;\; x = 1024, \\
p(x) &= 0.04 \;\; for \;\; x = 2048, \\
p(x) &= 0.01 \;\; for \;\; x = 1536, \\
p(x) &= 0.018 \;\; for \;\; x = 3072, \\
p(x) &= 0.12 \;\; for \;\; x = 3584, \\
p(x) &= 0.05 \;\; for \;\; x = 4096,
\end{aligned}$$

Statistic	**Value**
Sample Size	228,460
Minimum	4
Lower Quartile	89
Median	8,192
Sample Mean	18,540
Upper Quartile	8,192
Maximum	262,100
Upper Outliers >	8,192
Sample Variance	2,594,147,998.30
Sample Standard Deviation	50,932.78
Coefficient of Variation	2.75
Coefficient of Skew	4.11
Coefficient of Kurtosis	15.83
Tail index	0

Table 6.13: HP request size statistics

Size	**Count**	**Frequency (%)**
4	9138	4
16	4569	2
56	9138	4
256	6854	3
4,096	4569	2
8,192	93,669	41
32,768	13,707	6
Other Sizes (< 1% each)	86,815	38

Table 6.14: Frequency table of HP request sizes

$$\begin{aligned} p(x) &= 0.02 \quad for \quad x = 4608, \\ p(x) &= 0.03 \quad for \quad x = 7680, \\ p(x) &= 0.07 \quad for \quad x = 8192, \\ &= 0 \quad otherwise \end{aligned} \tag{6.2}$$

6.4 Operation Type

6.4.1 OLTP Operation Type

The OLTP workload has more write than read operations as shown in Table 6.15. The read/write ratio for the OLTP workload is 0.34. This could be due to the fact that most read operations are absorbed by the file system level cache[Hsco03].

Operation Type	**Count**	**Frequency (%)**
Read	139,035	25
Write	410,285	75

Table 6.15: Frequency table of OLTP operation types

6.4.2 Web Operation Type

Table 6.16 shows the frequency table for the Web operation type data set. The read/write ratio for this workload is 5,048.37. As can be see from the frequency table almost all the operations are reads (i.e., 99.98% of the total number of operations). This is understandable as a Web search application does not typically write much data to secondary storage.

Operation Type	**Count**	**Frequency (%)**
Read	535,127	99.98
Write	106	0.02

Table 6.16: Frequency table of Web operation types

6.4.3 HP Operation Type

Table 6.17 shows the frequency table for the HP operation type data set. The read/write ratio for the HP workload is 1.13.

Operation Type	**Count**	**Frequency (%)**
Read	121,084	53.00
Write	107,376	47.00

Table 6.17: Frequency table of HP operation types

6.4.4 Modelling Operation Type

Assuming that the operation type parameter of a given workload is not auto-correlated, a random variable defined as the operation type of an I/O request in that particular workload is a binomial process. For example, assuming that the OLTP operation type is not autocorrelated, we can define a random variable, say X, as the operation type of an I/O request in the OLTP workload whose probability distribution is a binomial function with the probability, p, of a success (i.e., a read) equal to 0.25. Using this probability mass function a data set similar to the OLTP operation type data set can be generated. R has a function which given p, and a positive integer, say n, will generate n values following the binomial process. With $p = 0.25$ and $n = 549,320$, we generated a data set of operation types of the same size as the OLTP operation type data set. Thereafter we calculated the read/write ratio of the generated data set and got 0.33 which is quite close to the 0.34 of the OLTP operation type data set.

6.5 Logical Seek Distance

As already defined in Section 1.1, logical seek distances are the differences between consecutive start addresses, either for all logical volumes together or for just one logical volume in a given storage system workload. Note that we analyzed start addresses in this way solely for modelling purposes. That is to come up with models in the form of probability distribution functions that can be used to generate start addresses for synthetic workloads. Contrary to what some people might expect, analyzing start addresses in this way is not so useful in understanding the locality distribution of the data stored in a storage system. This is due to the fact that a single logical volume may be spread over several physical volumes. Therefore two consecutive start addresses in a trace for the same logical volume may be referring to two different physical volumes and two consecutive start addresses for different logical volumes may be referring to the same physical volume. In short, we can not tell how the data is distributed among the physical volumes using the logical seek distances.

Below are the analysis results of OLTP logical seek distances for logical volume number 20 and Web logical seek distances for logical volume number 0.

6.5.1 OLTP Logical Seek Distance

Table 6.18 gives the key statistics of the OLTP logical seek distances measured in blocks for logical volume number 20. The distribution of this data set is positively skewed with the coefficient of skew value of 0.02. The histogram of the data set without outliers is shown in Figure 6.12. Table 6.19 shows a frequency table of the absolute values of the data set. We have presented the absolute values to minimize the number of intervals/bins over which the logical seek distances are split.

6.5.2 Web Logical Seek Distance

Table 6.20 gives the key statistics of the Web logical seek distances measured in blocks for logical volume number 0. The coefficient of skew is zero meaning that the distribution of this data set is symmetric. The distribution of logical seek distances for all the

Statistic	Value
Sample Size	94,514
Minimum	-500,400
Lower Quartile	2
Median	7
Sample Mean	7.49
Upper Quartile	16
Maximum	500,400
Lower Outliers <	-47
Upper Outliers >	61
Sample Variance	1,307,866,952.63
Sample Standard Deviation	36,164.44
Coefficient of Variation	4,830.83
Coefficient of Skew	0.02
Coefficient of Kurtosis	4.31
Tail index	0.74

Table 6.18: Statistics of OLTP logical seek distances for logical volume 20

Range Number	Range	Count	Frequency (%)
1	0 - 20,000	65,060	68.84
2	20,000 - 40,000	12,655	13.39
3	40,000 - 60,000	5,291	5.6
4	60,000 - 80,000	3,720	3.94
5	80,000 - 100,000	3,677	3.89
6	100,000 - 120,000	4,100	4.34
7	120,000 - 520,000	11	0.01

Table 6.19: Frequency table of absolute OLTP logical seek distances for logical volume 20

Web logical volumes is also symmetric. The ratio of the number of positive logical seek distances to the number of negative logical seek distances is one. We observed that for each positive logical seek distance there is a corresponding negative logical seek distance of the same magnitude. Such values do not necessarily occur in close succession and we have no explanation for this very interesting phenomenon. An explanation may well be found by determining how the search algorithm(s) employed by the application work(s). Table 6.21 is a frequency table of the absolute Web logical seek distances for logical volume number 0. The histogram of the data set without outliers is shown in Figure 6.13.

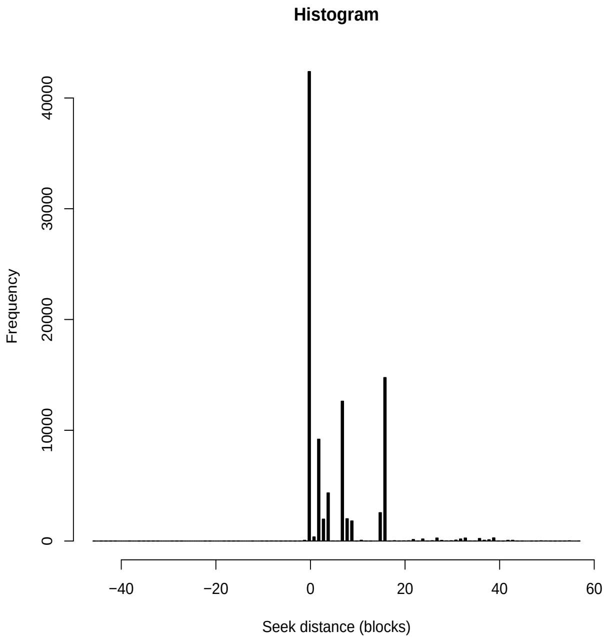


Figure 6.12: Histogram of OLTP logical seek distances for logical volume number 20 without outliers

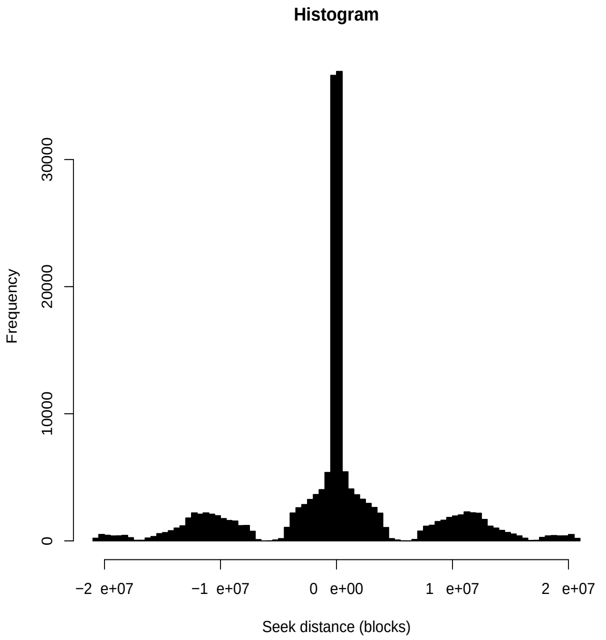


Figure 6.13: Histogram of Web logical seek distances for logical volume number 0 without outliers

Statistic	**Value**
Sample Size	180,693
Minimum	-34,930,000
Lower Quartile	-3,449,000
Median	64
Sample Mean	311.3
Upper Quartile	3,452,000
Maximum	34,940,000
Lower Outliers <	-20,711,552
Upper Outliers >	20,711,680
Sample Variance	181,047,111,999,023
Sample Standard Deviation	13,455,374.84
Coefficient of Variation	43,224.22
Coefficient of Skew	0
Coefficient of Kurtosis	1.16
Tail index	0

Table 6.20: Statistics of Web logical seek distances for logical volume 0

6.5.3 Modelling Logical Seek Distance

Since the logical seek distance parameter is discrete, it is appropriate to use a probability mass function to model it. Statistics for both the OLTP and Web logical seek distances show that the range of values that the logical seek distance can take on is very large. As a results defining a random variable as the logical seek distance of an I/O request and formulating a probability mass function with all the possible values is not feasible. Therefore we suggest two approaches that can be used to model the logical seek distance parameter.

A. First Approach

In the first approach we suggest that a random variable, X, should be defined as the range number in which the logical seek distance of an I/O request will fall and then define a probability mass function in terms of X. For example, we can define a probability function of X, using Table 6.21, to model the absolute Web logical seek distances as:

$$\begin{aligned} p(x) &= 0.4055 \quad for \quad x = 1, \\ p(x) &= 0.1216 \quad for \quad x = 2, \\ p(x) &= 0.0146 \quad for \quad x = 3, \end{aligned}$$

Range Number	**Range**	**Count**	**Frequency (%)**
1	0 - 2,000,000	73,276	40.55
2	2,000,000 - 4,000,000	21,974	12.16
3	4,000,000 - 6,000,000	2,633	1.46
4	6,000,000 - 8,000,000	4,125	2.28
5	8,000,000 - 10,000,000	12,363	6.84
6	10,000,000 - 12,000,000	16,902	9.35
7	12,000,000 - 14,000,000	12,246	6.78
8	14,000,000 - 16,000,000	4,809	2.66
9	16,000,000 - 18,000,000	1,162	0.64
10	18,000,000 - 20,000,000	3,279	1.81
11	20,000,000 - 22,000,000	4,000	2.21
12	22,000,000 - 24,000,000	2,677	1.48
13	24,000,000 - 26,000,000	1	0
14	26,000,000 - 28,000,000	0	0
15	28,000,000 - 30,000,000	1,589	0.88
16	30,000,000 - 32,000,000	11,596	6.42
17	32,000,000 - 34,000,000	6,180	3.42
18	34,000,000 - 36,000,000	1,881	1.04

Table 6.21: Frequency table of absolute Web logical seek distances for logical volume number 0

$$\begin{aligned} & ... \\ & ... \\ & ... \\ p(x) &= 0.0104 \quad for \quad x = 18, \\ &= 0 \quad otherwise \end{aligned} \tag{6.3}$$

Similarly a probability function of X can be defined for the entire range of Web logical seek distances and not just the absolute values. To generate n logical seek distances for a synthetic workload representative of the Web workload, one can use this probability function to generate n range numbers. Then for each range number generated, say k, produce a random value within the limits of range k. For instance, if range number 1 has a range (-2,000,000 - 0) in a frequency table such as Table 6.21, then for each value of 1 generated as a range number a logical seek distance value between -2,000,000 and 0 should be produced.

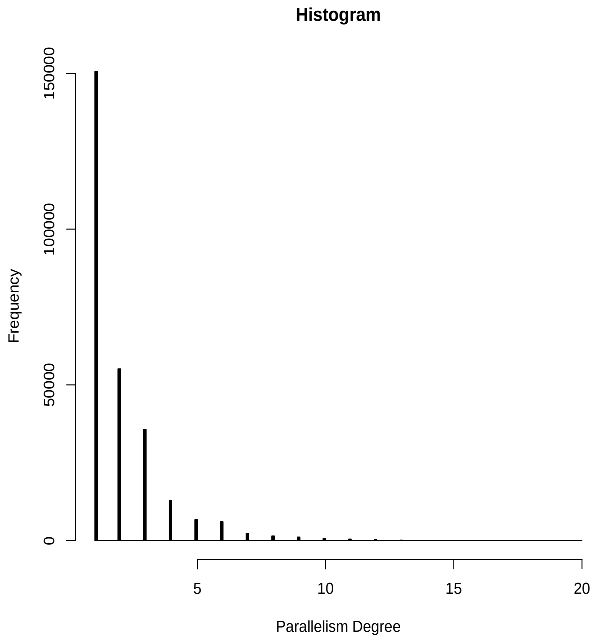


Figure 6.14: Histogram of OLTP parallelism degrees

B. Second Approach

The second approach involves finding a probability density function matching a given data set of logical seek distances as an *approximate* model. This means that we have to presume that the logical seek distance parameter is a continuous variable, after all, it can take on values from a very large range. However, it should be clear from Figures 6.12 and 6.13 that none of the usual probability density functions would fit the logical seek distances for both OLTP and Web. However, it might be possible to model the data sets using a mixture of two or three functions. Modelling by using a mixture of functions is beyond the scope of this study.

6.6 Degree of Parallelism

6.6.1 OLTP Degree of Parallelism

Table 6.22 and Table 6.23 show the key statistics and the frequency table of the OLTP parallelism degree data set, respectively. The minimum number of I/O requests issued at the same time in the workload is 1. From Table 6.23 we can see that 55% of the parallelism degree data set have a value of 1. It is important to realize that the value of 1 actually implies a single I/O request per timestamp (i.e., no I/O parallelism).

After doing some simple arithmetic, one will discover that about 150,690 or 27% of the 549,320 OLTP I/O requests are singles. Therefore 73% of the requests are not singles. 20% of the parallelism degree data have a value of 2, meaning that there are about 54,796 I/O request pairs issued at the same time representing 19% of all the OLTP I/O requests. The maximum OLTP parallelism degree is 20. Figure 6.14 shows the distribution of the parallelism degree data in the form of a histogram. This figure and Table 6.23 show that there are very few values in the parallelism degree data set between 9 and 20. Only 1% of the values in the data set are equal or greater than 9.

Statistic	**Value**
Sample Size	273,983
Minimum	1
Lower Quartile	1
Median	1
Sample Mean	2.00
Upper Quartile	2
Maximum	20
Upper Outliers >	7
Sample Variance	2.66
Sample Standard Deviation	1.63
Coefficient of Variation	0.81
Coefficient of Skew	2.67
Coefficient of Kurtosis	9.97
Tail index	0.09

Table 6.22: OLTP parallelism degree statistics

Size	**frequency (%)**
1	55
2	20
3	13
4	5
5	2
6	2
7	1
8	1
≥9	1

Table 6.23: Frequency table for OLTP parallelism degrees

As already mentioned in Section 6.2 there is no I/O parallelism in the Web workload. On the other hand, the HP workload has very little I/O parallelism, i.e., 99.99% of the I/O requests were issued as singles.

6.6.2 Modelling Degree of Parallelism

Like logical volume number, data size, operation type and logical seek distance, the parallelism degree workload parameter can be modelled as a probability mass function. By defining a random variable, say X, as the number of I/O requests issued at a given timestamp, we can state a probability mass function for the OLTP parallelism degree as:

$$\begin{aligned}
p(x) &= 0.5500 \quad for \quad x = 1, \\
p(x) &= 0.2000 \quad for \quad x = 2, \\
p(x) &= 0.1300 \quad for \quad x = 3, \\
&... \\
&... \\
&... \\
p(x) &= 0.0007 \quad for \quad x = 20, \\
&= 0 \quad otherwise
\end{aligned} \tag{6.4}$$

Parameter	**OLTP,** $r(1)$	**Web,** $r(1)$	**HP,** $r(1)$	**95% Confidence Interval Limits** OLTP:Web:HP
Logical Volume Number	0.1263	0.0778	-	(-0.0027, 0.0027): (-0.0027, 0.0027)
Inter-arrival Times	0.1598	-0.0168	0.1011	(-0.0027, 0.0027): (-0.0027, 0.0027): (-0.0091, 0.0091)
Request Size	0.3576	0.0245	0.0730	(-0.0027, 0.0027) (-0.0027, 0.0027): (-0.0091, 0.0091)
Operation Type	0.3470	0.3867	0.3664	(-0.0027, 0.0027): (-0.0027, 0.0027): (-0.0091, 0.0091)
Logical Seek Distance	-0.4507	0.1741	-	(-0.0027, 0.0027) (-0.0027, 0.0027):
Parallelism Degree	0.0870	-	-	(-0.0038, 0.0038)

Table 6.24: Autocorrelation values

6.7 Auto- and Cross-correlation

6.7.1 Auto-correlation

Table 6.24 contains the autocorrelation values at lag 1 (i.e., $r(1)$), with the corresponding 95% confidence interval limits, for the OLTP, Web and HP workload parameters. As stated in Section 4.3.6, the 95% confidence interval limits are $-2/\sqrt{n}$ and $+2/\sqrt{n}$, where n is the sample size. For a data set that is autocorrelated, $|r(1)| > 2/\sqrt{n}$ at 95% confidence level[Lowa04]. In all the three workloads, the autocorrelation values for all the parameters are outside the 95% confidence intervals. Therefore we can say that these parameters are autocorrelated with 95% confidence level. The implication of this discovery is that the models for each of these parameters should be adjusted to incorporate the autocorrelation behavior. This we have not done and suggest it for future work.

Parameters	Logical Volume Number	Inter-arrival Times	Request Size	Operat-ion Type	Logical Seek Distance	Parall-elism Degree
Logical Volume Number	*	0.02	0.09	-0.06	0.14	-0.03
Inter-arrival Times	0.02	*	0.09	-0.14	-0.04	-0.02
Request Size	0.09	0.09	*	0.07	0.01	-0.01
Operat-ion Type	-0.06	-0.14	0.07	*	0.04	0.06
Logical Seek Distance	0.14	-0.04	0.01	0.04	*	0
Parall-elism Degree	-0.03	-0.02	-0.01	0.06	0	*

Table 6.25: Coefficients of correlation for OLTP workload parameters.

6.7.2 Cross-correlation

Table 6.25, Table 6.26 and Table 6.27 show the coefficients of correlation between any two OLTP, Web and HP workload parameters respectively.

Unlike in the case of auto-correlation where one can compute confidence intervals

Parameters	Logical Volume Number	Inter-arrival Times	Request Size	Operation Type	Logical Seek Distance
Logical Volume Number	*	0.02	0.01	0.02	0.20
Inter-arrival Times	0.02	*	-0.10	0	0.01
Request Size	0.01	-0.10	*	-0.01	0.07
Operation Type	0.02	0	-0.01	*	0.01
Logical Seek Distance	0.20	0.01	0.07	0.01	*

Table 6.26: Coefficients of correlation for Web workload parameters

Parameters	Inter-arrival Times	Request Size	Operation Type
Inter-arrival Times	*	-0.04	-0.04
Request Size	-0.04	*	0.03
Operation Type	-0.04	0.03	*

Table 6.27: Coefficients of correlation for HP workload parameters

which can be used to determine whether a given parameter is auto-correlated or not, we did not come across a similar notion with cross-correlation. Our conclusions on cross-correlation are based on the premise that the closer the absolute coefficient of correlation for two parameters is to 0 the more independent the two parameters are and vice versa[Fdal95, Reki99]. Based on the foregoing and the values in the tables, we can conclude there are little correlations between the parameters for all the three workloads. In the OLTP workload, the correlations between logical seek distances and logical volume numbers, and between inter-arrival times and operation type are the strongest both with 0.14 as the absolute coefficient of correlation. In the Web workload the correlation between the logical seek distances and the logical volume number is the strongest with 0.2 as the coefficient of correlation.

If the correlations between the workload parameters are so minimal, then we can use the models for individual workload parameters to produce synthetic workloads representative of these workloads. Otherwise we would need to come up with multivariate probability distribution functions which is also beyond the scope of this study.

Chapter 7

Conclusion and Future Work

7.1 Conclusion

In this dissertation, we have explained our work which involved analyzing storage system workloads using a statistical methodology. We developed a general software tool, ESSWA, for analyzing the workloads in the form of I/O traces. Given an I/O trace file as input, it provides statistics for the workload parameters as the output. From the statistics one can possibly derive mathematical models, in the form of probability distribution functions, for the workload parameters. ESSWA has a number of advantages over other storage system workload analyzers described in literature in terms of:

- user friendliness,
- extendability,
- flexibility,
- maintainability,
- provision of multiple reporting formats,
- provision of a number of trace manipulation functions such as trace filtering,
- etc.

We experimented with ESSWA and produced some useful results. For example, from the results we obtained, we are able to tell the fraction of I/O requests serviced by each available logical volume; whether there is I/O parallelism or not; and the read/write ratio in a given workload. If the workload is unbalanced among the available logical volumes, the system administrator might be required to reconfigure the storage system to balance the load to improve performance. The fact that a particular workload exhibits I/O parallelism, as is the case with one of the workloads that we analyzed, implies that it would be probability beneficial to implement the I/O rescheduling feature in the underlying storage system to improve performance. The read/write ratio helps in deciding whether to optimize for both read and write operations or just for one of the two. In one of the workload that we analyzed, the results show that most of the operations are read operations. Therefore, optimizing the storage system handling this kind of a workload for read operations can improve performance. This can be done, for example, by converting part, if not all, of the write cache to read cache. In the same workload, we further observed that the distribution of logical seek distances is symmetric. Clearly, knowing why the logical seek distances are symmetric would be useful to optimize the storage system to handle this behavior efficiently.

Our results also show that I/O request sizes are operating system dependent. They depend almost entirely on the particulars of the file system used by the operating system, rather than the requirements of the application.

Concerning modelling storage system workloads, among other things, our results show that the exponential distribution function, which is usually used to model inter-arrival time, is not the best model for this parameter in the three workloads that we analyzed. Instead, we found the Weibull, lognormal and beta probability density functions to be better models. Secondly, our results show that the assumption made in many studies that logical seek distances are always uniformly distributed is incorrect. Our results show that the probability of having a logical seek distance falling in some intervals of values is much higher than other intervals in the workloads that we analyzed. Therefore, using probability mass functions, as we have done in this study, to model logical seek distance is more appropriate.

7.2 Future work

Although ESSWA, in its present state, can be used to perform storage system workload analysis and produce useful results, more analysis functions and features can be added to it as enhancements. For example, it can be enhanced so that it can be able to:

- analyze storage system workloads in terms of other parameters besides those that we considered in this study such as process ID of the process responsible for an I/O request, I/O mode (i.e., either synchronous or asynchronous), I/O request sequence number, file objector pointer, etc.

- analyze two or more workloads at the same time, e.g., to determine the correlation between two workloads' I/O events. This is necessary when deciding to pool storage system resources to increase throughput as explained in section 2.3.2(G).

- perform more trace manipulations, e.g., given trace data collected over a period of time, say T seconds, filtering them to retain data for a specific duration of time, say τ seconds, where $0 < \tau < T$.
- perform staged analysis. For example, if the system is able to analyze two or more workloads at the same time, then it should be able to compute the average I/O request size for each workload, store the averages and then compute the average of these averages.

Appendix A

R Code for ESSWA Back-end

```
library(RDCOMServer)
library(evd) # For fitdistr()

###########main_function######################

AnalysisServer <- function(dataSizes=c()) {


############Function to convert array of strings to array of
##decimals ######################
ConvertFunction<-function(DataArray) { Results <-
type.convert(DataArray, na.strings = "NA", as.is = FALSE, dec =
".") return(Results); }


############Function to calculate Inter-arrival times#####
CalculateIAT<-function(TS) { #multiplying by 1000000 to change
seconds to microseconds
  len1 = length(TS)
  len2 = len1 - 1

  IAT <- 1: len1
  for(i in 1: len2 )
  {
   IAT[i] <- (TS[i+1]-TS[i])*1000000;

  }
  IAT[len1] = IAT[len2]
  return(IAT)
}




############Function to calculate seek distances#####
CalculateSD<-function(SA) {
  len1 = length(SA)
  len2 = len1 - 1

  SD <- 1: len2
  for(i in 1: len2 )
  {
   SD[i] <- SA[i+1]-SA[i];
  }
 SD[len1] = SD[len2]

return(SD) }


############Function to convert string opcode to integer
##opcode#####
ConvertOpcode<-function(RW,ReadString, WriteString)
{
  len1 = length(RW)
  RWOpNum <- 1: len1
  for(i in 1: len1 )
  {
   #if ( (toupper(RW[i])=="READ") || (toupper(RW[i])=="R")  )
   if (  toupper(RW[i])==toupper(ReadString)   )

       RWOpNum[i] = 1
   #else if (  (toupper(RW[i])=="WRITE") || (toupper(RW[i])=="W") )
   else if ( toupper(RW[i])==toupper(WriteString)  )

       RWOpNum[i] = 2
   else
       RWOpNum[i] = 3

  }
return(RWOpNum) }




#########Read trace file#################
readTraceFile <- function(fileName, args, COMseparator,
ReadString, WriteString) { library(tools) separator1 =
COMseparator separator2 = ’’

par1 = 0

a <-c(0)
b <-c("")
a[1] <- -1
a[2] <- -1
b[1] <-"0"
b[2] <-"0"
LV=a;
SD=a;
DS=a;
RWOpNum=a;
IAT = a;
TS = a;
SA = a;

g<-c()
g[1]<-COMseparator
g[2]<-COMseparator
```

```
IOTraffic<-list(LV=a,SA=a,DS=a,RWOpNum=b,TS=a)

options(show.error.messages = FALSE)
try(IOTraffic<-read.table(file_path_as_absolute(fileName),sep=separator1,
fill = TRUE, colClasses =
c("character","character","character","character", "character",
"character", "character", "character", "character",
"character")),silent = FALSE) options(show.error.messages = TRUE)

if (( IOTraffic[[2]][1] != "") && ( IOTraffic[[3]][1] != "") ) {
LV[1]=2 LV[2]=2

IOTraffic4 <- IOTraffic

write.table(IOTraffic1, file = "trace3.txt", append = FALSE, quote
= FALSE, sep = "  ", eol = "\n", na = "NA", dec = ".",col.names =
FALSE, row.names = FALSE)

options(show.error.messages = FALSE)
try(IOTraffic4<-read.table("trace3.txt",sep=separator2, fill =
TRUE, colClasses = c("character","character","character",
"character", "character", "character", "character", "character",
"character", "character")),silent = FALSE)
options(show.error.messages = TRUE)

if (separator1==',' ) { write.table(IOTraffic4, file =
"trace4.txt", append = FALSE, quote = FALSE, sep = "  ", eol =
"\n", na = "NA", dec = ".",col.names = FALSE, row.names = FALSE)

options(show.error.messages = FALSE)
try(IOTraffic<-read.table("trace4.txt",sep=':', fill = TRUE,
colClasses = c("character","character","character","character",
"character", "character", "character", "character", "character",
"character")),silent = FALSE)
options(show.error.messages = TRUE)

IOTraffic1<-list()
for(i in 1: 10 )
 {
   IOTraffic1[[i]] <- IOTraffic[[i]];
  IOTraffic1 <- IOTraffic;
 }

write.table(IOTraffic1, file = "trace3.txt", append = FALSE, quote
= FALSE, sep = "  ", eol = "\n", na = "NA", dec = ".",col.names =
FALSE, row.names = FALSE)

IOTraffic4<-list() options(show.error.messages = FALSE)
try(IOTraffic4<-read.table("trace3.txt",sep=separator2, fill =
TRUE, colClasses = c("character","character","character",
"character", "character", "character", "character", "character",
"character", "character")),silent = FALSE)
options(show.error.messages = TRUE)
}

args <-c()
args[1]<-0
args[2]<-0
args[3]<-3
args[4]<-5
args[5]<-1

if (args[1] == 1)
  LV <- IOTraffic4[[1]]
else if (args[1] == 2)
  LV <- IOTraffic4[[2]]
else if  (args[1] == 3)
  LV <- IOTraffic4[[3]]
else if  (args[1] == 4)
  LV <- IOTraffic4[[4]]
else if ( args[1] == 5)
  LV <- IOTraffic4[[5]]
else if  (args[1] == 6)
  LV <- IOTraffic4[[6]]
```

```
else if  (args[1] == 7 )
  LV <- IOTraffic4[[7]]
else if ( args[1] == 8)
  LV <- IOTraffic4[[8]]
else if  (args[1] == 9)
  LV <- IOTraffic4[[9]]
else if ( args[1] == 10)
  LV <- IOTraffic4[[10]]
else
  LV <- 0


if (args[2] == 1)
  SA <- IOTraffic4[[1]]
else if (args[2] == 2)
  SA <- IOTraffic4[[2]]
else if  (args[2] == 3)
  SA <- IOTraffic4[[3]]
else if  (args[2] == 4)
  SA <- IOTraffic4[[4]]
else if  (args[2] == 5)
  SA <- IOTraffic4[[5]]
else if  (args[2] == 6)
  SA <- IOTraffic4[[6]]
else if  (args[2] == 7 )
  SA <- IOTraffic4[[7]]
else if  (args[2] == 8 )
  SA <- IOTraffic4[[8]]
else if  (args[2] == 9 )
  SA <- IOTraffic4[[9]]
else if  (args[2] == 10 )
  SA <- IOTraffic4[[10]]
else
  SA <- 0

if (args[3] == 1 )
  DS <- IOTraffic4[[1]]
```

```
else if (args[3] == 2 )
  DS <- IOTraffic4[[2]]
else if  (args[3] == 3 )
  DS <- IOTraffic4[[3]]
else if  (args[3] == 4 )
  DS <- IOTraffic4[[4]]
else if  (args[3] == 5 )
  DS <- IOTraffic4[[5]]
else if  (args[3] == 6 )
  DS <- IOTraffic4[[6]]
else if  (args[3] == 7 )
  DS <- IOTraffic4[[7]]
else if  (args[3] == 8 )
  DS <- IOTraffic4[[8]]
else if  (args[3] == 9 )
  DS <- IOTraffic4[[9]]
else if  (args[3] == 10 )
  DS <- IOTraffic4[[10]]
else
  DS <- 0

if (args[4] == 1 )
  RW <- IOTraffic4[[1]]
else if (args[4] == 2 )
  RW <- IOTraffic4[[2]]
else if  (args[4] == 3 )
  RW <- IOTraffic4[[3]]
else if  (args[4] == 4 )
  RW <- IOTraffic4[[4]]
else if  (args[4] == 5 )
  RW <- IOTraffic4[[5]]
else if  (args[4] == 6 )
  RW <- IOTraffic4[[6]]
else if  (args[4] == 7 )
  RW <- IOTraffic4[[7]]
else if  (args[4] == 8)
  RW <- IOTraffic4[[8]]
```

```
else if  (args[4] == 9 )
  RW <- IOTraffic4[[9]]
else if  (args[4] == 10)
  RW <- IOTraffic4[[10]]
else
  RW <- 0

if (args[5] == 1 )
  TS <- IOTraffic4[[1]]
else if (args[5]== 2 )
  TS <- IOTraffic4[[2]]
else if  (args[5] == 3 )
  TS <- IOTraffic4[[3]]
else if  (args[5] == 4 )
  TS <- IOTraffic4[[4]]
else if  (args[5] == 5 )
  TS <- IOTraffic4[[5]]
else if ( args[5] == 6 )
  TS <- IOTraffic4[[6]]
else if  (args[5] == 7 )
  TS <- IOTraffic4[[7]]
else if  (args[5] == 8 )
  TS <- IOTraffic4[[8]]
else if  (args[5] == 9 )
  TS <- IOTraffic4[[9]]
else if ( args[5] == 10 )
  TS <- IOTraffic4[[10]]
else
  TS <- 0


##########remove rows except read and write operation
if ( (length(RW) > 1) && (par1==1)  ) {
 k=1 RW1<-c() LV1<-c()
DS1<-c() TS1<-c() SA1<-c()

for(i in 1: length(RW) ) {
```

```
if (  ( toupper(RW[i])=="W" )  || (
toupper(RW[i])=="WRITE" ) || ( toupper(RW[i])=="READ" )  || (
toupper(RW[i])=="R" ) ) if ( (
toupper(RW[i])==toupper(WriteString) ) || (
toupper(RW[i])==toupper(ReadString) )    )
 {
  RW1[k] = RW[i] ;

  if (length(LV) > 1)
  {
  LV1[k] = LV[i] ;
  }
  if (length(SA) > 1)
  {
  SA1[k] = SA[i] ;
  }

  if (length(DS) > 1)
  {
  DS1[k] = DS[i] ;
  }
  if (length(TS) > 1)
  {
  TS1[k] = TS[i] ;
  }

  k = k+1;
 }
}

RW<-c() for(i in 1: k-1 )
  {
   RW[i] <- RW1[i];
 }

if (length(LV) > 1)
  {
```

```
  LV<-c()
  for(i in 1: k-1 )
  {
   LV[i] <- LV1[i];
   }
}

if (length(SA) > 1)
  {
  SA<-c()
  for(i in 1: k-1 )
  {
   SA[i] <- SA1[i];
   }
}

if (length(DS) > 1)
  {
  DS<-c()
  for(i in 1: k-1 )
  {
   DS[i] <- DS1[i];
   }
}

if (length(TS) > 1)
  {
  TS<-c()
  for(i in 1: k-1 )
  {
   TS[i] <- TS1[i];
   }
}

} #end if length(RW) > 1
```

```
####convert character LV to numeric.
if (length(LV) > 1) {
options(show.error.messages = FALSE) try(LV<-ConvertFunction(LV),
silent = FALSE) options(show.error.messages = TRUE) #LV<-
type.convert(LV, na.strings = "NA", as.is = FALSE, dec = ".") }

####convert character TS to numeric.
if (length(TS) > 1)
{TS<-type.convert(TS, na.strings = "NA", as.is = FALSE, dec = ".")
options(show.error.messages = FALSE) try(TS<-ConvertFunction(TS),
silent = FALSE) options(show.error.messages = TRUE) }

#####convert character SA to numeric.
if (length(SA) > 1) {
SA<-type.convert(SA, na.strings = "NA", as.is = FALSE, dec = ".")
options(show.error.messages = FALSE) try(SA<-ConvertFunction(SA),
silent = FALSE) options(show.error.messages = TRUE)

}

######convert character DS to numeric.
if (length(DS) > 1) { DS<-
type.convert(DS, na.strings = "NA", as.is = TRUE, dec = ".")
options(show.error.messages = FALSE) try(DS<-ConvertFunction(DS),
silent = FALSE) options(show.error.messages = TRUE)

}

####calculate IAT
if (length(TS) > 1) {
options(show.error.messages = FALSE) try(IAT <- CalculateIAT(TS),
silent = FALSE) options(show.error.messages = TRUE) }

#####calculate SD.
if (length(SA) > 1) {
options(show.error.messages = FALSE) try(SD <- CalculateSD(SA),
silent = FALSE) options(show.error.messages = TRUE) }
```

```
######Convert character opcode to number opcode
if (length(RW) >
1) { options(show.error.messages = FALSE) try(RWOpNum <-
ConvertOpcode(RW, ReadString, WriteString), silent = FALSE)
options(show.error.messages = TRUE) }

} #end if (( IOTraffic[[2]][1] != "") && ( IOTraffic[[3]][1] !=
#"") )

return(list(LV,SD,DS,RWOpNum,IAT, TS, SA, g))
}#end of read



###Function to filter SD, PD, IAT in the Traces ###
FilterFunction <- function(DataArray1, DataArray2, DataArray3,
LVno){ k=1 DataArray11<-c() DataArray22<-c() DataArray33<-c()

for(i in 1: length(DataArray1) ) { if ( DataArray1[i]==LVno ) #if
( DataArray1[i]==1 )

 {
  DataArray22[k] = DataArray2[i]
  DataArray33[k] = DataArray3[i]
  k = k+1;
 }
}

 DataArray2<-c()
 DataArray3<-c()

 DataArray2 <- DataArray22
 DataArray3 <- DataArray33

 IAT<-CalculateIAT(DataArray3)
 listPD <- ParallelDegrees(DataArray3)
 PD <- listPD[[1]]
```

```
  len2 <-length(DataArray2)
  SD <- 1: len2
  for(i in 1: len2-1 )
  {
   SD[i] <- DataArray2[i+1]-DataArray2[i];
  }
  SD[len2] =DataArray2[len2-1]

  return(list(SD,IAT, PD))
}


###Function to filter LV, DS, OP in the Traces ###
GenericFilterLVFunction<- function(DataArray1, DataArray2,
DataArray3, LVno){ k=1 DataArray11<-c() DataArray22<-c()
DataArray33<-c() DataArray4<-c()

for(i in 1: length(DataArray1) ) { if ( DataArray1[i]==LVno )
 {
  DataArray22[k] = DataArray2[i] ;
  DataArray33[k] = DataArray3[i] ;
  DataArray4[k] = DataArray1[i] ;
  k = k+1;
 }
}

  DataArray2<-c()
  DataArray2 <- DataArray22;

  DataArray3<-c()
  DataArray3 <- DataArray33;

  return(list(DataArray2,DataArray3,DataArray4))
}
```

```
########## function to generate frequency table with a threshold
#frequency #################
 DataFrequency <- function(DataArray, minPercent){
 DataArrayf<-factor(DataArray)
 DataArrayfr<-table(DataArrayf)
 DataArraylevels <-levels(DataArrayf)
 len <-length(DataArray)

#initialize variable for intervals, frequencies and counter
  GroupValue<-c(0)
  FrequenceValue <- c(0)
  GroupValue[1]<-0
  FrequenceValue[1] <- 0
  GroupValue[2]<-0
  FrequenceValue[2] <- 0
  j=0

 for(i in 1:length(DataArraylevels) )
 {
   if ( ((DataArrayfr[[i]]/len)*100) >= minPercent[1])
   #if ( (DataArrayfr[[i]]/len) > 10)
    {
     j=j+1
     GroupValue[j] = DataArraylevels[i]
     FrequenceValue[j] = DataArrayfr[[i]]
    }
 }
return(list(GroupValue,FrequenceValue)) }

########## function to generate frequency table #################
usageFrequency <- function(DataArray){
 DataArrayf<-factor(DataArray)
 DataArrayfr<-table(DataArrayf)
 DataArraylevels <-levels(DataArrayf)
 GroupValue<-c(0)
 FrequenceValue <- c(0)
 for(i in 1:length(DataArraylevels) )
 {
     GroupValue[i] = DataArraylevels[i]
     FrequenceValue[i] = DataArrayfr[[i]]
 }
return(list(GroupValue,FrequenceValue)) }

########## function to generate bins #################
HistCells<-function(DataArray){
 HistAttributes <-hist(DataArray, breaks = "scott", plot=FALSE)
 HistAttributes <-hist(DataArray, plot=FALSE)
 CellBreaks<-HistAttributes$breaks
 CellCounts<-HistAttributes$counts
 return(HistAttributes)
 }


########## function to calculate Parallellism Degrees#########
ParallelDegrees <- function(DataArray){
 DataArrayf<-factor(DataArray)
 FrequenceValue <- c(0)
 DataArray111<-1:length(DataArray)

 for (i in 1:length(DataArray) )
 {
     DataArray111[i] = 1
 }
 FrequenceValue <- tapply(DataArray111,DataArrayf, sum)
 return(list(FrequenceValue))
 }


##########function to calculate the AutoCorrelation function#####
AutoCorrelation <- function(DataArray, lag){
 acfVAR<-acf(DataArray, lag.max = lag, type = c("correlation"),
 plot = FALSE)
 tempValue<-acfVAR$acf[lag+1,1,1]
 return(tempValue)
 }


#####function to calculate the coefficient of
#cross-correlation#####

CrossCorrelation <- function(DataArray1, DataArray2){
 corValue<-cor(DataArray1, DataArray2) if (is.na(corValue)) {
 corValue = 100 }
 return(corValue)
 }


##########function to calculate the tail index#####
heavyTailedness<-function(dataArray) {
  for(i in 1: length(dataArray) )
  {
   if  (dataArray[i]==0 )
       dataArray[i] <- 1}
 library(aws)
 tIndex<-awstindex(dataArray)
 return(tIndex$tindex)
 }


######function to determine outlier limits for a data set#####
Outliers <- function(DataArray){
  dataSummary <- summary(DataArray)
  dataSummary[[2]]->lowerQuartile;
  dataSummary[[5]]->UpperQuartile;
  LowerOutlier <- dataSummary[[3]]-6*(UpperQuartile - dataSummary[[3]])
  UpperOutlier <-dataSummary[[3]]+6*(UpperQuartile - dataSummary[[3]])
  ExtremeValues <- 1: 2
  ExtremeValues[1] <-LowerOutlier
  ExtremeValues[2] <-UpperOutlier
  return(ExtremeValues)
```

```
}


######function to calculate minimum, lower quartile, median,
#mean, upper quartile and maximum###########################
FiveNumbers <- function(DataArray){
  dataSummary <- 1: 6
  dataSS <- summary(DataArray)
  for(i in 1: 6 )
  {
   dataSummary[i] = dataSS[[i]]
  }
  return (dataSummary)
}


######function to calculate  Variance, Standard Deviation,
#Coefficient of variation, Coefficient of skew and Coefficient of
#kurtosis.#######################################################
OtherStatistics <- function(DataArray){
  dataS <- c(0)
  dataS[1]<- var(DataArray)
  dataS[2]<- sd(DataArray)
  dataMean <-mean(DataArray, na.rm = TRUE);
  dataStd <- sd(DataArray, na.rm = TRUE);

  #calculate coefficient of variation, CV
  CV <- dataStd / dataMean
  dataS[3]<- CV

  #calculate coefficient of skew
  ksum = c(0);
  ksum <-1:length(DataArray);
  for (i in 1:length(DataArray) )
     {
      ksum[i] =  ( ( DataArray[i]-dataMean )/dataStd )^3  ;
     }
```

```
tempSum <- sum(ksum)
dataSkew <-(     (  length(DataArray)     )
               /  (   (length(DataArray)-1) * (length(DataArray)-2))
              * tempSum
          )
dataS[4]<- dataSkew

##calculate coefficient of kurtosis
ksum = c(0);
ksum <-1:length(DataArray);
for (i in 1:length(DataArray) )
   {
    ksum[i] = (( DataArray[i]-dataMean )/dataStd )^4;
   }
tempSum <- sum(ksum)
dataKurtosis <-(     (  length(DataArray) * ( length(DataArray)+1 )     )
               /  (   (length(DataArray)-1) * (length(DataArray)-2)
               * (length(DataArray)-3)     )
              * tempSum
            ) -
           (  3 * (length(DataArray)-1)^2     )/(   (length(DataArray)-2)*
           (length(DataArray)-3)     )
dataS[5]<- dataKurtosis
if (is.na(dataS[1] ))
{
dataS[1] = 0
}
if (is.na(dataS[2] ))
{
dataS[2] = 0
}
if (is.na(dataS[3] ))
{
 dataS[3] = 0
}
if (is.na(dataS[4] ))
{
  dataS[4] = 0
  }
  if (is.na(dataS[5] ))
  {
  dataS[5] = 0
  }
  return(dataS)
  }


######This function calculates the lambda discrepancy statistic.
#It basically calls the function developed by Lourens Walters and
#handles erroneous behavior ##########################

lambdaSquared <- function(observed, distribution) {
 parameterlist<-list()
 errorcode = -1
 if(distribution=="lognormal"){ parameterlist = list(zeta=1000,
   sigma=1000) }
 if (distribution=="normal"){ parameterlist =
   list(mean=1000, sd=1000) }
 if (distribution=="exponential"){
   parameterlist = list(beta=1000) }
 if (distribution=="gamma"){ parameterlist = list(shape=1000,
   rate=1000) }
 if (distribution=="beta"){ parameterlist = list(shape1=1000,
   shape1=1000) }
 if (distribution=="pareto"){ parameterlist =
   list(alpha=1000, beta=1000) }
 if (distribution=="weibull"){
    parameterlist = list(gamma=1000, alpha=1000) }
 lambda=1000
 fitResults<-list(lambda=lambda,parameters=parameterlist,
 errorcode=errorcode)

options(show.error.messages = FALSE)
try(fitResults<-lambdaSquared1(observed, distribution), silent =
FALSE) options(show.error.messages = TRUE) return(fitResults) }


#####definition of the DCOM interface ##########################
 list(  lambdaSquared = lambdaSquared,
   lambdaSquared = lambdaSquared,
   readTraceFile = readTraceFile,
   beta.distr = beta.distr,
   beta.density = beta.density,
   beta.variates = beta.variates,
   pareto.distr = pareto.distr,
   pareto.quantile = pareto.quantile,
   pareto.density = pareto.density,
   pareto.variates = pareto.variates,
   bin.actual = bin.actual,
   bin.expected = bin.expected,
   combine.bins = combine.bins,
   AutoCorrelation = AutoCorrelation,
   CrossCorrelation = CrossCorrelation,
   FiveNumbers = FiveNumbers,
   Outliers =Outliers,
   OtherStatistics =OtherStatistics,
   DataFrequency=DataFrequency,
   usageFrequency=usageFrequency,
   heavyTailedness=heavyTailedness,
   ParallelDegrees =ParallelDegrees,
   HistCells=HistCells,

   .properties = c("dataSizes", "sigma"),
   .help = c(lambdaSquared = "generate a sample of values",
     beta.distr  = "CDF values from this distribution",
     beta.density = "quantile values from this distribution",
     beta.variates = "values of the density function for
     this distribution"
      ))

} #end AnalysisServer
```

```
##########Register the Main Function as DCOM Object #######
def=SCOMIDispatch(AnalysisServer, "ESSWAServer") def@classId =
getuuid("c484d2f9-21f5-49ac-8c8d-2007e12245d7")
registerCOMClass(def)
```

Bibliography

[Gang95] G. R. Ganger, "Generating representative synthetic workload: An unsolved problem," *In Int. CMG Conference*, pp. 4-8, Nashville, TN, USA, 1995. Computer Measurement Group.

[Hsco03] W. Hsu *et al*, "Characterization of I/O traffic in personal and server workloads," *IBM Systems Journal*, vol 42, no. 2, 2003, pp. 347- 372.

[Hsco01] W. Hsu *et al*, "Characterization of Production Database Workloads and TPC Benchmarks," *IBM Systems Journal*, vol 40, no. 3, 2001, pp. 781-802.

[Whsu01] W. Hsu *et al*, "I/O Reference Behaviour of Production Database Workloads and TPC Benchmarks-An analysis at the Logical Level," *ACM Transactions on Database Systems*, vol 26, 2001, pp. 96-143.

[Drac00] D. Roselli *et al*, "A comparison of File System Workloads," *In Proceedings of USENIX Annual Technical Conference*, San Diego, CA, 2000, USENIX Association, Berkeley, CA (2000), pp. 93-109.

[Vktr03] A. Veitch *et al*, "The Rubicon workload characterization tool," *SSP technical report HPL-SSP-2003-13*, HP Laboratories, March 2003.

[Zksr96] W. Courtright II *et al*, "RAIDframe: rapid prototyping for disk arrays," *In Proceedings of the 1996 Conference on Measurement and Modelling of Computer Systems (SIGMETRICS)*, vol 24, 1996, pp. 268-269.

[Zksr90] P. Chen *et al*, "Maximizing performance in a striped disk array," *In Proceedings of ACM SIGARCH Conference*, 1990, pages 322-331, ACM.

[Krao93] K. Ramakrishman *et al*, "Trace Driven Analysis of Write Caching Policies for Disks," *In Proceedings of ACM Conference on Measurement and Modelling of Computer Systems (SIGMETRICS)*, Santa Clara, CA, 1993, ACM, New York(1993), pp. 13-23.

[Vktr93] C. Ruemmler *et al*, "UNIX Disk Access Patterns," *In Proceedings of USENIX Winter Conference*, San Diego, CA, 1993, USENIX Association, Berkeley, CA (1993), pp. 405-420.

[Ikpo95] J. Heath *et al*, "Analysis of Disk Workloads in Network File Server Environments," *In Proceeding of Computer Measurement Group (CMG) Conference*, Nashville, TN, 1995, Computer Measurement Group (1995), pp. 313-322.

[Kppo98] S. Gribble *et al*, "Self-Similarity in File Systems," *In Proceedings of ACM Conference on Measurement and Modelling of Computer Systems (SIGMETRICS)*, Madison, WI, 1998, ACM, New York (1998), pp. 141-150.

[Hhpd96] G. Hansen *et al*, "Physical Database Systems," Database Management and Design, Prentice-Hall India, 2nd ed, 1996, pp. 323-346.

[Assa02] A. Silberschatz *et al*, "Storage and File Structures," Database System Concepts, McGraw Hill, 4th ed. 2002, pp. 393-414.

[Kkpr92] K. Ramakrishnan *et al*, "Analysis of File Traces in Commercial Environments," *Performance Evaluation Review*, vol. 20, No. 1, June 1992.

[Douc99] J. Douceur *et al*, "A Large-Scale Study of File-System Contents," *Proceedings of the 1999 Sigmetrics Conference* June 1999, pp. 59-70.

[Sien94] T. Sienknecht *et al*, "The Implications of Distributed Data in Commercial environment on Design of Hierarchical Storage Management," *Performance Evaluation*, 20, May 1994, pp. 3-25.

[Rose92] M. Rosenblum *et al*, "The Design and Implementation of a Log-Structured File System for UNIX," *ACM Trabsactions on Computer Systems*, 10(1), pp. 26-52, February 1992.

[Bake91] M. Baker *et al*, "Measurements of a Distributed File System," *Proceedings of the Thirteenth Symposium on Operating Systems Principles*, pp. 198-212, December 1991.

[Whsu03] W. Hsu, "Dynamic Locality Improvement Techiques for Increasing Effective Storage Performance, Ph.D. thesis, University of California, Berkeley, CA (December 2002). Available as Technical Report CSD-03-1223, Computer Science Division University of California, Berkeley(January 2003).

[Dpjh98] D. Patterson *et al*, "I/O Devices - Computer Organization and Design," Morgan Kaufmann Publishers Inc, 2nd ed. 1998, pp 644-648.

[Knhs86] Kronenberg *et al*, "VAXclusters: A Closely-coupled Distributed System," *ACM Transactions on Computer Systems*, Vol. 4, No. 2, May 1986.

[Oust85] J. Ousterhout *et al*, "A Trace-Driven Analysis of UNIX 4.2 BSD File System," *Proceedings of the Tenth Symposium on Operating Systems Principles*, pp. 15-24, December 1985.

[Bozm91] G. Bozman *et al*, "A Trace-Driven Study of CMS File References," *IBM Journal of Research and Development*, 3595-6, pp. 815-828, September-November 1991.

[Zhou99] M. Zhou *et al*, "Analysis of Personal Computer Workloads," *Proceedings of the Seventh International symposium on Modeling, Analysis and Simulation of Computer and Telecommunication Systems (MASCOTS)*, pp. 208-217, October 1999.

[Zhou93] B. Pasquale *et al*, "A Static Analysis of I/O Characteristics of Scientific Applications in a Production Workload", Conference on High Performance Networking and Computing archive, *Proceedings of the 1993 ACM/IEEE conference on Supercomputing*, pp. 388-397.

[Phsz04] P. Harrison *et al*, "Calibration of a Queuing Model of RAID Systems," *In Proceedings of Practical Application of Stochastic Models*, Imperial College, London, September 2004.

[Wwha04] W. Hsu *et al*, "The Performance Impact of I/O Optimization on Disk Improvements," *IBM Journal of Research and Development*, 48(2):255-289, March 2004.

[Dwsco92] D. Scott, "Multivariate Density Estimation," John Wiley and Sons, Inc, 1992.

[Sppm90] S. Pederson *et al*, "Estimating Model Discrepancy," *TECHNOMETRICS*, vol 32, pp. 305-314, 1990.

[Mast86] M. Stephens, Tests Based on EDF Statistics, STATISTICS: textbooks and monographs, no 68, pp. 97-185, 1986.

[Dwsc92] D. Scott, "On Optimal and Data-Based Histograms," *Biometrica*, no. 66 pp. 605-610, 1979.

[Stfs92] "SPC TRACE FILE FORMAT SPECIFICATION," Revision 1.0.1. Storage Performance Council (SPC), 1060 El Camino Real, Suite "F" Redwood City, CA 94062-1645, © 1999, 2002 Storage Performance Council.

[Dwsc92] Les Underhill *et al*, "Data Survey", IntroSTAT, 5th ed., Juta and Co Ltd, pp. 1-44, 1994.

[Grim95] K. Grimsrud, "Rank disk performance analysis tool", *Intel Corporation White Paper*, available from http://developer.intel.com/design/ipeak/stortool.

[IBM99] IBM DFSSMS/MVS Optimizer. Product information available from http://www.storage.ibm.com/software/ opt/optprod.htm.

[Toua91] H. Touati *et al*, "Reducing and Manipulating Complex Trace Data", *Software Practice and Experience*, 21(6), pp. 639-655, June 1991.

[Sack03] D. J. Sacks, "Disks system and internal bandwidth wars", *Technonlogy Insights White Paper from IBM*, March 2003.

[Lowa04] L. O. Walters, "A Web Browsing Workload Model For Simulation", a dissertation submitted to the department of computer science, faculty of science at the University of Cape Town in fulfillment of the requirements for the degree of master of science, May 2004.

[Tjst03] T. J. Stewart, "Topics in Probability Theory" notes, Department of Statistical Sciences, University of Cape Town, Dec 2003.

[Fdal95] F. Daly *et al*, "Elements of Statistics", Addison-Wesley Publishing Company, pp. 1-41, 1995.

[Reki99] R. Kirk, "Statistics: An Introduction", Baylor University, 4th ed., 1999.

[Pgsk05] P. G. Sikalinda *et al*, "Analyzing Storage System Workloads", To appear *In Proceedings of South African Telecommunication Networks and Applications Conference*, South Africa, September 2005.

[Wnve04] W. N. Venables *et al*, "An Introduction to R", *Notes on R: A programming Environment for Data Analysis and Graphics*, Version 2.0.1 (2004-11-15).

[Rdct04] R development Core Team, "Writing R Extensions", Version 2.0.1 (2004-11-15).

[Fewa04] F. Wang *et al*, "File System Workload Analysis For Large Scale Scientific Applications", *Proceedings of the 21st IEEE / 12th NASA Goddard Conference on Mass Storage Systems and Technologies*, April 2004, College Park, Maryland USA.