Statistical Characterization of Storage System Workloads for Data Deduplication and Load Placement in Heterogeneous Storage Environments

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

Nohhyun Park

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

David J. Lilja

November, 2013

Acknowledgements

I like to thank my advisor, David J. Lilja, who has suffered through more than a handful insensible ideas and lack of experience. Also members of ARCTiC lab, Biplob Debnath, Shruti Patil, Eric Seppanen, Weijun Xiao, Peng Li and William Tuohy for their friendship and insights.

My thanks also goes to student members of CRIS, Weiping He, Anna Kryzhnyaya, Muthukumar Murugan, Joe Naps, Dongchul Park and Guanlin Lu for interesting projects and fun trips.

I would also like to thank my committee members, David Du, Ulya Karpuzcu and John Sartori for their valuable comments and guidance.

I would like to thank Jerry Fredin of NetApp for his support during the data deduplication project. I also like to thank Philip Shilain and Fred Douglis of EMC for their feedbacks on numerous projects. I would like to thank Dave B. Anderson and Luke W. Friendshuh for their support during the IO workload characterization project. I would also like to thank Irfan Ahmed and DRS team at VMware for their support during the Romano project.

I would like to thank DARPA, NSF and CRIS member companies for funding my projects as well as Minnesota Supercomputing Institute for providing the valuable compute cycles.

I would also like to mention Sungwook and Sangyeon Jeong whom I promised to include in my acknowledgement in exchange for occasional free lunch at their place.

I could not have survived long years of graduate school without the support of my parents whose interest in technology and willingness to invest in my education allowed me to find and pursue my passion.

Although mostly distraction then help, I would also like to thank my brother, Nohguen, for the occasional fun times we were able to have in last 6 years.

My special thanks goes to Hyewon and Ciana.

Dedication

To my parents.

Abstract

The underlying technologies for storing digital bits have become more diverse in last decade. There is no fundamental differences in their functionality yet their behaviors can be quite different and no single management technique seems to fit them all. The differences can be categorized based on the metric of interest such as the performance profile, the reliability profile and the power profile.

These profiles are a function of the system and the workload assuming that the systems are exposed only to a pre-specified environment. Near infinite workload space makes it infeasible to obtain the complete profiles for any storage systems unless the system enforces a discrete and finite profile internally. The thesis of this work is that an acceptable approximation of the profiles may be achieved by proper characterization of the workloads. A set of statistical tools as well as understanding of system behavior were used to evaluate and design such characterizations. The correctness of the characterization cannot be fully proved except by showing that the resulting profile can correctly predict any workload and storage system interactions. While this is not possible, we show that we can provide a reasonable confidence in our characterization by statistical evaluation of results.

The characterizations of this work were applied to compression ratio for backup data deduplication and load balancing of heterogeneous storage systems in a virtualized environments. The validation of our characterization is validated through hundreds of real world test cases as well as reasonable deductions based on our understanding of the storage systems. In both cases, the *goodness* of characterizations were rigorously evaluated using statistical techniques. The findings along the validations were both confirming and contradicting of many previous beliefs.

Contents

List of Tables

List of Figures

Chapter 1

Introduction

1.1 Current State of Practice

Storage system behavior is harder to predict and model than almost any other components in the computer system. This is believed to be due to the statefulness of the system, complexity of the IO stack and heterogeneity in the underlying technology such as magnetic disks and flash memory. However, we believe that the main culprit is the unpredictable workload to which the system is being subjected to. The current storage systems are designed based on a set of specific workloads. It is assumed that these workloads represents the real workload space. Therefore, the optimizations are typically static and designed for a small number of workloads. Furthermore, there is no requirement for real time monitoring of workloads since it is assumed that the entire workload space is already captured. Finally, the performance of two storage systems are compared only within those small set of test vectors and the result is deemed final.

1.2 Objectives

The goal of this dissertation is not to provide one system that fits all workloads. Rather, we would like to identify what kind of systems exists in a pool of heterogeneous storage systems and direct appropriate workloads to systems that are better suited for the workload. More importantly, we would like a sense of what to expect from the system given a workload such that overall progress can be planed.

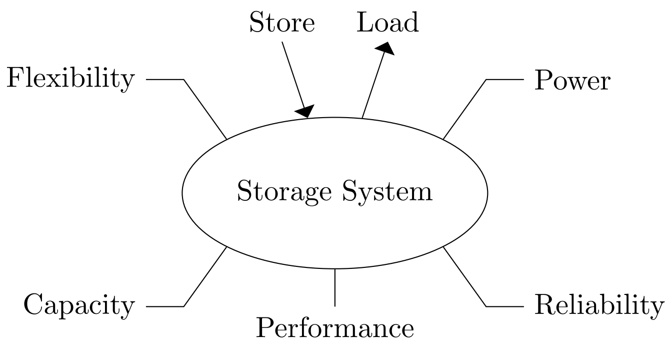


Figure 1.1: Storage system operation and attributes. While there are main two operations, there are many attributes of the system that we are interested in.

The thesis of this dissertation is that to meet these goals we need means to quantify workload characteristics such that the system can measure workload. Such characterization allow the mathematical models of the system responses which in turn allows systems to be designed to dynamically adapt to a wider spectrum of workloads. Additionally, it allows system comparison to be based on their response characteristics to different workloads rather than a few predefined workloads. Finally, we apply the knowledge of the system response characteristics to predict and if possible improve overall system performance.

1.3 Overview

A storage system is a system that supports two operations, *load* and *store*. The *load* operation allows system to change its internal state and the *store* operation allows this state to be verified. A storage system must further ensure that the interval state is immutable in an absence of a *store* operation.

There are numerous attributes of storage systems which are of our interest. The first type of attributes, *flexibility* attribute, describe the allowed state transitions. A generic storage system allows all states to be reached from every other state and will be the focus of this work. The second type of attribute, *capacity* attribute is the number of the states and defines the storage *capacity*. The third type of attribute, *power* attribute, defines power requirement for maintaining the states, state transition and state verification.

The fourth type of attribute, *reliability* defines the probability of state transition in the absence of *store* operation. The last attribute, *performance* attribute, defines speed at which the state transitions occur as well as verification of the states. These classification of attributes as well as the two primary operations are shown in Figure 1.1.

The performance evaluation of storage systems is a difficult task due to lack of representative workloads. Storage benchmark tools test a single aspect such as random read performance at a time. While this approach does allow us to gain useful insights to the system behavior, current practices are largely inadequate due to large benchmark input parameter space. For example, a benchmark with 10 parameters require at least 1024 experiments if conducted exhaustively even if we only test extreme values of each parameter. Furthermore, there is no way to tell if the benchmark being used is enough to test all realistic scenarios. As a result, researchers and developers rely on multiple benchmarks with ad hoc input parameters.

We propose a method to quickly identify input parameters that have high effect on the performance metric of interest. We also show that using multiple benchmarks is unnecessary at times and a good benchmark can cover all operational space by providing a control over key parameters that affect the performance metric being measured.

Workload consolidation is a key technique in reducing costs in virtualized datacenters. When considering storage consolidation, a key problem is the unpredictable performance behavior of consolidated workloads on a given storage system. In practice, this often forces system administrators to grossly over-provision storage to meet application demands. In this paper, we show that existing modeling techniques are inaccurate and ineffective in the face of heterogeneous devices. We introduce *Romano*, a storage performance management system designed to optimize truly heterogeneous virtualized datacenters. At its core, Romano constructs and adapts approximate workload-specific performance models of storage devices automatically, along with prediction intervals. It then applies these models to allow highly efficient IO load balancing.

End-to-end experiments demonstrate that Romano reduces prediction error by 80% on average compared with existing techniques. The result is improved load balancing with lowered variance by 82% and reduced average and maximum latency observed across the storage systems by 52% and 78%, respectively.

The compression and throughput performance of data deduplication system is directly affected by the input dataset. We propose two sets of evaluation metrics, and the means to extract those metrics, for deduplication systems. The First set of metrics represents how the composition of segments changes within the deduplication system over five full backups. This in turn allows more insights into how the compression ratio will change as data accumulate. The second set of metrics represents index table fragmentation caused by duplicate elimination and the arrival rate at the underlying storage system. We show that, while shorter sequences of unique data may be bad for index caching, they provide a more uniform arrival rate which improves the overall throughput. Finally, we compute the metrics derived from the datasets under evaluation and show how the datasets perform with different metrics.Our evaluation shows that backup datasets typically exhibit patterns in how they change over time and that these patterns are quantifiable in terms of how they affect the deduplication process. This quantification allows us to: 1) decide whether deduplication is applicable, 2) provision resources, 3) tune the data deduplication parameters and 4) potentially decide which portion of the dataset is best suited for deduplication.

1.4 Contributions

Some of the findings in the work validates many of the previous techniques. We show that the use of synthetic benchmarks is a valid technique to evaluate the performance is a given storage system under different workloads. We also show that some of the well known characteristics such as randomness and IO size are indeed a key characteristics that affect the performance. However, we also invalidate some of the traditional techniques. Especially, we show that the effects of the workload on different types of storage systems are non-linear and it no longer makes sense to define a storage system performance as a scalar. The result of this finding invalidates the techniques that compares two storage systems based on few performance measurements using synthetic benchmarks.

The followings are the contributions of this dissertation.

We provide a statistically rigorous method to differentiate different workloads in terms of the system performance. Specifically, we show that the different storage benchmark program actually cover similar operational spaces. Furthermore, we show that only few benchmark parameters are able to capture over 50% of system performance variation [2].

We also show that the workload characteristics can be effectively constructed from these parameters as a full factorial combination of these parameters. Additionally we show means to minimize number of terms in the characteristics by using Analysis of Variance(ANOVA). Finally, we show that such characterization is able to model the performance of storage system with high accuracy even when using simple linear regression [3].

We also show how such model can be used to construct a load balancing algorithm for virtualized IO workloads in a heterogeneous storage systems. Algorithm itself is a simple stochastic optimization algorithm known as simulated annealing. However, it is able to decrease the worst case average latency of a virtual disk by 70% and decrease the average latency by 20% [3].

Finally, we show that such technique can be applied to other metrics such as compression ratio and throughput in the backup workload. Here we apply our knowledge of the system to derive characterization metrics and apply simple regression techniques to derive the model. We show that we can predict the compression ratio and write throughput with a high degree of accuracy [4] as well as read throughput [5].

1.5 Organization

This dissertation is organized as follows.

- Chapter 1 provides the overview and main contributions.
- Chapter 2 provides a brief overview of the storage systems that are used as part of the experimental platform.
- Chapter 3 give high level description of the statistical tools used to identify and quantify various components of the workload and to describe their interactions with the system.

- Chapter 4 shows means to efficiently compare different benchmark programs ans suggest how they should be used to evaluate storage system performance.
- Chapter 5 describes a method to model storage system performance as a function of workloads characteristics in a accurate and efficient way.
- Chapter 6 presents Romano, a load balancing system designed on top of storage performance model constructed in Chapter 5
- Chapter 7 shows the workload model can effect both the capacity and performance of backup storage system.
- Chapter 8 lists some of the related works in the area.
- Chapter 9 concludes the thesis work with discussion of some interesting future works.

Chapter 2

Background

2.1 Storage Virtualization

The term *storage virtualization* can be applied very broadly. In fact, any type of IO indirection to provide a virtual block address space can be seen as a type of virtualization.

In this work, we focus on the storage virtualization where a single storage unit provides a multiple volumes through distributed locking mechanism and IO redirection [6, 7, 8].

We have used VMware's VSphere 5.1 as our test platform [9] and will describe how it virtualizes its storage stack in this section.

Figure 2.1 outlines high level view of virtualized storage stack.

For storage controllers, VMware hypervisor provides guest with LSI Logic or Bus Logic driver interface regardless of the underlying hardware. These drivers converts the IO requests to privileged IN and OUT instructions which are trapped by the hypervisor and is handed by the SCSI emulation layer called *vSCSI* layer. This allows every IO to be inspected per VM, per virtual disk basis [10]. This provides an efficient mechanism to measure workload load characteristics at highest resolution.

There are two main functional components required for storage virtualization. The first is the command translation and the second is the address mapping. These are provided by the hypervisor through SCSI virtualization layer.

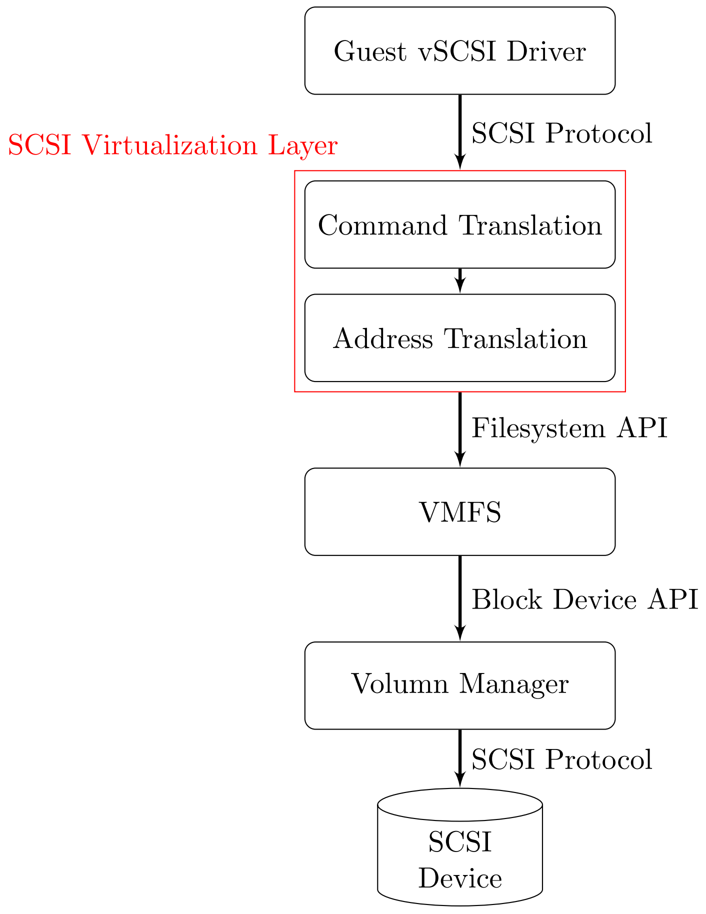


Figure 2.1: Virtualized IO Stack. The raw SCSI command is remapped to a file operation on VMFS via virtualization layer.

For example, READ CAPACITY command should return the last *logical block address*(LBA) of the device or `FFFFFFFFh` [11]. The command translation layer should translate this command to stat command to retrieve the number of blocks in a file.

For READ and WRITE operations, the requesting LBA and the block size needs to be translated into filename, offset and size by the address translation layer.

Once the translation is complete, the resulting filesystem call is passed to VMFS layer which manages metadata consistency through distributed locks. Like any filesystem, VMFS translates the filesystem calls to the underlying block device calls which than is translated into the underlying SCSI command through the volume manager.

The backing storage system can be any device that understands SCSI protocol including FC and SAS.

2.1.1 Storage Live Migration

One of the main benefits of virtualization is decoupling of underlying Hardware from the execution environment. Such decoupling allows the execution environment and all its virtual resources to be migrated while the execution is live.

VM migration across different hosts have been available for a long time [12]. However, it relied on the shared storage since migration of entire storage contents would prove to be too expensive. To hide the expensive migration cost of storage systems, a mirroring scheme is used [13]. The storage system is divided into separate regions and are copied one by one. IF the region that is already copied is written to, the write is mirrored to the remote storage. If the region that is being copied is written to, the write is buffered to be flushed upon completion. This process will continue until only a single region to be copied is left at which time the VM will suspend for the final leg of the copy. The typical down time for 32GB virtual disks was shown to be around 0.2s [13].

The live migration of virtual disks provide mechanism to control the workload-system interactions. We will explore this aspect in detail in Chapter 6.

2.2 Data Deduplication

Data deduplication provides a means to efficiently remove redundancies from large datasets. This process is made possible by first dividing up the data into segments and representing each segment with a much smaller hash value. A redundant segment of data is then easily identified through a hash table lookup. The efficiency of the process comes at the cost of possible data loss through hash collisions but it is understood that the collision probability is negligible compared to the soft error rate of the storage system if an appropriate hash function is used [14, 15, 16, 17]. Another performance factor is the granularity of compression which is limited to the size of duplicate segments. Two segments that are off by a single bit will result in no compression.

Despite these costs, data deduplication has steadily gained its place in backup [18, 19, 15], archive [20] and virtual machine storage solutions [21, 22, 23] due to its potentially

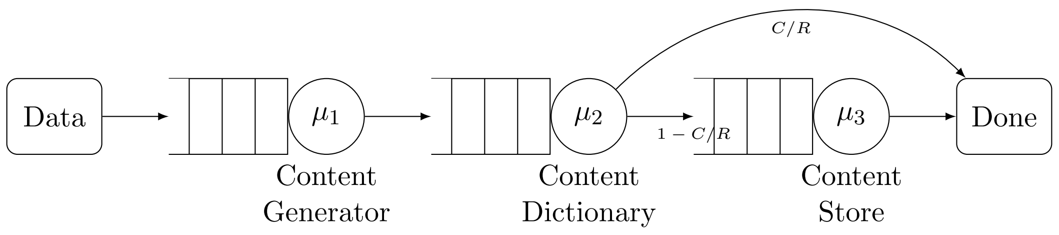


Figure 2.2: Our simplified deduplication system model. It is essentially 3 server open queuing system. At the content dictionary, C/R of the segments exit the system and 1-C/R of the segments are passed on to content store. All data are assumed to be immediately available at the memory and all three servers have exponentially distributed service time.

huge reduction in storage space and IO elimination.

The deduplication process itself can be divided into largely three parts as shown in Figure 2.2. First part of the process generates a sequence of segments from input data stream which we call the *content generator*. In the second phase, the deduplication system generates a dictionary of segments which is in turn used to identify the duplicate segments which we call the *content dictionary*. In the last phase, new segments are stored on a persistent storage system which we call the *content store*. The later two components form a *content addressable storage*(CAS).

2.2.1 Content Generator

The goal of the content generator is to maximize the redundant segments while minimizing the number of segments generated. These two goals often conflict with each other since the amount of duplicate data tends to increase with smaller segments size. This is due to the fact that the generation of segments are typically done in stateless manner to keep up with the huge data size and stringent throughput requirements. Recent papers try to overcome this limitation through a two level definition of contents [24, 16]. However, these approaches come at the extra computation cost and are applicable only when there are less stringent throughput requirements.

These are the three most common methods to define the segment boundaries also known as the *anchors* [15, 17].

- *File based*: A file boundary becomes the segment boundary.
- *Size based*: The anchors are designated every k bits which determines the boundaries for the segment. Therefore, any addition or deletion of data results in shifted anchors which is propagated until the end of the data stream. This effect is known as the *boundary shifting problem* [17].
- *Content based*: An anchor is generated based on the content of the data which becomes the boundaries for the segments. Therefore the anchors are shifted together with the contents in the case of addition and deletion of the data.

The *content based* approach is the most popular method used in data deduplication due to the boundary shifting problem found in the size based method. Average, maximum and/or minimum segment size are usually specified for the content based approach. This allows system to expect segments of the bounded size which makes the segment handling simpler but also ensures that amount of size variation is limited which can cause loss of redundancy [25].

From the system perspective, the content generator is the source generator for the CAS. Typically, any information about the original data stream is lost after this point. This justifies our looking the dataset as a sequence of segments since beyond this point, the system is effectively decoupled from the original dataset.

2.2.2 Content Dictionary

The content dictionary is typically a hash table which is keyed by hash of segments. Hash functions used typically provide one-way property as well as collision resistance. While only the collision resistance is required for correct functionality, one-way property is also attractive to ensure that the malicious users cannot corrupt the system by generating hash collisions.

For a large dataset, the content store itself becomes significantly large. To relax the memory requirements for the deduplication system, the content dictionary is typically stored on the disk [26, 19, 15, 27] . Only a portion is cached onto the memory at a time. Furthermore, any writes to the index table must be serialized which require expensive locks [23]. Therefore, the caching algorithm for the index table is one of the major performance factor within the deduplication system.

To avoid unnecessary accesses to the index table, Bloom filter [28] is used to quickly determine segments that are not in the index table [15]. While the backup stream typically exhibit an inherent spatial locality between the segment instances [15], more intelligent caching schemes that uses data similarities have also been proposed [19, 27].

Regardless of caching scheme, it is clear that content dictionary determines the data path for any given segment. While every segment must read the index table at least once, only the new segments result in it's update. Furthermore, these new segments must be passed down to the content store to be store on the disk. Problem is more complicated for the read where segments maybe fragmented in various places both for the actual contents and the dictionary. A recent work duplicated heavily used segments over the physical disk such that the average seek time can be minimized [29]. However, such approach assumes small working set of segments and is heavily workload dependent.

In this work we assume nothing about the caching policy of the content dictionary or any other auxiliary data structure to minimize the access to the dictionary. We only assume that the different segment types results in different resource requirements in regular read/write operations.

2.2.3 Content Store

Once the segments are identified as new content that needs to be stored, it is passed down to the content store to be processed. Various different mechanisms are possible. The content may already be on the disk and the content store only generates a logical mapping and freeing of segments [23, 30] or use log-based filesystem to optimize for the in-band writes [15].

The content store is not affected specifically by the deduplication system. While inherent *copy-on-write* (COW) support of storage subsystem [31, 32] allows easier sharing of segments, the fundamental operations of underlying storage does not differ from dataset to dataset. Therefore, we focus mostly on the behaviors of the content dictionary in this paper.

Chapter 3

Statistical Tools

3.1 Plackett-Burman designs

The *Plackett-Burman design* (PB design) [33] minimizes the number of experiments required to estimate the effect of independent variables on the dependent variable. Basically, it estimates the effect of single input parameter on an output in such a way that variance of this effect due to other inputs is minimized. The effect is an approximation of the full factorial design but with linear complexity. Required assumption is that higher order interaction effects are negligible. PB design has been widely used to evaluate effect of benchmark input parameters on processor performance [34] and database query processing performance [35], for example.

Formally, the main effect m_1 of input variable x_1 on output y is defined to be

$$m_1 = \frac{[\sum f(x_1', x_2, ..., x_n) - \sum f(\overline{x_1}, x_2, ..., x_n)]}{2^n} \tag{3.1}$$

where x_1' represents some extreme value of x_1 and $\overline{x_1}$ represents the mean of x_1. f is an unknown function which maps all x_i to y. The summation is carried out for all possible combination of x_2 through x_n. Hence, the effect of x_i is simply maximum deviation of y given maximum deviation of x_i.

The goal of PB design is to estimate all m_is such that

$$S = \sum_j (\widehat{y_j} - M - a_{j1}m_1 - a_{j2}m_2 ... - a_{jn}m_n + \alpha)^2 \tag{3.2}$$

is minimized. $\widehat{y_j}$ is simply the sampled value of y at time j and M is mean value of all y_js. a_{ji} takes value ±1 based on whether the x_i had positive or negative effect on $\widehat{y_j}$. α is the higher order effects which are ignored based on the observations made by Fisher [36]. Since there are n values to be estimated, the minimum number of experiments required to derive a unique set of m_i is $n+1$. Plackett and Burman proved that S can indeed be minimized using $\lceil (n+1)/4 \rceil * 4$ experiments assuming $\alpha = 0$ [33].

3.2 Independent Component Analysis

While the PB method quantifies effect of each input parameters to output of interest, it does have one major drawback. Because the effect estimation assumes any higher order effects to be zero, the estimated effects of any main factor is partially aliased with any interactions not including that particular main factor.

In contrast, *Principal Component Analysis* (PCA) and *Independent Component Analysis* (ICA) estimate the effect of uncorrelated components of original factors [37]. These components may not reflect any intuitively meaningful parameter. However, they do provide a uncorrelated set of factors that can be derived from the original ones. The limitation of the PCA is that it requires the original factors to be normally distributed. ICA is a more generic approach in that such assumption is not required [38]. It has been observed that ICA performs better than PCA in capturing processor workload space [39].

In essence, ICA tries to identify statistically independent components from given factors.

Formally, ICA estimates a matrix $\mathbf{W}$ such that given $\mathbf{x} = (x_1, x_2, ..., x_n)$ with joint *probability density function* (pdf) $p_x(x_1, x_2, ..., x_n)$ such that $\mathbf{y} = \mathbf{W} \times \mathbf{x}$ has joint pdf $p_y($$\mathbf{y}$ with following property.

$$E[y_1^{p_1}, y_2^{p_2}, ..., y_m^{p_m}] = E[y_1^{p_1}]E[y_2^{p_2}]...E[y_m^{p_m}] \tag{3.3}$$

for every integer value of p_i and $m \leq n$. Equation 3.3 is the definition of independence.

3.3 Linear Regression

A general model that predicts P given W is:

$$P = f_P(w_1, w_2, w_3, ...) + \epsilon \tag{3.4}$$

where w_i's are set of controlled variables and P is metric of interest. Estimating function f_P without any assumption is computationally expensive if not impossible. Therefore, we restrict the form of f_P to a generic linear model of the form:

$$\begin{gathered} P = \beta_0 + \beta_1 w_1 + \beta_2 w_2 + ... + \epsilon = W\beta + \epsilon \\ W = \begin{bmatrix} 1 & w_1 & w_2 & w_3 & ... \end{bmatrix},\ \beta = \begin{bmatrix} \beta_0 \\ \beta_1 \\ \beta_2 \\ ... \end{bmatrix} \end{gathered} \tag{3.5}$$

One thing to note here is that the linear model only dictates how the β values interact with the controlled variables. The controlled variables themselves can be in the form of any scale. This means that contrary to what many believe, W does not need to have linear relationships with P. The goal is to estimate β such that ϵ is acceptable. Typically, we assume that the effects of any factors not being modeled result in white noise and that the ϵ should be normally distributed.

3.4 Analysis of Variance(ANOVA)

ANOVA is a widely used technique to separate the total variation (*SST*) in a set of measurements into a component due to random fluctuations (*SSE*) and a component due to actual difference among the alternatives (*SSA*) [40].

Intuitively, we can assume that the size of the actual component is 0 and calculate the probability of this assumption being true. If the assumption is highly unlikely to be true than we can reject this assumption. This assumption is typically called the *null hypothesis* and the procedure is called *hypothesis testing*. Given n experiments for each of k alternatives, total variation is sum of squares of difference between each of $n * k$ experiments and overall mean, $\overline{y}$. Similarly, the variation due to the effects of the alternatives is the sum of squares of the differences between the mean of the measurements for each alternatives and overall mean. Lastly, the variation due to errors is the sum of the squares of the differences between the individual measurements and the mean of all measurements for a particular alternative, $\overline{y_j}$.

$$\overline{y_j} = \frac{\sum_{i=1}^{n} y_{ij}}{n} \tag{3.6}$$

$$\overline{y} = \frac{\sum_{j=1}^{k} \sum_{i=1}^{n} y_{ij}}{kn} \tag{3.7}$$

$$SSA = n \sum_{j=1}^{k} (\overline{y_j} - \overline{y})^2 \tag{3.8}$$

$$SSE = \sum_{j=1}^{k} \sum_{i=1}^{n} (y_{ij} - \overline{y_j})^2 \tag{3.9}$$

$$SST = \sum_{j=1}^{k} \sum_{i=1}^{n} (y_{ij} - \overline{y})^2 \tag{3.10}$$

$$= SSA + SSE \tag{3.11}$$

To test if *SSA* is statistically significant, we use *F-test*. We can say that *SSA* is significantly higher than *SSE* at α level significance if the calculated *F-statistic* is larger than the *critical F value*, $F_{[1-\alpha;(k-1),k(n-1)]}$. Note that $k-1$ is the degree of freedom for *SSA* and $k(n-1)$ is the degree of freedom for *SSE*. We can calculate *F-statistic* of the experiment by:

$$\begin{aligned} F\text{-}statistics &= \frac{SSA(k(n-1))}{SSE(k-1)} \\ &= S_a^2 / S_e^2 \end{aligned} \tag{3.12}$$

where S_a^2 is the variance of *SSA* and S_e^2 is the variance of *SSE*. Therefore, F-statistic is nothing but a ratio of the signal variance and the noise variance. *Critical F values* defines the minimum required ratio for the signal to have a statistical significance. *Critical F values* for a given confidence are tabulated in numerous literature [40].

A more compact representation of the same information is the *p-value* which is defined to be

$$p = 1 - P(F \leq f) \tag{3.13}$$

where $P(F \leq f)$ can be found from *F-density function* which can be found in numerous statistics text books [40]. *P-value* represents the probability that SSA is not statistically significant which is typically rejected if the *p-value* is less than 0.05.

Chapter 4

Storage Benchmark Programs

Storage system performance is one of the most critical factors in meeting overall system performance expectations. The performance variances of typical storage systems today can vary by orders of magnitude [41]. It is very easy to get misled by the performance numbers without understanding the relationship between the storage workload and its performance variances.

Characterizing the workload for storage systems is typically much more complex than it is for the processors. In a simple uni-processor case, it is possible to calculate metrics such as instruction per cycle based on the various cache misses, mis-predictions and number of instructions. However, storage systems typically have a much more state-dependent responses. For example, a cache miss does not always result in the same penalty, and the service time for a given request depends not only on the current request but also on all requests in the queue as well as the layout of the data. Therefore, two very similar workloads may perform very differently since the small differences could force the state transition path into completely different directions. For example, current layout of the data could force a sequential write to become fragmented. This then cause the sequential read of the same data to be slow. This in turn has effect on any requests that are scheduled after it within the request buffer.

We evaluated current storage benchmarking tools and analyze the effects of different parameter settings on various performance metrics. Based on the experiments, we quantify the effects of various parameters for three benchmarks. In addition, we also evaluate which parameters are needed to effectively cover the entire storage system performance space and we propose a new benchmark tool based on our evaluation.

4.1 Limitations of Current Approach

The best method for benchmarking a storage system would be to deploy the system under test in the real environment and look at the execution time of the processes that will be run. Obviously this approach is too costly to evaluate various feature and performance enhancements. Therefore, system architects and designers resort to either a set of traces collected from a real system or a set of synthetic benchmarks.

The problem with using a real trace is that there is no way to determine the coverage of the workload space. For example, a given Exchange server trace may perform very well on system A. However, it is difficult to determine if system A will perform as well for all Exchange servers. Furthermore, it is impossible even to provide any range of potential performances with a single trace. As mentioned above, a small change in the trace could result in a very different performance results.

Numerous synthetic benchmarks have been proposed over last few decades. The major benefit of these benchmarks is that they allow you to change the characteristics of the workload in a quantitative manner. However, current benchmarks either provide numerous parameters to set without providing any insights to the importance of each parameter [42, 43, 44, 45, 46, 47] or they provide a representative set of parameters that claim to represent a wide range of applications. When there are numerous parameters to set, users typically test a few set of ad hoc settings based on their experience which does not always result in an accurate representation of real workloads.

4.2 Benchmark Programs under Test

As a proof of concept, we evaluate two different benchmarks tools, PostMark [44] and FIO [42].

Typically, the evaluated systems are only tested on single configuration of a given benchmark. This is problematic since system A may perform better than system B for one configuration of the benchmark but not the others. It is critical to either define when the system A performs better or show that system A performs better B in wide

Name	Level Low	Level High
min_file_size	512B	4KiB
max_file_size	4KiB	16MiB
init_file_count	1000	10000
transaction_count	10000	100000
read_size	512B	32KiB
write_size	512B	32KiB
file_system_buffer	false	true
read_append_ratio	1:9	9:1
create_delete_ratio	5:5	9:1
directory_count	1	1000

Table 4.1: PostMark input parameters and input levels assigned to each parameter.

range of workloads by testing with multiple configurations that provide wide coverage.

4.2.1 PostMark

PostMark was designed for filesystem benchmarking with specific interest to performance of handling small file accesses in Internet software [44]. It provides 10 different parameter settings shown in Table 4.1 together with their level settings. Most of level decisions were straight forward except the *create_delete_ratio*. Initially, we set the low bound to be 1:9 where 10% of the operations are create and 90% deletion. However, we soon realized that the bound is too unrealistic. If the deletion of file happens more than the creation, the system will soon become empty of files. Therefore, we set the low bound to be 50% where the creation and deletion is equally like to occur.

Aside from these, it also provide two more benchmarking parameter for setting the random variable seed and controlling result format. The *transaction_count* only controls the duration of the run and not the arrival rate or the IO depth. Therefore, it is also a benchmark parameter. The *file_system_buffer* parameter is a system parameter which is forces all writes to bypass the filesystem buffer.

PostMark lacks control over arrival pattern and does not support sequential access pattern. Since these two characteristics play a major role in how the storage system performs, we can guess that the coverage of PostMark will not be great. Also, Postmark does not perform any overwrites which suggest that Postmark is unsuitable for

Name	Level Low	Level High
operation	read	write
access_pattern	sequential	random
files_used	1	100
min_file_size	512B	1MiB
max_file_size	1MiB	1GiB
min_block_size	512	4KiB
max_block_size	4KiB	64KiB
io_depth	1	100
overwrite	false	true
fsync	false	true
thinktime	0us	1000us
write_buffer_sync	false	true
file_service	roundrobin	random
thread_count	1	8
threads_similarity	false	true
posix_fadvise	false	true
async_io_engine	false	true
io_engine_queue	false	true
directio	false	true
buffer_alloc	malloc	mmap

Table 4.2: FIO input parameters and input levels assigned to each parameter.

evaluating storage systems where overwrites are expensive such as SSDs.

Postmark reports seven performance metrics, *transaction per second* (tps), create tps, read tps, append tps, delete tps, read throughput and write throughput.

4.2.2 Flexible IO Tester (FIO)

FIO was designed for benchmarking as well as stress/hardware verification [42]. It works directly on block devices as well as files. In this experiment, we tested it on files for the sake of consistency with PostMark.

FIO has over 30 different input parameters. However, some of those parameters are completely dependent on other parameters and was discarded. Also, some of the parameters, when set, caused bus errors on the system we tested. As a result, we extracted 20 input parameters as shown in Table 4.2. Every experiment was ran for 10 minutes which was sufficient to ensure that the system reached a steady state.

FIO can generate multiple independent streams of requests. In this study, we evaluate the effect of multiple streams as function of two parameters, *thread_count* and *threads_similarity*. The thread_count parameter is self-explanatory. When the thread_similarity is true, we generate either set of very similar streams with only difference being the random seed. When it is false, we generate threads that are at maximum distances from each other in the parameter space. This can be achieved by choosing different experiments within PB design.

FIO also allows you to control number of outstanding requests controlled by *io_depth*. The *posix_fadvise* option allows to you enable file advisory information to the operating system [48]. FIO provides 13 different types of IO engines. Some of them did not work on our system but we use 4 based on the two characteristics of the IO engines, synchronous/asynchronous and multiple/single buffer.

FIO provides a detailed distribution of performance measurements as well as aggregated numbers. In our study, we use 4 metrics, read throughput, write throughput, average read latency and average write latency. These metrics were chosen not only because they are important performance metrics but also so that the results can be compared against PostMark for out coverage analysis. For the experiments with more than 1 thread, the throughputs are simply added and the latency is averaged.

4.3 Quantative Analysis

4.3.1 Experiment Flow

The experiment is designed to evaluate the effect of various parameters of different benchmarks and coverage of each benchmark. We first use PB design to evaluate the effect of each parameter with minimum number of experiments. Once the effect has been calculated, we perform ICA on the performance results from each benchmark to estimate equal number of independent components. These components are clustered to generate a view of coverage of each benchmark.

This lets us analyze what are the key parameters and whether both benchmarks are needed for a through evaluation of storage systems.

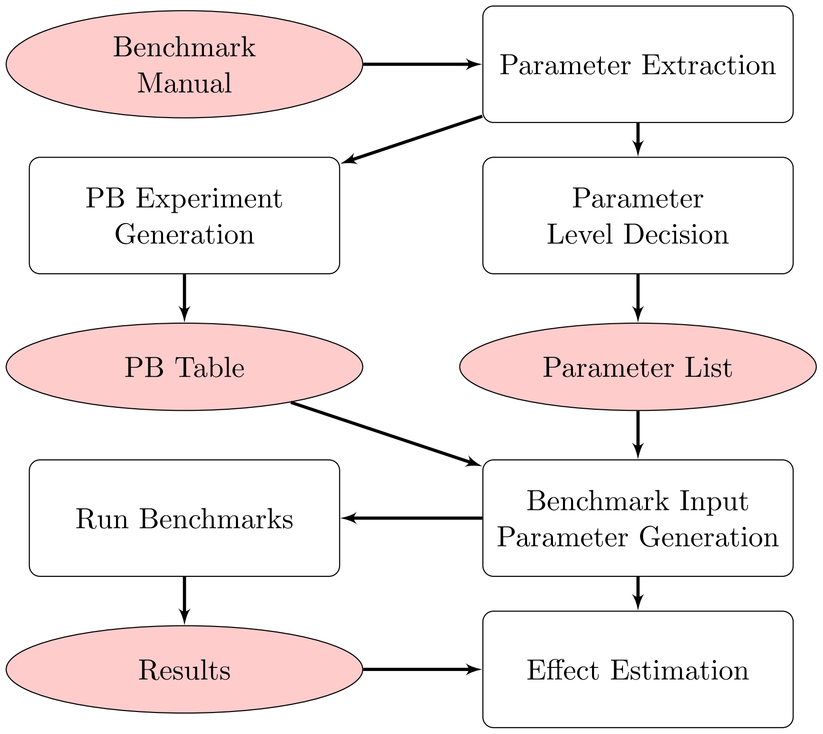


Figure 4.1: Experiment flow for estimating the parameter effects on performance.

Evaluation of Benchmark Parameter Effect

First, we use PB method to determine the effect of each parameters on each benchmarks separately. The experiment flow for effect estimation is shown in Figure 4.1. *Parameter Extraction* phase tries to extract parameters from benchmark documentation in such a way that each parameter can be varied independently of others. This is not always easy as shown in the description of the benchmarks.

For every parameter extracted, a two extreme level values are designated as shown in Table 4.1 and Table 4.2. Also, once the number of parameter is known, the size of PB design is also known which is required to generate PB design table. Since the number of parameters for Postmark is 10 and FIO is 22, PB table of size 12 and 24 are used respectively. While any Hardamard matrix [49] can be used, we use the matrices shown in equation 4.1 for 11 factor experiment. PB table for 23 factors can be found in

Plackett and Burman's original paper [33] and we omit it due to the space constraint.

$$\mathbf{PB}(12) = \begin{bmatrix} 1 & 1 & -1 & 1 & 1 & 1 & -1 & -1 & -1 & 1 & -1 \\ -1 & 1 & 1 & -1 & 1 & 1 & 1 & -1 & -1 & -1 & 1 \\ 1 & -1 & 1 & 1 & -1 & 1 & 1 & 1 & -1 & -1 & -1 \\ -1 & 1 & -1 & 1 & 1 & -1 & 1 & 1 & 1 & -1 & -1 \\ -1 & -1 & 1 & -1 & 1 & 1 & -1 & 1 & 1 & 1 & -1 \\ -1 & -1 & -1 & 1 & -1 & 1 & 1 & -1 & 1 & 1 & 1 \\ 1 & -1 & -1 & -1 & 1 & -1 & 1 & 1 & -1 & 1 & 1 \\ 1 & 1 & -1 & -1 & -1 & 1 & -1 & 1 & 1 & -1 & 1 \\ 1 & 1 & 1 & -1 & -1 & -1 & 1 & -1 & 1 & 1 & -1 \\ -1 & 1 & 1 & 1 & -1 & -1 & -1 & 1 & -1 & 1 & 1 \\ 1 & -1 & 1 & 1 & 1 & -1 & -1 & -1 & 1 & -1 & 1 \\ -1 & -1 & -1 & -1 & -1 & -1 & -1 & -1 & -1 & -1 & -1 \end{bmatrix} \quad (4.1)$$

Each row of the PB table represents an experiment and columns represent parameters. -1 value represents a *low level* while 1 represents a *high level*. It should be noted that not every column can be assigned a parameter. You can leave any of the columns out for actual run of the experiments. However, it is important to ignore the effect of the unused column later.

The number of experiment required is equivalent to the number of rows of the matrix. Every experiment is repeated three times and mean is used to calculate the effect. Given n experiment result $\mathbf{y}$, the effect, e, of a parameter represented by column i of PB matrix is simply calculated by

$$e = \frac{\mathbf{PB}_i \cdot \mathbf{y}}{\sum_i (\mathbf{PB}_i \cdot \mathbf{y})}. \quad (4.2)$$

Here the denominator is a simple normalization factor to ensure that sum of all effects is 1.

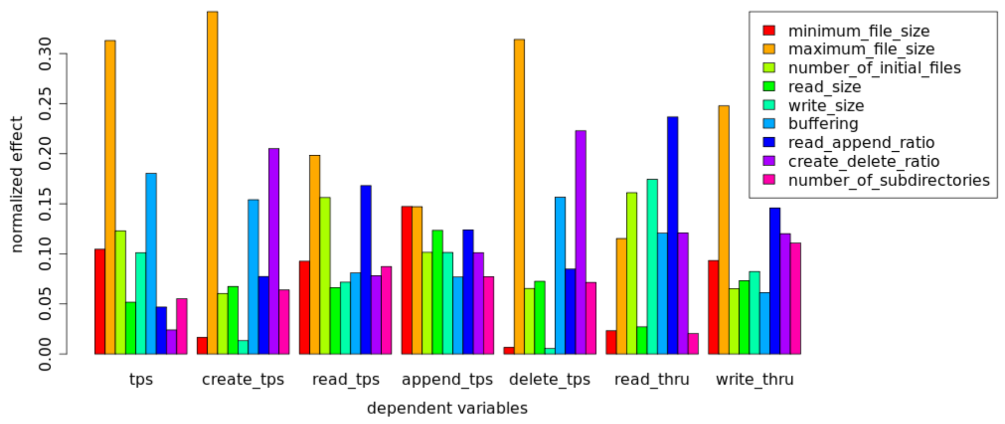


Figure 4.2: PostMark parameter effects. You can see that some parameters have high effect on a particular kind of performance metric while some have high effect on almost all performance metrics.

Evaluations of Benchmark Coverage

We use R JADE package [50] to generate 2, 4, 8 independent components from the input parameters and the read and the write throughput measured from each benchmark. Since the read and the write throughputs are the only common metrics reported from the two benchmarks, only they can be directly compared.

Once components are estimated, we evaluate the Euclidean distances between all components which corresponds to a single performance metric are calculated. Based on the distance, the components are clustered. The number of clusters to be used were estimated using dendrograms [51].

We evaluate the best number of components for clean clustering and determine the coverage of each benchmark on two performance metrics.

4.4 Results

4.4.1 Effects

Figure 4.2 shows effect of each parameter on each performance metrics reported by Postmark. The *maximum_file_ size* and *buffering* are the clearly two most important

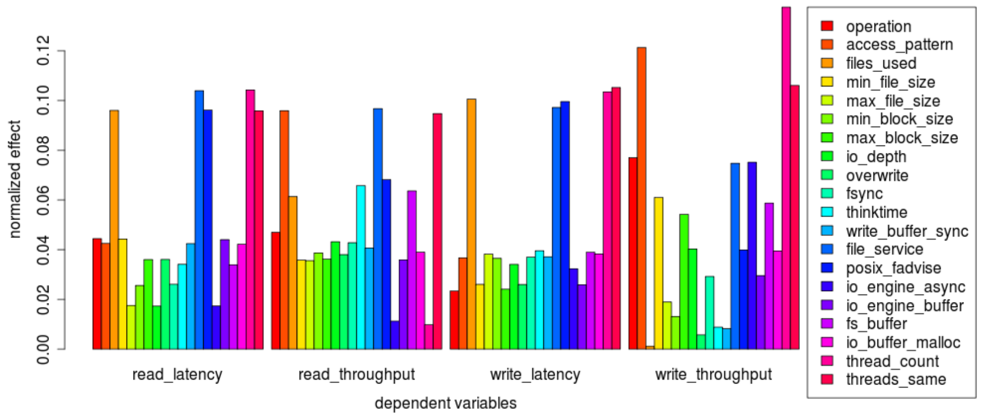


Figure 4.3: FIO parameter effects. Effect of application threading is the most pronounced in this benchmark. However, it will be shown later that the effect of threading can be triggered by other parameters as well.

---

factor in deciding the overall tps. The *buffering* parameter controls whether file system buffer should be bypassed using direct IO and the importance of the parameter is obvious. However, the effect of the *maximum_file_ size* is less obvious. Potentially, the high effect is due to the fact that larger file sizes allow a room for longer sequential accesses while the small file sizes force overall access pattern to behave in a more random fashion.

You can also see that different performance metrics are effected differently. The *create_delete_ratio* is clearly and obviously important for create tps and delete tps while for append tps, file sizes are more important. Another interesting to note is that while read throughput is affected the most by the *read_append_ratio*, the write throughput is much more impacted by the file size.

Table 4.3 shows list of parameters with the most effect on each performance metric such that sum of the effects exceed 50%of overall effects. It is shown that roughly 50% of the all performance variances can be captured using just 2-4 parameters. For example, three parameters, *maximum_file_size*, *number_of_initial_files* and *buffering*, contribute to 61% of overall tps variation observed. This equate to 4 to 16 experiments for exhaustive evaluation. This is a huge improvements over 2048 experiments required for original 12

Performance Metric	Parameters	Total Effect
tps	maximum_file_size number_of_initial_files filesystem_buffer	0.6164
create_tps	maximum_file_size create_delete_ratio	0.5469
read_tps	maximum_file_size number_of_initial_files read_append_ratio	0.5229
append_tps	minimum_file_size maximum_file_size read_size read_append_ratio	0.5418
delete_tps	maximum_file_size create_append_ratio	0.5369
read_thru	number_of_initial_files write_size read_append_ratio	0.5722
write_thru	maximum_file_size read_append_ratio create_delete_ratio	0.5137

Table 4.3: PostMark input parameters required to cover at least 50% of overall effect for each performance metrics reported. The maximum number of parameters required does not exceed 4.

parameters. Furthermore, it is clear that different input combinations must be tested for each performance metric which must be taken into consideration when benchmarking storage systems.

Figure 4.3 shows the effect of each parameter on four performance metrics measured and Table 4.4 shows list of parameters with the most effect on each performance metric such that sum of the effects exceed 50%of overall effects. It is interesting to note that the most important parameters for read and write latencies are the same. We can assume that read and write operations themselves do not affect the latency in our system. This can expected, since on HDD, read and write operations do not result in significantly different response time. Furthermore, we see that the *threads_same* parameter is important for all metrics. This indicates that the interference between

Performance Metric	Parameters	Total Effect
read_latency	files_used file_service posix_fadvise thread_count threads_same	0.5406
read_throughput	access_pattern write_buffer_sync threads_same	0.5462
write_latency	files_used file_service posix_fadvise thread_count threads_same	0.5060
write_throughput	access_pattern thread_count threads_same	0.5170

Table 4.4: FIO input parameters required to cover at least 50% of overall effect of each performance metrics reported. The maximum number of parameters required does not exceed 5.

the threads can seriously affect the performance. While the *thread_count* parameter is also important, the effect on the read throughput is shown to be negligible. Both throughputs are sensitive to the *access_pattern* which is expected but surprisingly the latency is not. Instead, they are much more sensitive to the *posix_fadvise* setting. This suggests that latency is not determined by the seek time but rather cache miss penalties.

4.4.2 Coverage of Benchmarks

The distance between the independent components as well as their clustering is shown in Figure 4.4 and Figure 4.5. In these graphs, each leaf node represents an independent component. Since the independent components have no physical meaning associated with them, we number them from 1 to 8 per benchmark. We only show the results of 8 components for the brevity. The closer the leaves are on the tree, more similar effect they have on the result. The *height* represents the difference between the subtrees. therefore, evenly spread out components of both benchmark on x-axis suggest similar

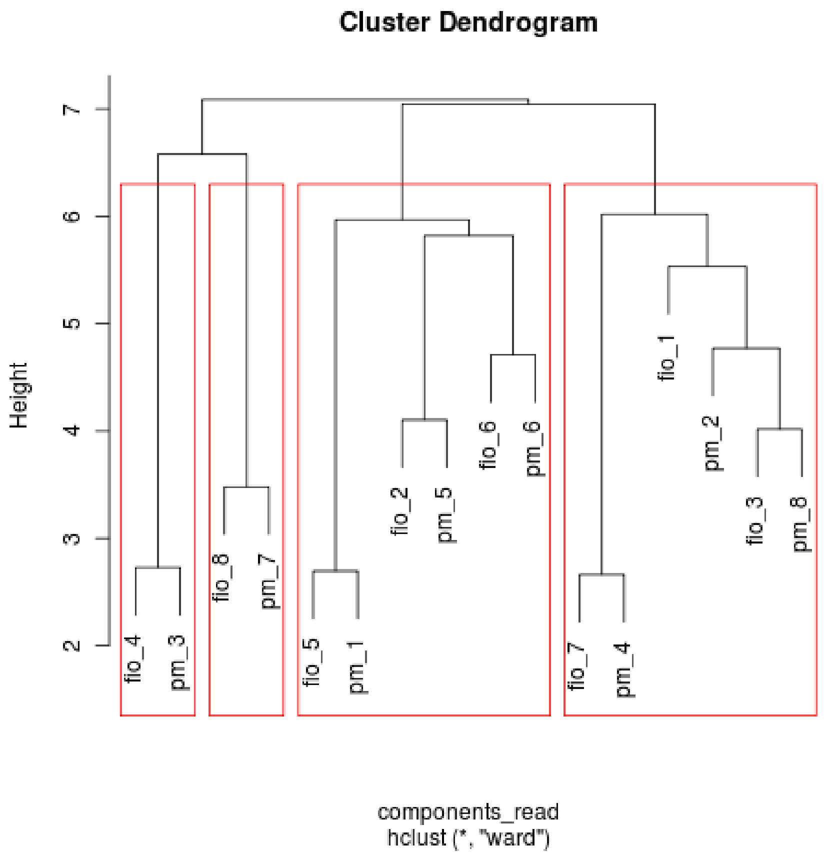


Figure 4.4: Read performance coverage of of independent components of PostMark and FIO.

---

coverage, while the height of the tree suggest the granularity of the coverage. For example, the FIO component 4 and PostMark component 3 have the similar effect on the read throughput based on the Figure 4.4. At the same time, FIO component 8 and PostMark component 7 also have similar effect. However, there is a large gap between the two sets of components which are not covered by any of the components as suggested by the *height* of the edges connecting the two subtrees.

An interesting observation is that for read throughput, FIO and Postmark have roughly the same coverage. This is interesting since PostMark have far fewer parameters. It can be concluded if the metric of interest is read throughput, than it does not matter

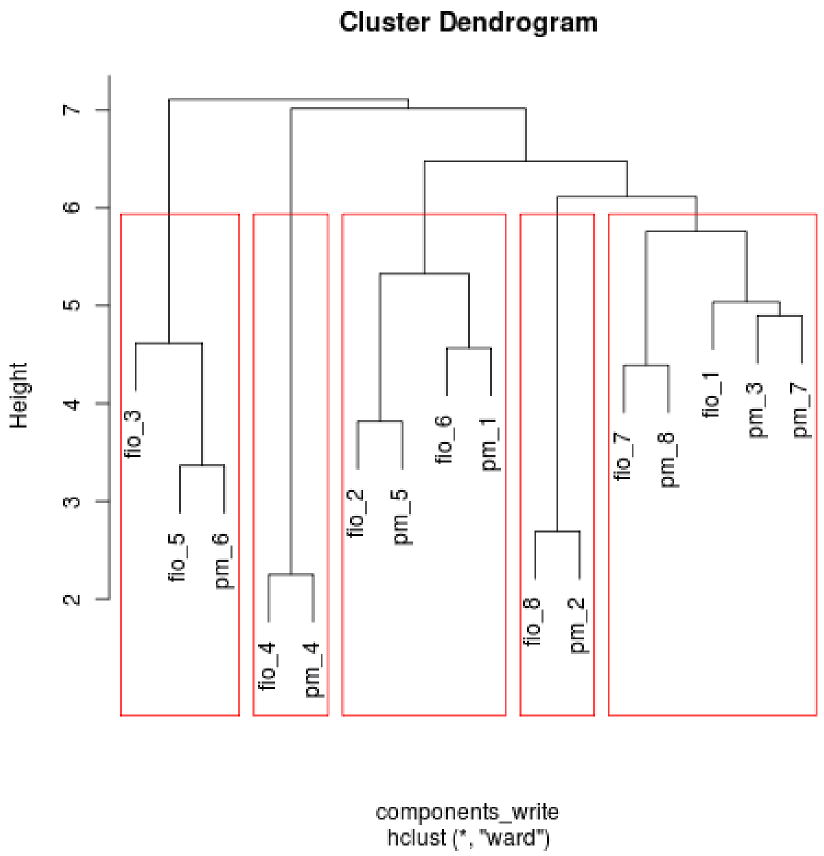


Figure 4.5: Write performance coverage of of independent components of PostMark and FIO.

---

which benchmark you use as long as you can test the key parameters thoroughly.

For the write throughput, FIO does provide wider coverage to the left with component 3, while PostMark provide wider coverage on the right with component 3 and 7. We can safely conclude that each benchmark provide one of more input parameters that affect the write throughput that is not provided by the other benchmark.

The input parameters that affect the read and write throughput performance shows some difference. They all depend on the size of the files, number of files and mix of operations. However, write throughput is also dependent on file creation and deletion which are not taken into account in FIO. Conversely, PostMark does not consider the number of threads which is another important factor provided by FIO.

We believe that a good benchmark should have all those components as inputs to provide a wider coverage.

4.5 Conclusion

Clearly, any storage performance evaluation should consider the effect of critical input parameters that has been described in this paper. We recommend that at least the critical parameters shown in Table 4.3 and 4.4 should be tested thoroughly to gain an approximate sense of the system's performance variation.

The methodology presented in this paper can be applied to any benchmarking tool with set of input parameters that can be adjusted. The key challenge is deciding the range of each parameter. Once this is done, the entire process can be automated.

Even if the system is designed for a specific workload, it is still beneficial to follow the methodology described in this paper. Since the characteristics of the actual workload is known, a tighter level bounds can be assigned to each parameter. The resulting effect of the parameters can describe how the system reacts to different parameters and can provide valuable insights to workload-system interactions.

We also show that different benchmarks can still cover roughly the same performance space if the input parameters are chosen appropriately. We suggest that performing a set of through experiments on a single benchmark provides more accurate description of system performance than running a single experiment on the multiple benchmarks.

However, it is clear that there are specific performance space that can only be explored with a specific benchmark. Since it is difficult to determine what benchmarks cover how much of the performance space, we propose a new benchmark with following properties.

- IO benchmark tool should provide maximum coverage of storage system's operational space with minimum parameter settings.
- The parameters of benchmarking tools should be as independent as possible.
- The parameters should be exclusively defined as either the system parameter, the workload parameter or the benchmark parameter.

- The target interface which the benchmark is testing needs to be clearly defined.
- A minimum validation of results should be handled. (eg. Results that suggest under-utilization of target system such that the performance result does reflect its capability.)

With this kind of benchmarking tool, most of the systems can be evaluated with a single benchmark with multiple parameter settings. This tool would allow different researcher and developers to report their performance in a more coherent manner which in turn makes performance comparison and reproducing the result easier.

As a future work, we plan to conduct a comprehensive analysis of more benchmarks and design a new benchmark that provides a wide coverage with few independent parameters.

Chapter 5

Storage Performance Model

Storage Performance Model predicts the performance of a workload on a given data store. Before describing Storage Performance Model in detail we define a few variables in Table 5.1 to help describe our model.

5.1 Storage Performance Model Framework

Figure 5.1 outlines how the *Storage Performance Model* is generated and maintained. The edges in Figure 5.1 represents the following procedure.

1. The test vector is run on data stores.

2. Average latencies, P, as well as the characteristics of the workload (X, R, Z and O) per data store are collected at 20 seconds interval.

W	Workload	M_W	Workload Model
S	Data Store	M_S	Data Store Model
P	Performance(Latency)	M_P	Performance Model(Prediction)
X	Random %	R	Read %
Z	IO size	O	Outstanding IO

Table 5.1: Variables defined to describe the Storage Performance Model.

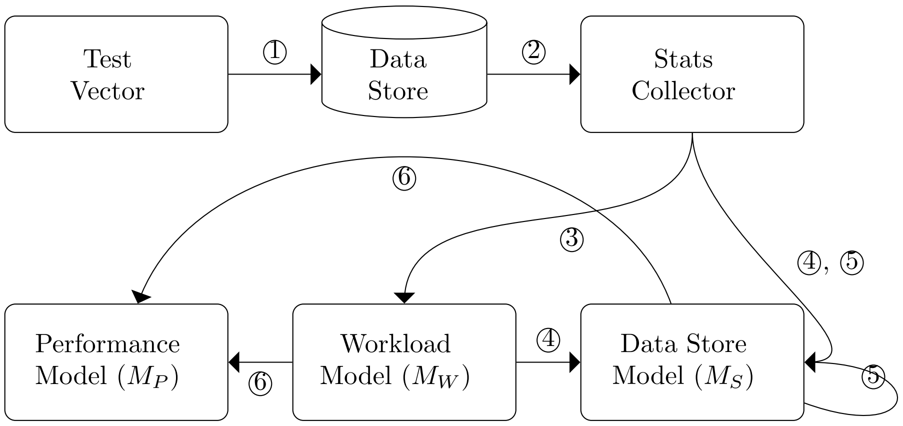


Figure 5.1: Storage Performance Model framework.

3. Workload model is generated by determining which of the workload characteristics and their interactions actually affect P.

4. From the workload model and the latencies, the data store model is generated via linear regression.

5. The model is updated periodically based on the new data collected while data stores are active.

6. The performance model is derived from the workload and data store model.

In this section we will describe and justify each step of the process involved in generating the performance model (M_P).

5.1.1 The Goal

A system's performance is mainly determined by the system itself and its workload. Therefore, there should exist a function, f_P, that can map a given data store (S) and workload (W) onto performance (P).

$$P = f_P(S, W) \tag{5.1}$$

We define *IO workloads* to be the set of all possible workloads, $W = \{w_i\}$ and *Performance*, P, to be the average IO latency. However, the dimensions of S and W are too large to be parameterized. Furthermore, even if they were not, deriving an accurate non-linear function f_P is difficult. Therefore, we assume the form of f_P to be some linear combination shown in Equation 3.5 and fit S and W into M_S and M_W such that P can be approximated.

$$P = M_W M_S + \epsilon = M_P + \epsilon \tag{5.2}$$

The goal of *Storage Performance Model* is to derive M_W and M_S such that resulting M_P is as close to the actual P as possible.

5.1.2 The Problem

Since we can measure P, we need to determine M_W to solve the Equation 5.2 for M_S. Once M_W is defined, we can use linear regression to derive M_S. Without the knowledge about the true distribution of ϵ, we assume it to be Gaussian and solve for M_S by:

$$({M_W}^T M_W)^{-1} {M_W}^T P = M_S \tag{5.3}$$

Equation 5.3 is proven to result in smallest ϵ if the ϵ is Gaussian distributed [52]. While the true distribution of ϵ may not be Gaussian, we found the resulting ϵ small enough to justify the assumption. Therefore, the problem is finding a good M_W such that ϵ can be minimized. There is a practical constraint we must satisfy when solving this problem. The characteristics of workloads that can be used to derive M_W must be collected from the system at run time without too much overhead. VMware ESXi 5.0 already collects read ratio (R), average request size (Z) and average number of outstanding IOs (O) at 20 seconds granularity [53]. We implement a mechanism to collect one more characteristic to measure amount of seeks (X) since it is well known that randomness of IO has large effect on storage performance. The problem is now reduced to finding a good M_W from X, R, Z and O such that ϵ is minimized.

5.1.3 Test Vector

To investigate how M_W can be derived from X, R, Z and O, we first define the space of all possible workloads in terms of those characteristics.

$$random\%(X) = \{r | r \in \mathbb{I}, 0 \leq r \leq 100\} \tag{5.4}$$

$$read\%(R) = \{o | o \in \mathbb{I}, 0 \leq o \leq 100\} \tag{5.5}$$

$$IOsize(Z) = \{s | s \in 2^n B, n \in \mathbb{N}, 9 \leq n \leq 19\} \tag{5.6}$$

$$OutstandingIO(O) = \{i | i \in \mathbb{N}, 1 \leq i \leq 128\} \tag{5.7}$$

Note that we limited the range of O and Z which is not enforced theoretically. However, they are typically enforced by the system resource limitations such as buffer size. While different limits can be used, we believe these to be widely applicable in today's systems. Another thing to note is that we have only defined the R and X within $\mathbb{I}$. While this may result in loss of resolution, R and X are always discrete in practice since there can only be finite number of requests and we assume that the close enough values always result in similar latency. The size of the entire space is $|O| * |Z| * |R| * |X| = 14,363,008$. If we tested each point in space for 2 minutes, it would take 54 years to explore all possible combinations. Therefore we sample this workload characteristics space such that;

$$X = \{0\%, 25\%, 50\%, 75\%, 100\%\} \tag{5.8}$$

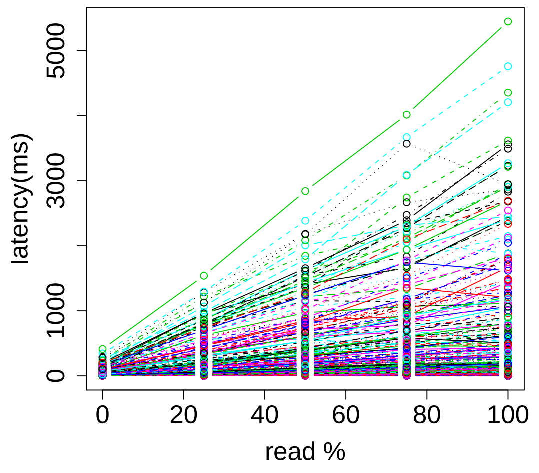
$$R = \{0\%, 25\%, 50\%, 75\%, 100\%\} \tag{5.9}$$

$$Z = \{1K, 2K, 4K, 8K, 16K, 32K, 64K\} \tag{5.10}$$

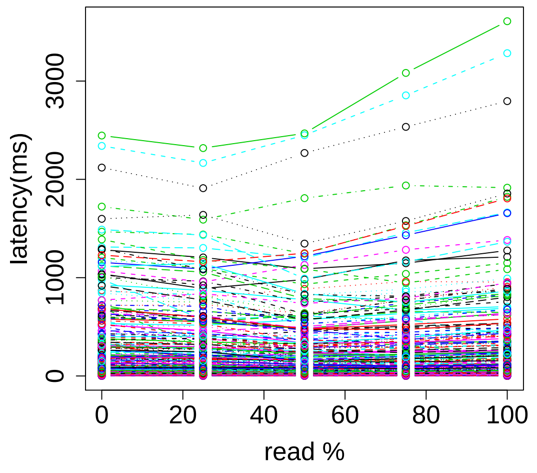
$$O = \{1, 2, 4, 8, 16, 32, 64\} \tag{5.11}$$

which would require 1600 tests and can be done in a couple of days. This sampled space is our *test vector*. The idea of vectorizing a multidimensional continuous space into a single discrete vector is analogous to the way application characteristics are represented in *HBench* [54].

The reduction in workload space due to sampling 1600 datapoints results in no significant loss of accuracy in the model. However, the space is still too large for the



(a) E1



(b) N2

Figure 5.2: Effect of *read%* variation on the *latency* of E1 and N2 data stores. Each line represent latency (P) vs. read% (R) while other parameters(O, Z and X) are fixed. The effect of R on E1 is much more predictable(therefore significant) than N2.

---

modeling to be carried out at low cost on on-line systems. The sampling technique that would provide a reasonable model with minimum testing is an interesting topic we leave for the future research.

We generate our test vector using Iometer [55].

5.1.4 Workload Model

How to describe an IO workload in a generic manner is an open question [56, 57, 58, 59]. Unlike Basil [60] and Pesto [61], our model generates a separate M_W for each data store. While it is tempting to derive M_W in a data store independent manner, we found the resulting performance model (M_P) to be neither robust nor accurate. This is because each data store interacts with the characteristics of the workload in a different manner.

Figure 5.2 shows the relationship between P and R for two data stores E1 and N2 from Table 5.2. The plot shows the latency change observed on two different data stores while varying R and fixing X, Z and O. It is clear that R has distinct effect of E1 but

Name	No. of disks	RAID	Interface	Make
E1	3	0	SATA	EMC
E2	6	5	FC	EMC
N1	3	5	SATA	NetApp
N2	7	6	FC	NetApp

Table 5.2: Summary of test data stores used in Storage Performance Model testing.

not on N2. This means that M_W for E1 should contain R but not N2. An important thing to note is that the inclusion of R in M_W for N2 will result in over-fitting since it is likely to only add to the noise. Therefore, removing R from the workload model actually results in a more robust model that is less sensitive to noise.

Another difficulty arises from the fact that the interactions between the characteristics are significant and cannot be ignored. While it is possible to use a simple tuple of X, R, Z and O, we found the resulting ϵ to be very large. Interactions between the characteristics can be factored in by simply adding additional terms for the products of two or more characteristics. For example, the interaction between X and R can be factored in to the model by adding the additional term $X : R = X * R$. Therefore, given 4 characteristics, there are $2^4 - 1 = 15$ possible terms including the interactions. To check which of the 15 terms actually have a significant effect on the performance, we conduct ANOVA at 95% confidence controlling the 15 terms (using the test vector) and observing the latency on four test data store shown in Table 5.2.

The *p-values* derived from the ANOVA test is shown in Table 5.3. These values have a significant effect on performance (P) with 95% confidence. For example, the M_W for E2 is a row vector of the form;

$$M_W = \begin{bmatrix} O & X * O & Z * O & X * R * O & X * Z * O \end{bmatrix} \tag{5.12}$$

The resulting M_W accounts for any interactions as well as their significance on a given data store. While the cost of the experiment is not cheap, this only has to be done once per data store type in the system.

E1		E2	
factor	p-value	factor	p-value
O	4.83e-06	O	< 2e-16
R:O	3.09e-08	X:O	7.92e-15
Z:O	3.45e-03	Z:O	< 2e-16
X:R:O	< 2e-16	X:R:O	< 2e-16
X:Z:O	4.51e-03	X:Z:O	3.98e-05
X:R:Z:O	5.99e-11		
N1		**N2**	
factor	p-value	factor	p-value
O	6.85e-03	O	3.06e-16
X:O	< 2e-16	X:O	< 2e-16
Z:O	< 2e-16	Z:O	< 2e-16
X:R:O	2.33e-05	X:R:O	4.68e-08
X:Z:O	< 2e-16	X:R:Z:O	< 2e-16
X:R:Z:O	< 2e-16		

Table 5.3: Result of factorial ANOVA with latency as response and characteristics of workload as the treatment. Only the factors with p-values less than 0.05 are shown. With 95% probability, these factors have a predictable effect on performance.

5.1.5 Data Store Model

Using the M_W, we find the M_S using Equation 5.3. Table 5.4 shows the resulting M_S for the four test data stores. The *coefficients* columns of each data store form a column vector which is M_S of that data store. The accuracy of M_S depends heavily on the number of data points. While initially it can be derived from our test vector, it can be updated from the data collected in production environment to improve its accuracy.

5.1.6 Performance Model

Finally, we are now ready to calculate M_P, which is the prediction of P, by taking the cross product of M_S and M_W. One interesting thing to note from Table 5.3 is that all the factors contain O. This is because the average latency (P) of all data stores are linear to the number of outstanding IOs (O). Note that it also means that P is not linear to X, R and Z.

We have tested 12 different data stores and verified that the linear relationship

E1		E2	
factor	coefficients	factor	coefficients
O	1.064	O	8.370e-01
R:O	2.828e-02	X:O	6.163e-03
Z:O	1.505e-05	Z:O	1.172e-05
X:R:O	2.277e-03	X:R:O	4.079e-04
X:Z:O	2.545e-07	X:Z:O	9.970e-08
X:R:Z:O	9.385e-09		
N1		**N2**	
factor	coefficients	factor	coefficients
O	1.428	O	1.310
X:O	1.653e-01	X:O	1.403e-01
Z:O	1.293e-04	Z:O	9.078e-05
X:R:O	-3.24e-04	X:R:O	1.579e-06
X:Z:O	2.052e-06	X:R:Z:O	1.514e-08
X:R:Z:O	2.613e-08		

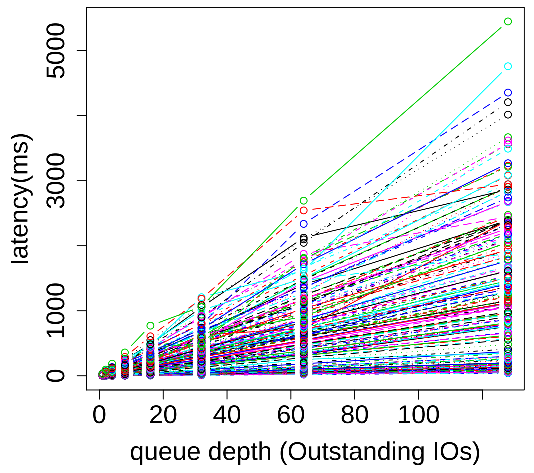
Table 5.4: Data store model (M_S) for four test data stores. (M_S) is a column vector whose entries are shown in the *coefficients* column of the each data store.

between the outstanding IOs and the latencies hold. Therefore, we can rewrite Equation 5.3 as;

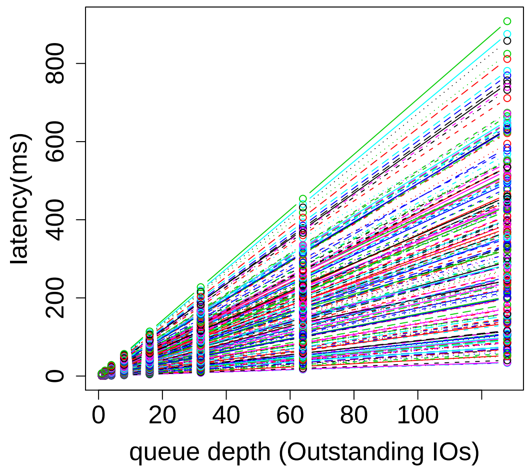
$$\begin{aligned} M_P &= M_W M_S = M'_W M_S O \\ M'_W &= \frac{M_W}{O} \end{aligned} \tag{5.13}$$

Since the *outstanding IO* (O) is the number of IO requests outstanding in the queue and P is time the requests spend in the system, by Little's law [62], $M'_W M_S$ is the average inter-arrival time of requests. It is also the average service time of requests at equilibrium and its inverse is the throughput (T) in IO/s. Pesto [61] describes the linear relationship between O and P in much more depth and we refer interested readers to their work.

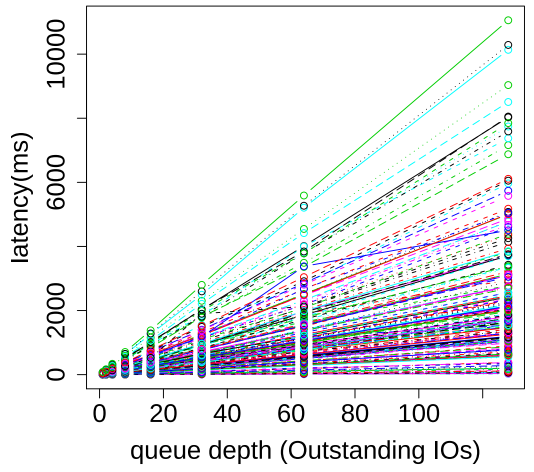
The Figure 5.3 shows latency (P) change as O is varied while fixing X, R and Z. Except for some exceptions caused by the measurement errors, all plots show linear relationship between O and P. $M'_W M_S$ is called *LQ-slope* in Pesto, and we continue to use the terminology for the sake of consistency. Therefore, we can rewrite Equation 5.13



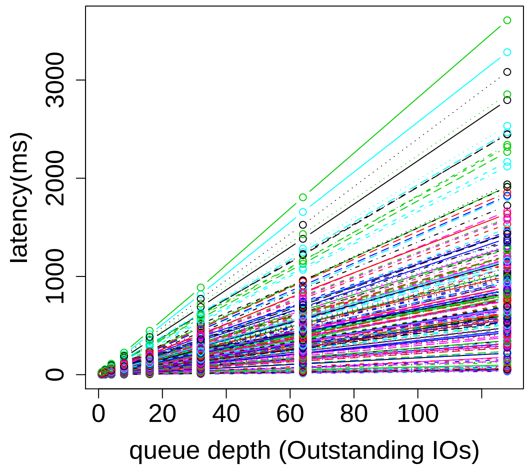
(a) E1



(b) E2



(c) N1



(d) N2

Figure 5.3: Effect of outstanding IO (O) variation (from 1 to 128) on the latency of test data stores. Each line represent latency (P) vs. outstanding IO (O) while other parameters are fixed. Except for the few noise, the relationship is clearly linear for all data stores.

to our final form of M_P.

$$P = M_P + \epsilon = LQslope * O + \epsilon = M'_W M_S O + \epsilon \tag{5.14}$$

There are two benefits of using Equation 5.14 compared with Equation 5.2. First benefit is that O no longer needs to be modeled. This effectively reduces the size of M_S and M_W simplifying the modeling process as well as prediction process. The second benefit comes when we derive the *Workload Aggregation Model* which we will describe in Chapter 6.

5.1.7 Prediction Interval

The most important feature of our model is that it provides the confidence of its own prediction. The model keeps track of the variance (σ^2) of ϵ from which the prediction interval is calculated to be;

$$M_P \pm \frac{1.96\sigma}{\sqrt{n}} \tag{5.15}$$

at 95% confidence. Therefore, the 95% of actual P should fall within the confidence interval. n is the number of data points and $\pm 1.96\sigma$ is used to capture the 95% of the Gaussian distribution which we assume is the distribution of ϵ. In short, the prediction interval is proportional to the variance of residuals and inversely proportional to the square root of the number of observations. If the prediction interval is too large, the prediction is useless. To tighten the interval we can do two things. First is to reduce the confidence level. However this approach may lead to too many bad predictions that could result in bad decision making. Another approach is to increase the number of observations. In our model, the performance statistics and the workload characteristics are collected in real time to update the M_S in effort to tighten the prediction interval.

5.1.8 Stats Collector

In order to update M_S, the *stats collector* collects workload characteristics and performance in real time. P is measured at vSCSI layer as the completion time subtracted from time of queueing per request. R and Z are simply read from IO request attributes. O is measured from the queue depth of the *vSCSI* [63] device which is created per virtual machine (VM) and virtualizes the SCSI device layer. We had to implement

	Basil	Pesto	FFLM	Proposed
Min.	-27380	-1.33	-1669	-1667
1st Qu.	-441.8	12.03	-0.1037	-8.610
Median	-105.2	52.67	-0.7699	-0.1033
Mean	-565.6	292.5	1.860e-13	1.191
|Mean|	642.4	292.5	44.43	44.34
3rd Qu.	-22.31	243.8	7.872	9.043
Max.	6242	10590	1141	1146

Table 5.5: Comparison of residuals (ϵ) for different modeling techniques. The residuals were aggregated from all four data store models. 1st Qu. and 2nd Qu. represents 25% and 75% quantile values respectively. |Mean| represents mean of absolute values of residuals. Units are in ms.

the mechanism to collect X at SCSI device layer since vSCSI is unaware of large seeks that result from seeking between multiple virtual disks in a data store. X is measured by simply looking at the percentage of seeks larger than 10 times the average request size. The reason for the multiplier is that some workloads exhibit significant variance in their request size. Since the seeks are measured in terms of address difference between consecutive requests, large requests could falsely register as seeks even when they are sequential. All stats are averaged and reported to *Resource Manager* at 20 seconds granularity.

5.2 Prediction Accuracy

This section evaluates the accuracy and the robustness of our performance model. Specifically, we use Basil [60] and Pesto [61] as the baseline models and assume that the *full factorial linear model* (FFLM) is the ideal case since it uses all possible linear combinations. To evaluate the accuracy we use *test vector* and run them against for 4 test data stores (Table 5.2). The resulting 6400 latencies are compared against the predictions of 4 techniques.

Table 5.5 shows aggregated ϵ values, the difference between the prediction (M_P) and the measurements (P), of different modeling techniques. The values were aggregated from latency prediction of all four data stores under test (from 6400 data points). It shows that the performance of Basil is unacceptable in a heterogeneous data store environment for most applications with high mean ϵ of $642ms$. This is expected since Basil generates a single model for any data stores. Pesto takes into the account the performance difference of each data store. However, it assumes that one data store is simply more powerful than the other regardless of the workload. While it provides a better accuracy than Basil by 60% on average, it still shows unreasonable maximum ϵ of $10s$ and average ϵ of $260ms$.

Our proposed model shows a much stronger predictive power with average ϵ of $44ms$ and the maximum ϵ of $1.3s$. This is an 82% reduction for the maximum residual and 83% reduction for the average residual compared to Pesto. In fact, our model performs slightly better than FFLM although the difference is too small to be meaningful. However, this model is capable of faster model updates and prediction since it only uses 4-5 terms versus 16 terms used by FFLM. Furthermore, 50% of all predictions (between the 1st and 3rd quantile) exhibit less than $10ms$ of error.

In short, our proposed model shows 80% improvement in overall prediction accuracy on heterogeneous workloads and data stores compared to the most recent technique [61].

5.3 Prediction Interval

Perhaps the most significant benefit of our model is its ability to provide a prediction interval at pre-defined confidence for its predictions. Figure 5.4 shows the average latencies of 20 different virtual disks running different workloads randomly selected from our test vector on four test data stores. The average was evaluated over two minutes. The predicted latency and its corresponding interval (shown as the error bar) was calculated using our model at 95% confidence (allowing 5% of measurements to be outside of the prediction interval).

Figure 5.4 shows that the predictions can be off by up to $500ms$ (*workload 16*). However, all measurements fall within the prediction interval. In fact, out of 1600 workloads ran against 4 data stores in the previous experiment, our model failed to capture the measured latency within its prediction interval 423 times which is 6.6% (close to predefined 5%) of 6400 predictions made.

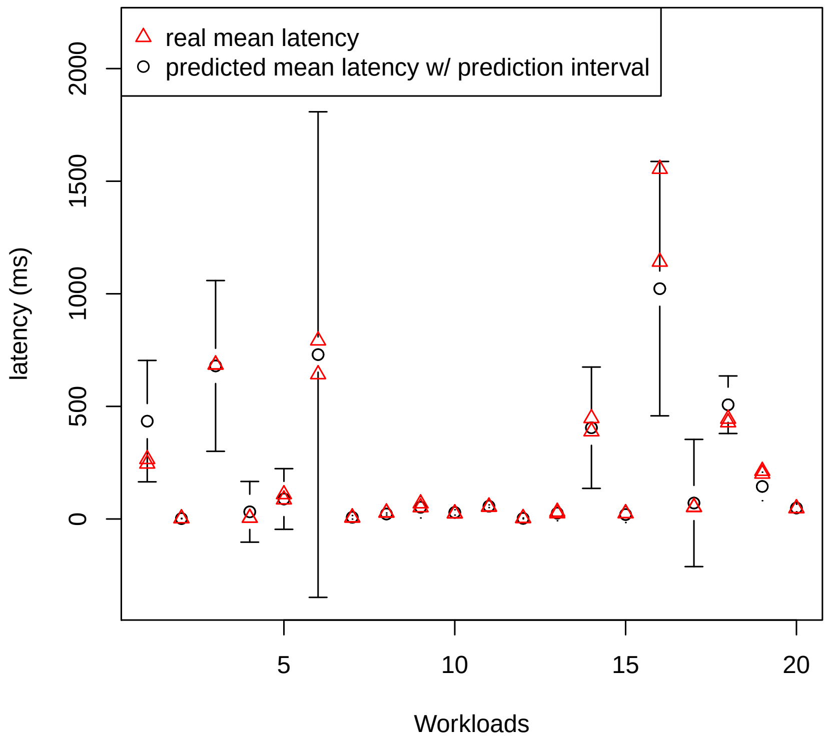


Figure 5.4: The average latency (over 20 second intervals) observed from 20 different workloads (running on separate virtual disks) on 4 different test data stores during 1 hour interval. The max and min latency is plotted. Also plotted are the model's 95% prediction intervals.

Another interesting thing is that there were roughly 5% of workloads whose confidence interval is so large, like the *workload 6* in Figure 5.4, that the prediction itself is meaningless. Pesto [61] claimed that these workloads had specific characteristics (such as purely sequential) and can be filtered out. However, we found them to be more of random occurrences and difficult to predict in advance. By default, the prediction is done every 8 hours to consider the possible moves in current VMware products [64]. However, we force prediction every 2 minutes and observed prediction interval change after 5 hours. In theory, the prediction interval should converge after about 2 hours (12 measurements), however we found the convergence to take a much longer time. We suspect the reason to be fluctuation in the workload. A detailed analysis and speeding up this convergence will be our future work.

E1	E2	N1	N2
25.798101	4.922433	16.666000	6.753044

Table 5.6: Pesto LQ-slope of four test data stores. The units are ms/io. According to these slopes, E2 is the fastest data store and E1 is the slowest.

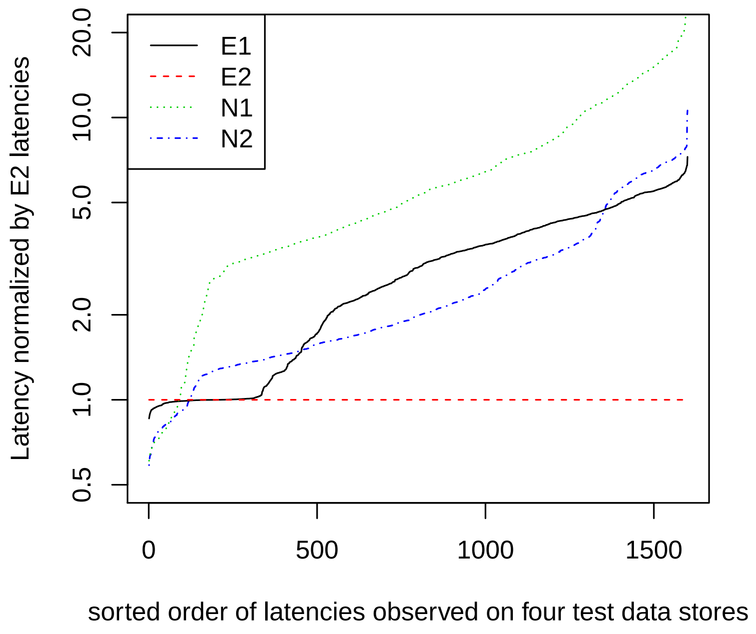
In short, our model is capable of providing prediction interval at a specified confidence level. The level of confidence can be lowered to trigger more aggressive load balancing with tighter prediction interval. On the other hand a higher confidence interval will result in moves that are only triggered when predicted benefit is large enough to over come the larger prediction interval.

5.4 Handling Heterogeneous Data Stores

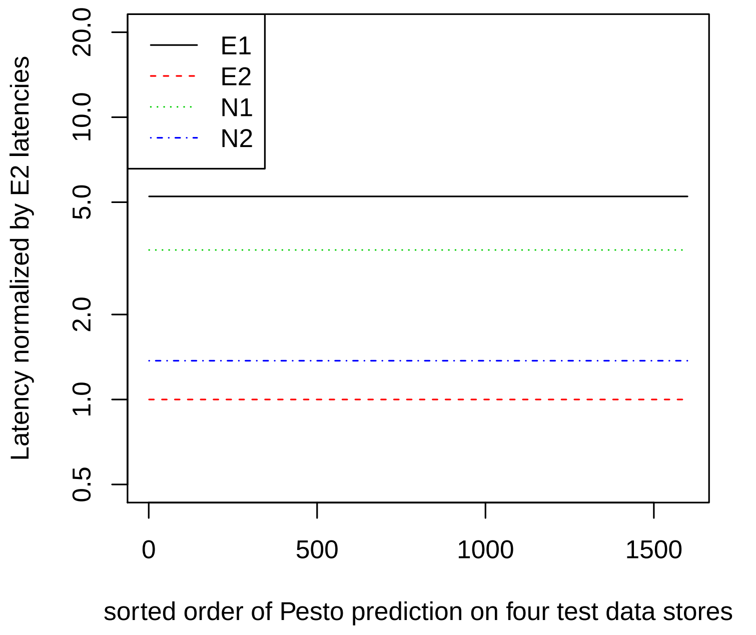
In this section, we evaluate the model's ability to handle heterogeneous data stores using Pesto [61] as the baseline. Basil is not compared here since it only generates a single model across the data stores and can not handle heterogeneity. Pesto injects a *4KB, random and read-only* workload into the data store to derive the *LQ-slope* for a given data store. Table 5.6 shows the Pesto LQ-slope of four test data stores. Since each data store has a fixed LQ-slope regardless of the workload, running same workloads on multiple data stores will always result in constant ratio equal to the LQ-slope ratio. Figure 5.5b plots this effect.

Figure 5.5a shows the latency normalized by E2 latencies from running our *test vector*. As expected, most of latencies observed in other data stores are larger than that of E2 (with the smallest LQ-slope hence the fastest) but surprisingly, there exist some corner cases where even the slowest E1 out performs E2. In fact, N1 seems to have the highest latency of all data stores on average even though its LQ-slope is lower than E1. Figure 5.4c shows that our model is able to capture this relative difference fairly accurately. For those corner cases where the model fails to capture the exact relative order of performance, the prediction intervals tend to be very large. Therefore, Romano will not make any predictions for those cases with high confidence indicating that actions should not be made based on the prediction values only.

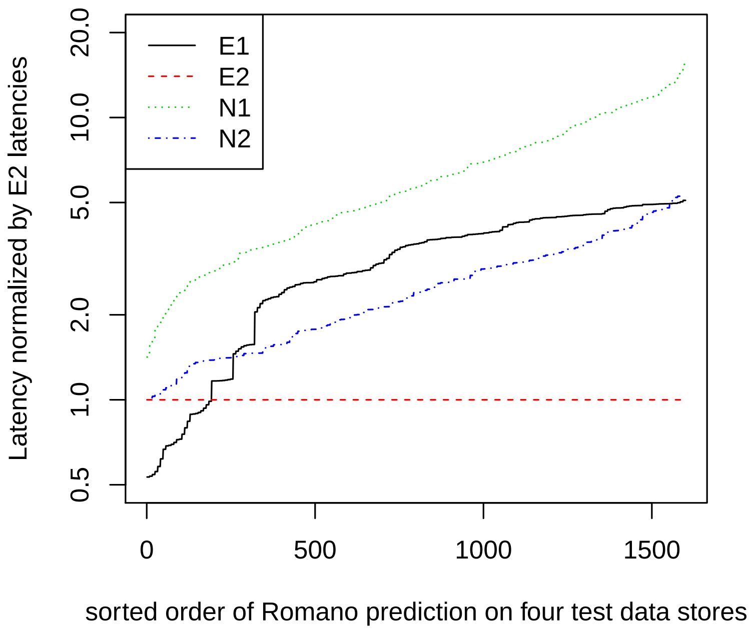
Figure 5.4a shows that the residual (ϵ) distribution of Pesto varies widely depending



(a) Measured Relative Latency



(b) Pesto Prediction - relative



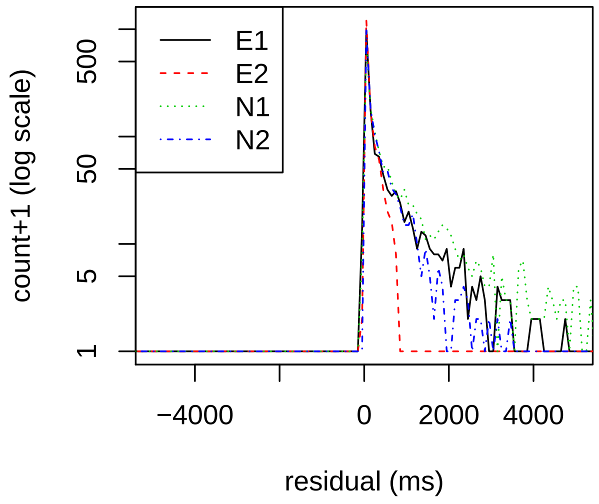
(c) Our Prediction - relative

Figure 5.3: Sorted normalized latency plot for four test data stores. It shows the relative latencies of E1, N1 and N2 compared to E2. The 1600 latencies measured from each data store were normalized by E2 and than sorted. The y-axis represents the normalized latency and the x-axis represents the sorted index. The actual measurements as well as predictions from Pesto and our model are plotted. Pesto assumes a linear performance difference between the workloads so there is only a linear difference between the data stores regardless of the workloads. For the actual measurement, all four lines cross each other, suggesting that data stores are seldom slower than the other data store for all workloads. While Romano does not capture this effect fully, it does capture the most distinct features of real measurements.

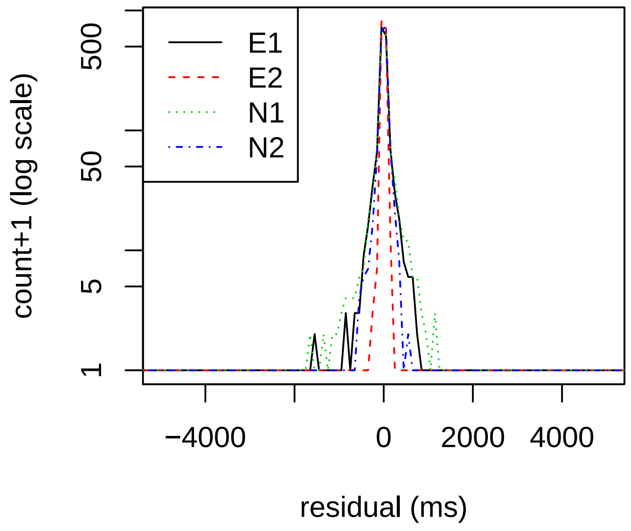
---

on the data store. The accuracy of Pesto on a single data store provides little information about how it might do on another data store. As the new data stores are introduced into the storage pool, the applicability of Pesto needs to be re-evaluated.

On the other hand, our model shows a much more even distribution across the data stores in Figure 5.4b. While there are some differences, prediction accuracy of our model is robust against the heterogeneity of data stores. Furthermore, the model's prediction interval further improves its robustness by generating larger interval for the data stores



(a) Pesto



(b) Proposed model

Figure 5.4: Distribution of residuals (ϵ) for Pesto and Romano. x-axis is the value of ϵ in ms. y-axis is number of occurrences+1. Addition of 1 was required to allow y-axis to be in log scale.

---

that are difficult to model. In short, proposed technique provides robust and adaptive models across the heterogeneous data stores with a single framework.

Chapter 6

Romano: IO Load Balancer

The cloud computing paradigm shift is built on virtualization. Most IT organizations and cloud service providers have deployed virtualized data centers in an effort to consolidate workloads, streamline management, reduce costs and increase utilization. Still, overall costs for *storage management* have remained stubbornly high. Over its lifetime, managing storage is often four times more expensive than its initial procurement [65]. The annualized total cost of storage for virtualized systems is often three times more than server hardware and seven times more than networking-related assets [66].

A key enabler for virtualization adoption is ability for a virtual machine (VM) to be efficiently migrated across compute hosts at run time [12, 67, 68]. Consequently, VM live migration has been utilized to perform active performance management in commercial products for some time [53]. Similarly, virtualization offers unprecedented dynamic control over storage resources, allowing both VMs and their associated virtual disks to be placed dynamically and migrated seamlessly around the physical infrastructure [13].

However, it is well-known that storage data associated with a VM is much harder to migrate compared to CPU and Memory state. Virtual disk migration time is typically a function of virtual disk size, size of the working set, source and destination storage state and capability, etc [13]. Migration can take anywhere from tens of minutes to a few hours depending on those parameters.

Given the high expense of data migration, intelligent and automatic storage performance management has been a relevant but difficult research problem [60, 61]. Virtualized environments can be extremely complex, with diverse and mixed workloads sharing a collection of heterogeneous storage devices. Such environments are also dynamic, as new devices, hardware upgrades, and other configuration changes are rolled out. Our goal of supporting heterogeneity is critical since data center operators want to keep the flexibility of buying the cheapest hardware available at any given time.

The key to automated management of storage lies in the ability to predict the performance of a workload on a given storage system. Without it, an experienced user must determine manually if migrating a virtual disk is beneficial, a complicated task. Previous work [60, 61] has relied on a heuristic model derived from empirical data.

When considering automated load balancing, the simplest approach would be to observe utilization of all data stores and offload a virtual disk from the most heavily loaded data store to the least heavily loaded data store.

There are several problems with this approach but the main problem is the unpredictability of the benefit of the moves. In fact, there exists no guarantee that the moves will even be beneficial. Storage systems do not react to different workloads in a homogeneous manner. A workload may stress one storage system significantly more than the other depending on the storage system configurations and optimizations.

The second problem is the workload interference. When multiple workloads are placed into a single storage systems, their effect on the storage system is not simply additive. While some workloads can be placed onto a single storage system and work with minimum interference, some experience huge performance hits.

Romano prediction model takes into account this heterogeneous response of storage systems and the interference between workloads improving the prediction accuracy by over 80%. Once these problems are recognized, it is obvious that the load balancing is no longer a bin packing problem as suggested by previous works [60, 61, 69]. Therefore, we introduce a new load balancing algorithm based on simulated annealing [70] that optimizes the overall performance in a stochastic manner. We show that Romano is capable of reducing the performance variance by 82% and reduce the maximum latency observed by 78%.

Romano makes following contributions.

Accuracy Romano residuals are reduced 80% on average compared to previous techniques. Furthermore, the residual is unbiased and does not propagate with recursive predictions.

Robustness Romano is able to specify the prediction interval which satisfies a specified confidence level. We show that the resulting prediction interval captures real performance with surprising accuracy.

Flexibility Different storage systems have different response characteristics to different workloads. Romano captures these differences by quantifying the effect of workload characteristics as well as the interactions of those characteristics on a given storage system.

Optimization Romano load-balancing algorithm avoids having system state stuck in a local optimum, instead identifying a global pseudo-optimal state through recursive prediction. At the same time, it also ensures that the most beneficial migrations are performed first to maximize the benefit.

We have prototyped Romano on VMware's ESXi and vCenter Server products. Minimal changes were made to the ESXi hypervisor to collect additional stats for vCenter where the core of Romano is implemented.

6.1 Romano Design

Romano is a generic framework that allows automated load balancing of the virtual disks in a heterogeneous storage environment. Specifically, Romano predicts the storage performance using statistical techniques allowing the systems to relocate the virtual disks without the human intervention.

Figure 6.1 shows an overview of Romano framework. Romano sits within the Resource Manager which maps physical storage resources to the virtual disks. Romano framework consists of *Romano Performance Model*, *Romano Aggregation Model* and a *Romano Load Balancer*. Romano Performance Model was described in Chapter 5. Romano Aggregation Model predicts the characteristics of multiple independent workloads when they are aggregated on a single data store. Romano Load Balancer balances the workloads across heterogeneous data stores such that the mean and maximum latency observed by the virtual disks are minimized.

These components will be described in detail next.

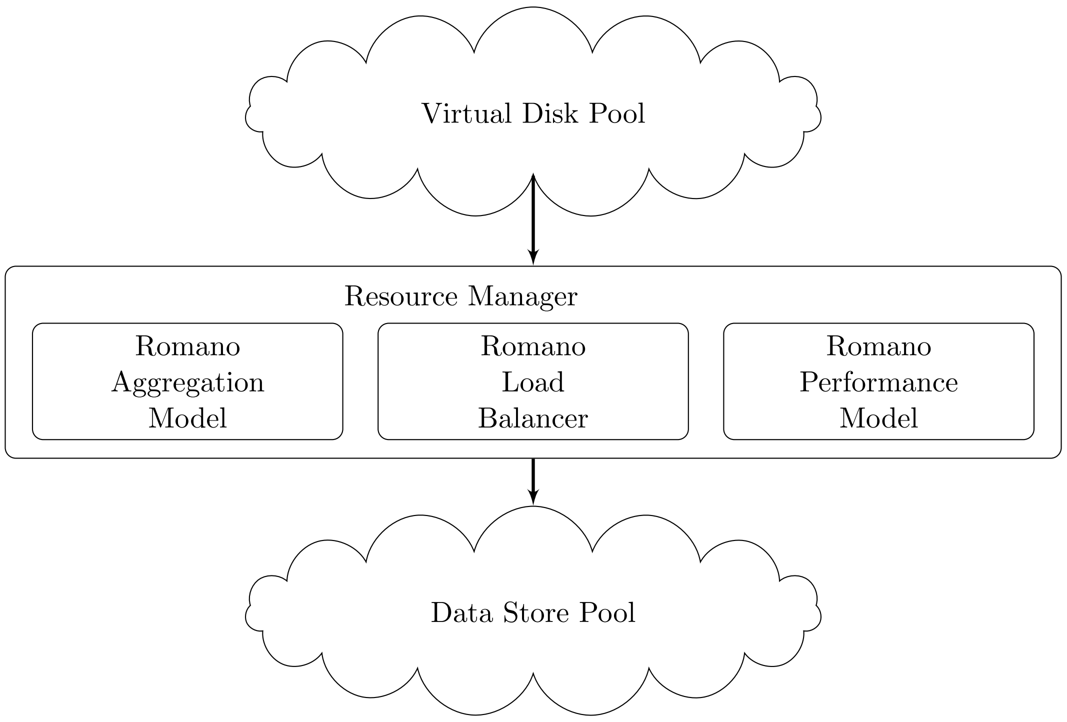


Figure 6.1: High level view of Romano. Romano maps virtual disks onto pool of data stores to satisfy the load balancing policy.

6.2 Romano Aggregation Model

Predicting the performance of a data store given a workload is not always enough. If the data store is already running different workloads, we need to figure out what the characteristics of aggregated workload will be.

In Basil, this is done by simply adding their workload parameter ω [61]. This made sense under their assumption that all data stores react to different workloads in a similar manner. Since ω represents the amount of work a data store has to process, aggregating workloads only involve addition of ω's.

Romano's workload model is a tuple of different combinations of characteristics. It makes more sense to aggregate each characteristic separately. In our closed loop assumption, aggregation of workloads does not affect the number of IOs each workload keeps in the system. Therefore we can safely assume that the *Outstanding IO*s (O) will simply be added.

However, *Size* (Z) and *Read Ratio* (R) can not be simply added together. It needs to be averaged based on ratio of number of requests seen by the two workloads that are being aggregated. Pesto weights each value by their O, assuming that higher O means higher rate of requests. However, O can be highly independent of arrival rate. For example, imagine a process that generates 5 IO requests and issues additional IO whenever a request completes. The arrival rate will only depend on the service rate of the requests and not on outstanding IOs which would be fixed at 5. For Romano this weight to be used for averaging is already available as the inverse of *LQ-slope*, which represents the throughput (T) from our performance model in Equation 5.14.

The *Random* (X) is tricky because the randomness of access cannot be defined per IO request and cannot be simply averaged.

Figure 6.2 shows the changes in the seek profile as you mix 0%, 50% and 100% random workloads on E1 data store. Our measurements show that the *LQ-slope* for these workloads are 0.476, 17.6 and 35.8 respectively. Therefore, the throughput (T), of these workloads are 2100.52, 56.77 and 27.93 respectively.

Note that *mseeks* represents seeks within a virtual disk while *lseeks* represents seeks across the virtual disks. If we define the *randomness* of the workloads as percentage of the seeks, the effect of aggregation is completely non-linear and hard to predict. However, we make the following observations.

1. Mixing sequential workload and non-sequential workloads results in very low *mseeks*.
2. The number of lseeks is determined by the lower throughput of two workloads.
3. The number of lseeks reduces the sequential and mseeks by the ratio of their average T.

More formally,

$$\overline{seq} = \frac{seq_1 T_1 * seq_2 T_2}{T_1 + T_2} \tag{6.1}$$

$$seq = \overline{seq} - (\overline{seq} * \%lseek) \tag{6.2}$$

$$\%lseek = \frac{\min(T_1, T_2)}{T_1 + T_2} * 2 * \overline{seq} \tag{6.3}$$

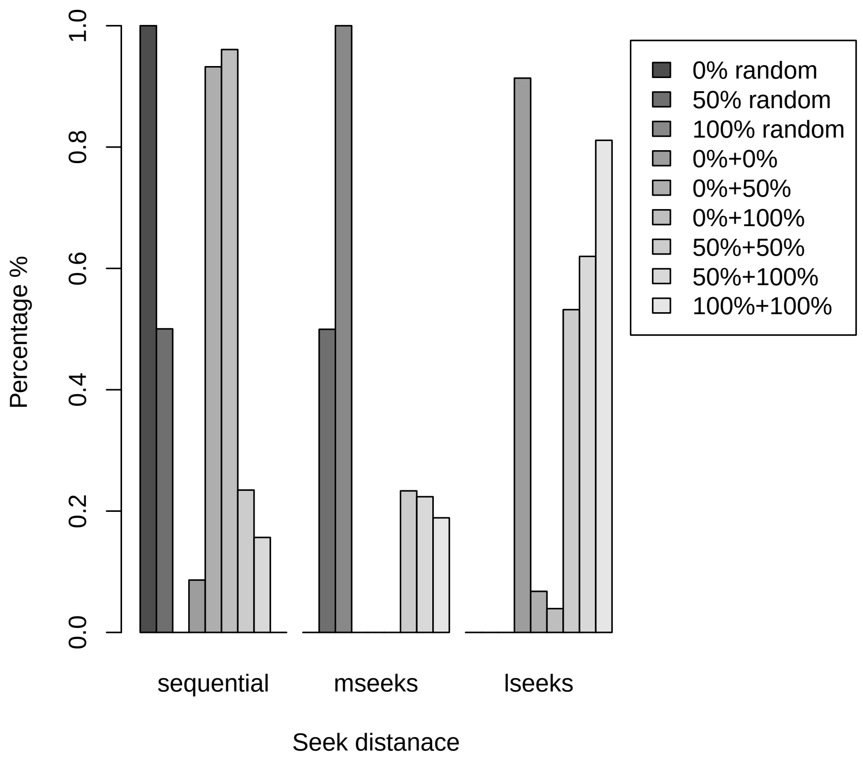


Figure 6.2: Seek profile of different workloads. Each color represents different mix of X for workload with $Z = 4KB$, $R = 100\%$ read, $O = 1$. *mseeks* represents seeks done within a virtual disk and *lseeks* represents seeks done across the virtual disks.

---

which can be solved for $X = 1 - seq$ to be:

$$R_{aggr} = f_X(X_1, X_2, T_1, T_2) = 1 - \frac{T_1 T_2 (1 - X_1)(1 - X_2)(1 + 2\min(T_1, T_2))}{T_1 + T_2}. \qquad (6.4)$$

Equation 6.1 is a simple throughput weighted average of the $sequential\%$ representing average $mseeks$. However, this does not account for additional seeks due to serving multiple virtual disks. Therefore Equation 6.2 subtracts the amount of $lseeks$ from the $sequential\%$. Note that the $\%lseek$ in the equation represents the percentage of *lseek*s. Intuitively, the storage system is typically serving the high throughput workload

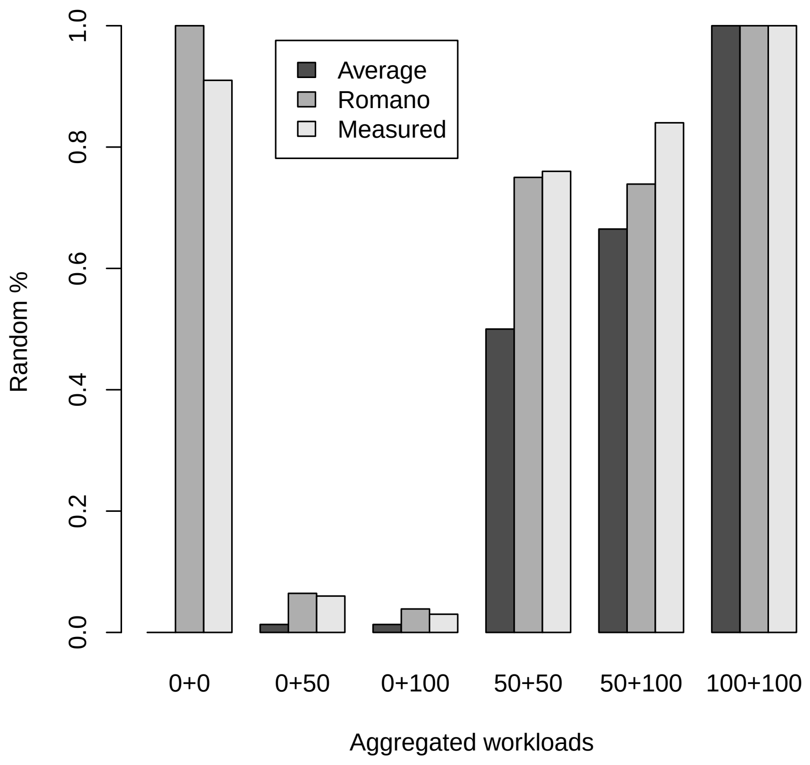


Figure 6.3: Comparison of f_R and T weighted average for the randomness of aggregated workloads. Each color represents different methods. Other characteristics are fixed at $S = 4KB$, $R = 100\%$ read, $O = 1$.

---

and occasionally seeks to low throughput workload which causes the $lseek$. Therefore, Equation 6.3 calculates the amount of $lseeks$ which is proportional to the ratio between the low throughput and the total throughput. This value is multiplied by $2 * \overline{seq}$ since almost every $lseek$ between the sequential workload is matched by a $lseek$ in the opposite direction to serve both virtual disks.

Figure 6.3 shows the result of f_X against simple T weighted average. The maximum and average error of doing T weighted average is 100% and 48% respectively. For f_X these values are 28% and 5% respectively. The maximum error in both cases are when aggregating two completely sequential workloads whose resulting X_{aggr} has inherently high variance.

Putting it together, *Romano Aggregation Model* is;

$$(X_{new}, R_{new}, Z_{new}, O_{new}) = (f_X(X_i, T_i), \frac{\sum_i R_i T_i}{\sum_i T_i}, \frac{\sum_i Z_i T_i}{\sum_i T_i}, \sum_i O_i) \tag{6.5}$$

where aggregating more than 2 workloads require f_X to be recursively applied to the workloads. In practice, this is naturally done since only a single virtual disk is migrated at a time.

6.3 Romano Load Balancer

The *Romano Load Balancer* is different from traditional load balancers [60, 61, 69] in two ways. Firstly, it optimizes the global state rather than a single storage migration. Secondly, it is proactive. The balancing mechanism does not wait until a over-utilization is detected. It continuously adjust the overall state whenever possible to ensure that systems are less likely to be over-utilized. This is made possible by Romano's accurate and robust modeling techniques.

6.3.1 Merit Metric

Our load balancing goal is to minimize the overall latency while reducing the maximum latency. To represent both aspect of our goal with a single metric, we define Romano load balancing metric of Merit (M) to be the form;

$$M = \left(\sum_i P_i^{\alpha} \right)^{\frac{1}{\alpha}} \tag{6.6}$$

where α controls the weight on the maximum latency. For example, if α is 1, M is simply the sum of all latencies. However, as α becomes larger, more weight is given to high latencies. Our experiment shows that 5 is a reasonable value for α. However, further exploration of α space is left for the future work.

Algorithm 1: Merit Update Function

```
updateMerit ($W_{cand}, S_{dest}, \Phi$) begin
    $S_{src} \leftarrow getS(W_{cand}, \Phi)$
    $W_{src} \leftarrow getW(S_{src}, \Phi)$
    $W_{dest} \leftarrow getW(S_{cand}, \Phi)$
    $W'_{src} \leftarrow$aggregate($W_{src} - W_{cand}$)
    $W'_{dest} \leftarrow$aggregate($W_{dest} + W_{cand}$)
    $P'_{src} \leftarrow$predict($S_{src}$, $W'_{src}$)
    $P'_{dest} \leftarrow$predict($S_{dest}$, $W'_{dest}$)
    return $merit(P'_{all})$
```

Algorithm 2: Simulated Anealing

```
minimizeMerit ($\Phi_i, M_i$) begin
    $\Phi \leftarrow \Phi_i$
    $M \leftarrow M_i$
    $\Phi_f \leftarrow \Phi_i$
    $M_f \leftarrow M_i$
    $K \leftarrow 0$
    while $K < K_{max}$ do
        $T \leftarrow$temperature($k/k_{max}$)
        $W_{cand} \leftarrow$randomChoose($W_{all}$)
        $S_{src} \leftarrow getS(W_{cand}, \Phi)$
        $S_{cand} \leftarrow$randomChoose($S_{all} - S_{src}$)
        $M' \leftarrow$updateMerit($W_{cand}, S_{cand}, \Phi$)
        if $G(M, M', T) > random()$ then
            $\Phi \leftarrow (W_{cand} \rightarrow S_{cand})$
            $M \leftarrow M'$
        if $M < M_f$ then
            $\Phi_f \leftarrow (W_{cand} \rightarrow S_{cand})$
            $M_f \leftarrow M'$
        $K \leftarrow K + 1$
    return $\Phi_f, M_f$
```

6.3.2 Load Balancing

Romano uses *simulated annealing* [70] to optimize the overall placement and then makes moves that result in maximum reduction of M. The reason for the first step is that greedy approaches used in previous works [60, 61, 69] could result in a local optimum

Algorithm 3: Romano Load Balancing Algorithm

```
loadBalance ($\Phi_i$) begin
    $M_i \leftarrow$merit($\Phi_i$); $Mlist \leftarrow \emptyset$
    ($\Phi_f, M_f$) $\leftarrow$minimizeMerit($\Phi_i, M_i$)
    foreach $W \in \Phi_f$ do
        if $getS(W, \Phi_f) \neq getS(W, \Phi_i)$ then
            $MoveCandidate \leftarrow W$
    while $MoveCandidate$ do
        foreach $W \in MoveCandidate$ do
            $M' \leftarrow$ updateMerit($W, getS(W, \Phi_f), \Phi_i$)
            append($Mlist, (M', W, getS(W, \Phi_f))$)
        sortDescendingByM($Mlist$)
        $(M, W, S) \leftarrow Mlist[0]$
        Move($W$, $S$)
        $M_i \leftarrow$ updateMerit($W, S, \Phi_i$)
        $\Phi_i \leftarrow (W \rightarrow S)$
        $MoveCandidate \leftarrow MoveCandidate - W$
```

that is far from the global optimum. The solution cannot be greedy since any optimal sub-placement needs to be reevaluated once a new workload is added due to the aggregation function. Adding a new data store is worse since every data store has its own performance model. While the simulated annealing process does not necessarily result in optimal placements, it is guaranteed to be better than finding a local optimal if the algorithm can remember the best state it has seen so far [71].

However, the resulting state may require too many workloads to be migrated. Therefore, once the target state is defined through simulated annealing, we use a greedy method to get there. This way, we can make moves that result in the largest reduction of M while always moving towards the optimal state. Of course, there may be a move that results in an even larger reduction but the move may prohibit the possibility of further reduction of M.

Algorithm 3 outlines the Romano load balancer. In this algorithm, Φ represents the current mapping of workloads to data stores which is essentially current state of the system. It requires two subcomponents, Alogrithm 1 and Algorithm2.

Algorithm 1 simply moves a virtual disk, W_{cand}, to a datastore, S_{dest} and recalculates the system's *merit* based on the performance prediction of the source datastore, S_{src}, and the target datastore, S_{dest}.

Algorithm 2 is the simulated annealing process where given a state and its merit, it will return a pseudo-optimal state and its merit which should be close to the global minima. This has worked surprisingly well for this application as it will be shown in the results. The *temperature* function and G functions are taken from the earlier works on simulated annealing [70]. The *temperature* function simply divides k/k_{max} by 1.2 every iteration. G function returns 1 if current merit is lower than previous merit so that the move is always accepted, but returns $e^{(M-M')/T}$ when current merit value is higher. Therefore, there exists a finite chance of accepting a move even when the merit increases. This probability decreases fast with the number of iterations and the difference between the new and old merit values. With every iteration, the function chooses a random workload to move and a random destination data store. New merit is calculated and the move is either rejected or accepted based on the value returned by G and a random number between 0 and 1. All our experiments required less than 200 iterations to complete. The convergence rate of simulated annealing is difficult to predict without any structural information of the state space. However, since we do not need the absolute minimum M, we can simply fix the maximum number of iterations to a reasonable number and choose the best state found.

The *loadBalance* function takes the current state (Φ) and calls on *minimizeMerit* function to determine the pseudo-optimal mapping of W on S. Once the mapping is determined, it uses a greedy approach to determine which moves to execute. *update-Merit* function is called on all possible moves to determine the next move. Note that *updateMerit* function executes in $O(1)$. The process is repeated until the pseudo-optimal mapping is reached. $getS(W, \Phi)$ represents the data store on which W is mapped in current State Φ while $getW(S, \Phi)$ represents all the workloads mapped onto data store S in state Φ.

updateMerit function takes the current system state (all Workload mappings and their performance) and calculates new *Merit* value. Note that our goal is to minimize the Merit value. The *aggregate* function is *Romano Aggregation Model* and *predict* function is *Romano Performance Model.* Therefore, *updateMerit* function is given a workload candidate to be migrated and destination data store. New aggregated workloads are

Workload	*IOSize*	*Read%*	*Random%*
Web File Server	4KB	95%	75%
Web File Server	8KB	95%	75%
Web File Server	64KB	95%	75%
Support DB	1MB	100%	100%
Media Streaming	64KB	98%	0%
SQL Server Log	64KB	0%	0%
Web Server Log	8KB	0%	100%
OLTP DB	8KB	70%	100%
Exchange Server	4KB	67%	100%
Workstation	8KB	80%	80%
VOD	512KB	100%	100%

Table 6.1: Workload characteristics suggested by Wang [1]. These workloads are used to ensure repeatable experiments.

calculated and their performances are predicted. Given the new performances of source and destination data stores, global merit is recalculated.

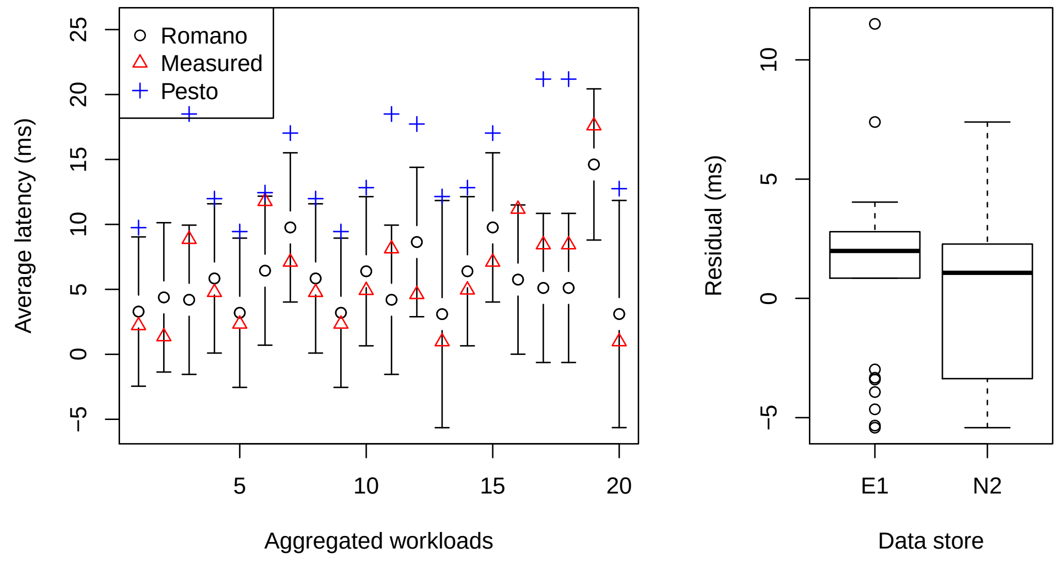
In practice, it is unlikely that all the moves will be carried out due to the cost involved with the migration. Furthermore, the workload themselves are going to change over time. Therefore, Algorithm 3 will periodically rerun *minimizeMerit* function and continually transition the system towards the newly found pseudo-optimal state without ever actually getting there.

6.4 Results

6.4.1 Workload Aggregation

To test the effect of workload aggregation, we need to able to generate a repeatable workload. We use the workload settings shown in Table 6.1 to simulate the real workloads. Previous works have shown that synthetic workloads do a good job of representing real workload's performance characteristics when the parameters are extracted from the real workloads [59, 2].

All possible combinations of two workloads were chosen from the 11 workloads shown in Table 6.1 resulting in 55 combinations. In Romano workload characteristics space, aggregating 2 workloads and 3 workloads are the same since the aggregation process



(a) Sample of latency prediction

(b) Residuals box plot

Figure 6.4: Result of workload aggregation. Two Workloads from Table 6.1 were placed randomly across two data stores, E1 and N2. Figure 6.4a shows 20 randomly sampled results from 110 placements on both data stores (55 each). Figure 6.4b shows the residual of all 55 placements on E1 and N2.

---

is commutative and associative. O was randomly assigned from 1 to 3. The test was repeated on two data stores for which the Romano model performed worst (see Figure 5.4b).

The workload aggregation method used in Pesto [61] is to average their workload metric ω weighted by OIO. To isolate the effect of workload aggregation, we average individual workload characteristics weighted by O and use *Romano Performance Model* to predict the latency.

The result is shown in Figure 6.4. Figure 6.4a shows the latency from 20 randomly selected experiments as well as the prediction interval of Romano. Only in 1 case, Romano failed to capture the latency within its prediction interval. In fact, only 3 out of 110 cases (2.7%) showed Romano failing to capture the measure latency. This is better performance than specified 5% confidence level. We believe that this is due to the unbiased nature of Romano residual shown in Figure 5.4b. As the workloads are aggregated, the residuals (ϵ) of *LQ-slope* prediction, used for workload aggregation, are more likely to cancel each other out.

The accuracy of Pesto's approach of O weighted average does not seem too bad at first but there are some cases where the prediction won't even register on the Figure 6.4a (workload 2 and 16). This is especially true if the workloads running together are highly skewed such that one workload has much higher throughput than the other. This verifies our assumption that the workload should be aggregated based on its throughput rather than O. his is especially true if the workloads running together are highly skewed such that one workload has much higher throughput than the other. This verifies our assumption that the workload should be aggregated based on its throughput rather than O. Figure 6.4a also shows that while Pesto consistently overpredicts the aggregated latency, Romano residuals are more unbiased shown in Figure 6.4b.

Another interesting result was that all 3 cases where Romano failed were observed on E1 data store. This is as expected from the amount of outliers shown in Figure 6.4b. We believe that amount of uncertainty introduced by workload aggregation is different on each data store. We leave the further investigation to future work.

6.4.2 Load Balancing Using Romano

In this section we randomly placed 8-14 workloads from Table 6.1 allowing repetition on 4 data stores. Romano load balancing algorithm from Algorithm 1 was ran to optimize the data placement by moving a single virtual disk at a time. The experiment was repeated 50 times.

Figure 6.3 shows the aggregated result of load balancing using Basil and Romano. It should be noted that the load balancing algorithm deployed by Basil and Pesto are the same. Both Basil and Romano does a good job of reducing the average latency by 47% and 52% respectively. However, the variance is reduced by 82% by Romano while Pesto reduces it by 37%. Most importantly, the maximum latency observed was reduced by 78% with Romano while only 21% by Pesto.

There are two factors which result in improved performance by Romano. The first is the more accurate modeling. Both Pesto and Basil assume that there exists only a constant performance difference between different data stores for any given workload. Therefore, it has tendency to move virtual disks to a data store that is more powerful regardless of the workload. Furthermore, their aggregation model is based on the outstanding IOs only. This allows workload that do not performance well together to be placed on a single data store. The result of these inaccuracies in the model forces the load balancing algorithm to place more workloads on the powerful data stores. In fact 51% of all the moves by Pesto were made to E2 data store compared to 28% for Romano. Figure 6.5b shows that resulting E1 and N1 latencies are smaller than that of Romano while E2 and N2 exhibit large latencies that we would like to avoid. In fact, Basil does not place any workload on E1 90% of the time even though E1 actually has lower latency for most workloads than N1 as shown in Figure 5.5a.

The second factor is the load balancing algorithm itself. Basil and Pesto uses a greedy approach [60] which terminates the algorithm once no beneficial moves are found. Romano uses simulated annealing to find the pseudo optimal placement first and than uses greedy approach to get to the optimal placement. This allows Romano to avoid moves that may result in maximum benefit but prohibits any further moves that may provide additional benefits.

One limitation of this approach is that it does require more moves to be made. Figure 6.4 shows the number of moves required for Basil and Romano. On average Basil requires 2.2 moves where as Romano required 2.6. Storage migration is expensive and should avoided if possible. However, we show that even if we limit the maximum number of moves to 2, Romano still out performs Basil as shown in Figure 6.5. This is due to the greedy approach in which Romano chooses moves once the pseudo-optimal placement is identified.

Figure 6.6 shows an example of latency changes for a single test case. It is shown that most critical balancing is done within the first couple of moves.

6.5 Conclusion

We have presented Romano, a load balancing framework for virtual disks on heterogeneous storage systems. The kernel of Romano is a performance predictor given a set of parameters that characterize the workloads and the storage devices. We have shown that Romano can outperform previous systems with same goals by up to 80% in prediction accuracy. This increased accuracy results in 78% reduction in maximum latency observed and 82% reduction in variance while load balancing.

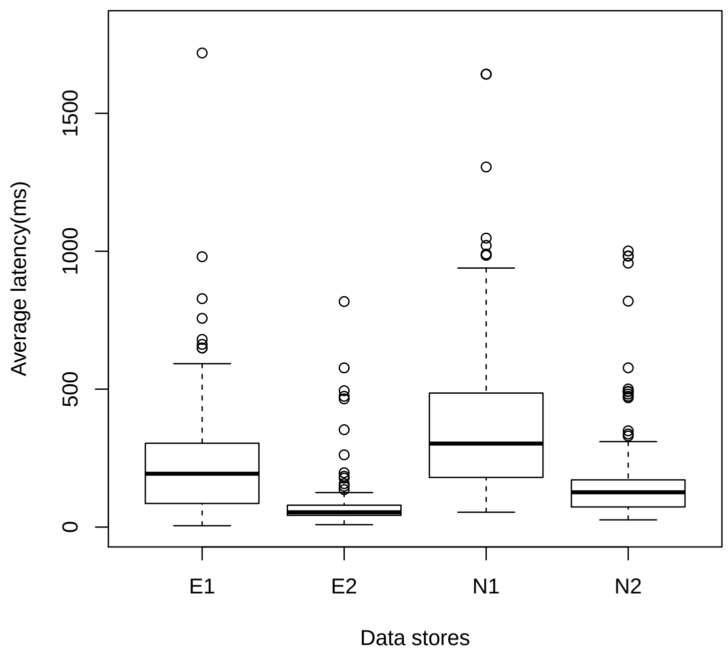
At a deeper level, Romano contributes a recognition that there is inherent noise in storage system performance that is not easily dealt with. Romano is capable of capturing this noise within the prediction interval. As more data is gathered there is higher confidence that the prediction interval will contain the measured latency.

Another key contribution of Romano is quantification of effect of various workload characteristics and their interaction on heterogeneous storage devices. We have shown that the performance differences between data stores cannot be described with a single number. More specifically, we have shown that the storage performance must be described in terms of the workload.

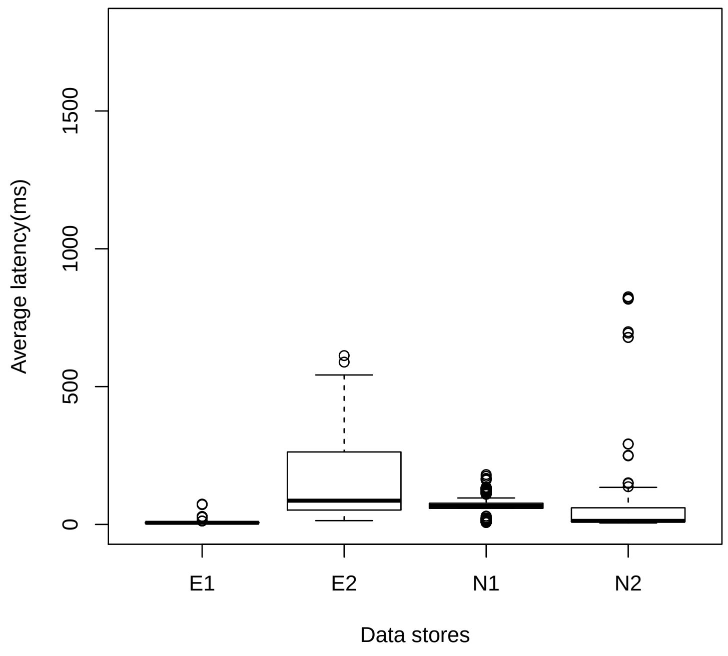
The last contribution follows the second contribution. Since the performance of storage systems effectively change with the workload, the load balancing problem is no longer a bin-packing problem as previously believed. Therefore we present a probabilistic approach to finding a pseudo-optimal mapping of workloads to the data stores. It is important to mention that probabilistic approach is only used to find the final state of the system and the moves are actually made greedily to speed up the convergence and minimize the number of moves that has to be made.

We believe that Romano not only provides means for more efficient load balancing but also in many other areas such as QoS management, power management and performance tunning in tiered storage.

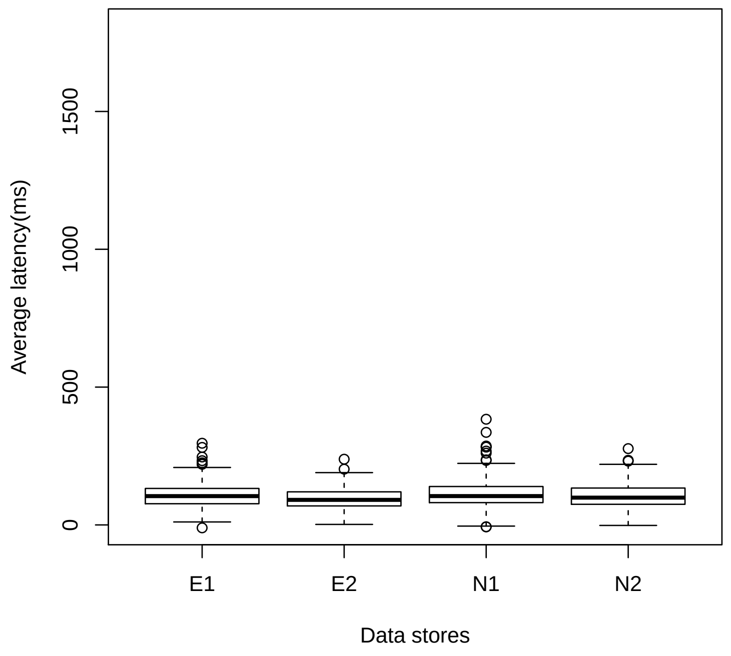
Whereas Romano raises the bar on the accuracy of practical modeling techniques, there remain ample opportunities for improvements in future work. Better interference modeling between workloads when they are placed on same data store is one such area where we are making good progress. Another work in progress is to come up with a better set of workload characteristics such that the workload description can be more complete.



(a) Initial data stores latencies.

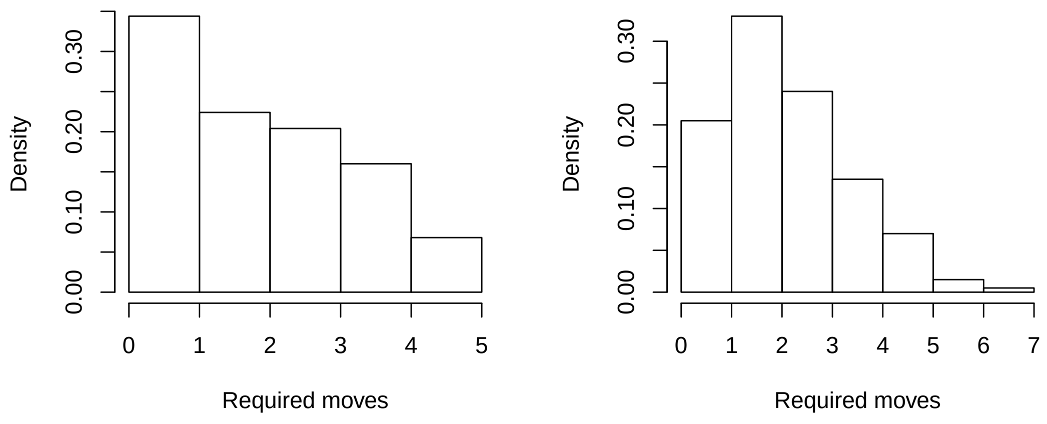


(b) Result of Basil load balancing



(c) Result of Romano load balancing.

Figure 6.3: Boxplot of average data store latencies on 50 different test cases. Each test randomly placed 8-14 workloads from Table 6.1 (allowing repetition) on 4 data stores. Basil's greedy approach is compared with Romano's random optimization technique.



(a) Basil

(b) Romano

Figure 6.4: The distribution of number of moves resulting from Basil and Romano.

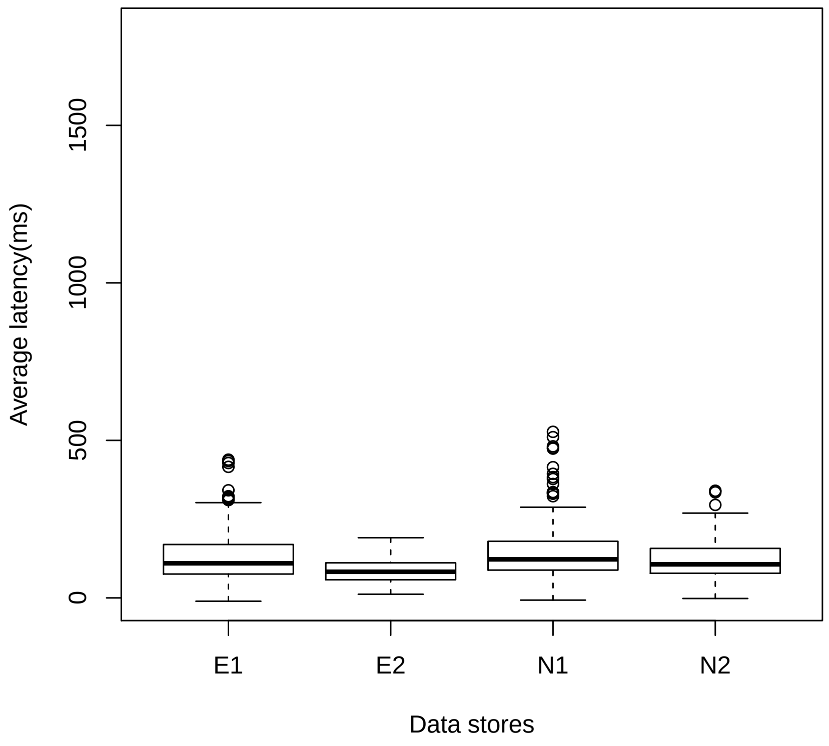


Figure 6.5: Data store latencies after maximum of 2 moves with Romano.

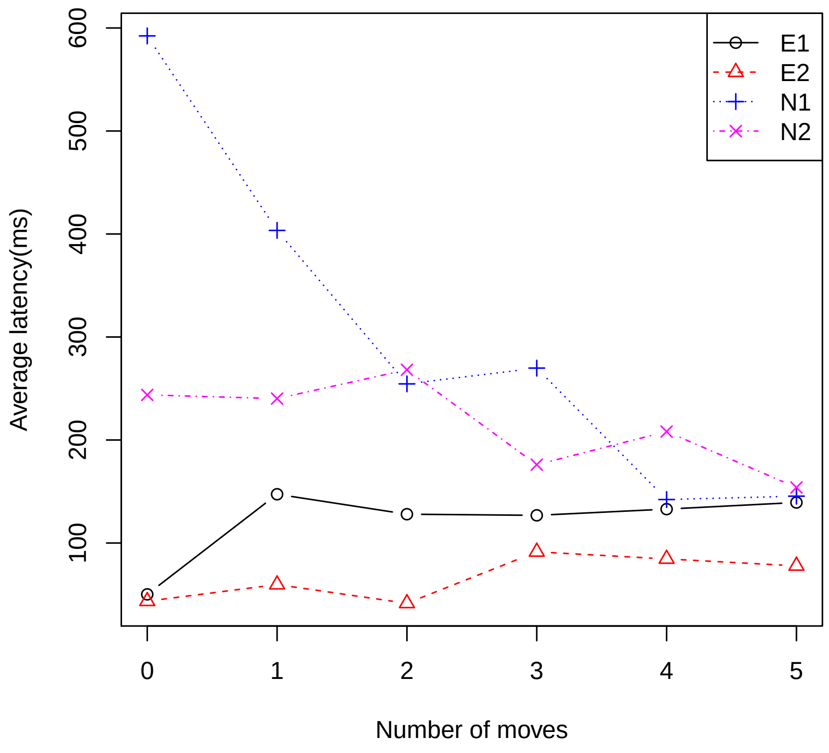


Figure 6.6: Typical Romano load balancing process with large number of moves.

Chapter 7

Backup Workload and Data Deduplication

As more and more backup systems opt to take advantage of data deduplication techniques, the variability in performance is becoming an issue. The cause of this variation can be categorized into two factors, *systemic variation* and *input variation*. Systemic variation is caused by the use of different algorithms and techniques as well as the underlying hardware deployed in the deduplication systems. The input variation is caused by different characteristics of the input datasets. The systemic variation is critical from the deduplication system designers' and vendors' perspectives since it allows them to compare two systems directly. However, the input variation is typically more critical from the customer's perspective when evaluating the potential benefits of data deduplication for different types of datasets.

Current metrics for characterizing the datasets for data deduplication are overly simplified and often inaccurate. For example, the compression ratio (CR) is typically estimated using the *average data change rate* ($\overline{dcr}$), which is the percentage of data change. Let R be the required retention period. Assuming a full backup per unit time, the compression ratio is simply:

$$CR = \frac{\text{Compressed Size}}{\text{Uncompressed Size}} = \frac{1 + (R-1) \cdot \overline{dcr}}{R}. \tag{7.1}$$

This model provides a simple estimation of the compression ratio but it can also be very inaccurate. We observed over 35% error using the $\overline{dcr}$ to characterize one of our datasets. One of our main contributions is providing a new set of metrics and models that allow much more accurate compression performance predictions to be made. In five out of six cases we test, the error reduction was over 50% when compared with the $\overline{dcr}$ method.

The main performance metrics for data deduplication systems are the compression ratio and the read/write throughput. While the latency could also be an issue for virtual machines, it can be mostly masked by high level caching and prefetching mechanisms. Unless specified otherwise, we use the term *performance* to represent both the throughput and compression together.

The throughput of the system is heavily dependent on both system and input factors. Thus, it is difficult to determine if one dataset outperforms the other in terms of throughput. However, there are features of datasets that are likely to benefit throughput in most systems. One obvious one is the compression ratio itself. A highly compressible dataset has lower IO requirements which in turn improves the throughput in most systems. The compression ratio is the most influential factor in determining the throughput. However, there are also other factors, such as spatial locality of the data segments and bursty arrival rates, which we examine in this paper.

Our main contribution in this paper is to provide a framework to characterize the input dataset for data deduplication systems. We 1) show that different datasets behave with a unique pattern that is quantifiable. Furthermore, we 2) provide analysis on how this pattern affect the deduplication system performance. As part of the framework, we 3) provide classification of segment types and their relations. We further provide parameters to show how the *composition* of these segment types change over time.

7.1 System Components

The input data itself is another system component that affects the performance. To understand how the dataset affects the performance, it is described in terms of how the data components can be separated once it is deduplicated. The segments, regardless of how they are defined, are the smallest units of data in data deduplication. Therefore, we define data as a sequence of segments rather than contiguous bits or characters. In

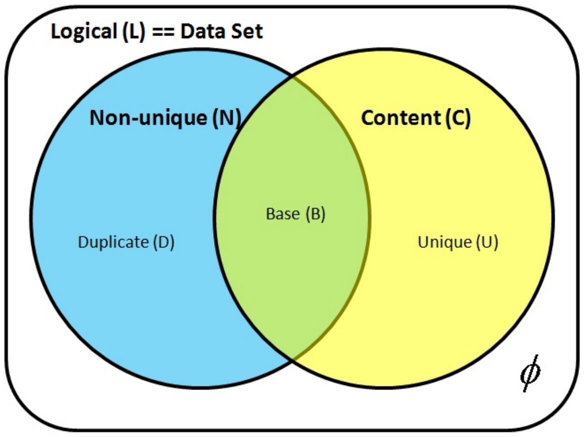


Figure 7.1: Venn Diagram of Segment Sets. The *logical segments* which is equivalent to the *dataset*, can be grouped into two separate sets N and C. The elements of N are segments whose content occurs more than once in the dataset while the elements of C are segments that must be stored in order to ensure no content information is lost from the dataset.

---

this work, we show that the distribution of these different segment types completely describes how the dataset affect the deduplication performance.

7.1.1 Queuing Model

The system can be viewed as simple open queuing system with three servers as seen in Chapter 2. We make very little assumptions of particular algorithms deployed. However, the functionality of each stage we have described in Chapter 2 exists in all conventional deduplication systems. In this system, the arrival rate of segments at the *content dictionary* is assumed to be exponentially distributed with the mean of average service rate at the *content generator*. The probability of the arriving segments to be found in the dictionary is equal to the compression ratio in which case it can by pass the *content store*. If we assume that each outcome is independent, the arrival rate at content store is also exponentially distributed. However, we show that this is not true and is one of the factors that affect the throughput of the system.

7.1.2 Segment Classification

We first define data segment as a tuple of *offset* and *size* within a dataset. Therefore every segment is uniquely identifiable regardless of its content. The only constraint under this definition is that the size of the data segment must be less than or equal to the size of the dataset. Also, the content of the data segment must exist within the dataset. There are no other restrictions to what may be defined as a segment. A dataset is then a mere concatenated sequence of data segments.

We begin classifying the data segments by defining the *logical segments* which is the set of all segments from which you can reconstruct the original dataset through reordering only. Under this definition, a *dataset* is nothing but an ordered list of logical segments. We also define *unique segments* and *non-unique segments.* The unique segments are a set of segments whose content is unique across the logical segments. Obviously the non-unique segments are the segments whose content is repetitive across the dataset. These non-unique segments have a unique property called the *degree of repetition* which represents number of times a particular content exist within a dataset.

From the perspective of the data deduplication system, the segments are really divided into segments whose contents must be stored and segments which can be represented only as a reference to an already stored segment. We further classify segments as either the *content segments* or the *duplicate segments* based on whether the content of the segment is required to reconstruct the logical segments. The content segments can be duplicated and reordered to construct the original dataset given the ordering information of contents. The duplicate segments are simply redundant contents whose only new information is the order of the content.

It is obvious from the two classifications that the duplicate segments must be the non-unique segments while the unique segments are content segments. This is shown in the Figure 7.1. As shown in the figure, there exists an intersection of non-unique segments and the content segments that are called the *base segments.* The base segments are minimum set of segments whose contents that must be stored to allow the deduplication system to reconstruct all the non-unique segments.

Therefore, the base segments, the unique segments and the duplicate segments are three sets of segments that are disjoint and whose union is the logical segments. A segment at any given time must be a member of one and only one of these sets.

Dataset	Number of Segments	Compression Ratio	Throughput (MB/s)
exchange	1,970,836	0.294	44.72
exchange_is	1,960,555	0.377	42.60
exchange_mb	1,951,271	0.973	27.20
fredp4	1,892,285	0.502	39.78
fredvar	1,922,420	0.209	48.36
workstation	1,966,577	0.403	38.92

Table 7.1: Datasets. Six datasets with the size of the logical segment set and two performance metrics of interest.

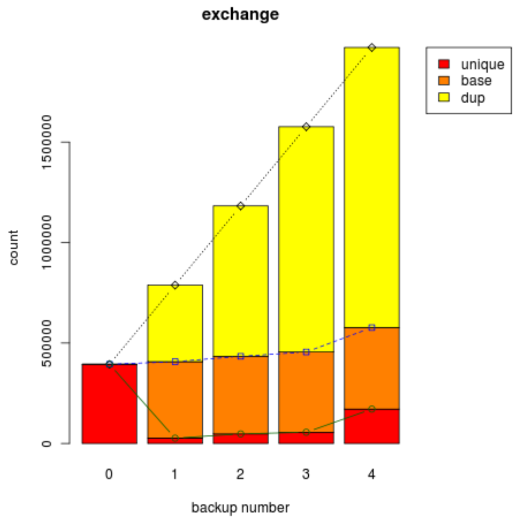
Parameters	Value	Parameters	Value
Content Generation Method	Content Based	Minimum Segment Size	4KB
Average Segment Size	8KB	Maximum Segment Size	16KB
Storage Maximum Throughput (sequential)	50MB/s	Maximum Network Throughput	100MB/s
Content Dictionary Cache Size	100,000 entries	Hash Function for Segment ID	SHA1 (160 bits)
# of Bloom Filter Hash Functions	4	Bloom Filter Size	1,000,000 entries

Table 7.2: Experimental deduplication parameters.

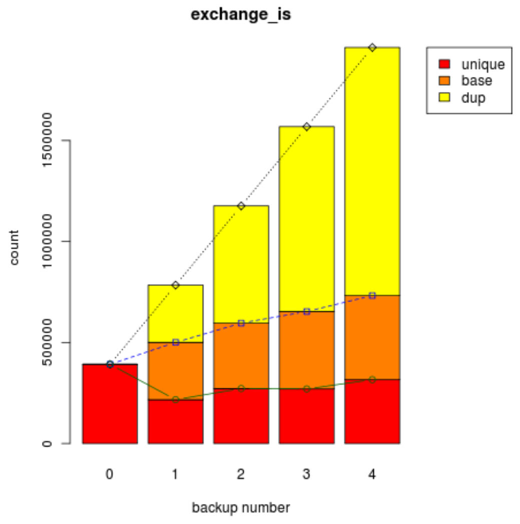
We also define a relational parameter between these sets.

Definition 1 (*scr*) *Segment compression ratio.* $scr = |C|/|L|$ *where* $L = \{logical\ segments\}$ *and* $C = \{content\ segments\}$.

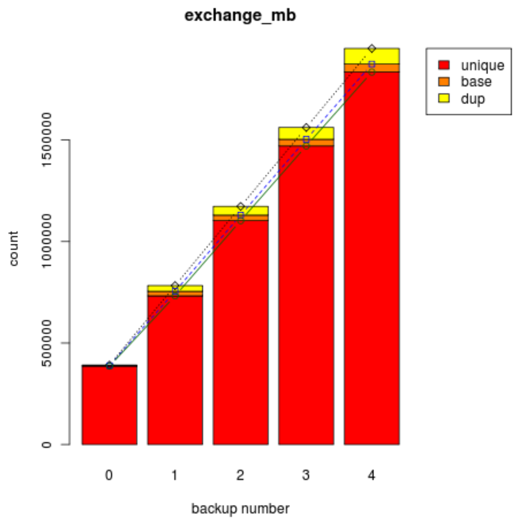
The *scr* obviously represents how much of the data is actually redundant in terms of segment count. Typical data deduplication systems have some bounded segment size which ensures that *scr* closely reflects the actual data reduction.



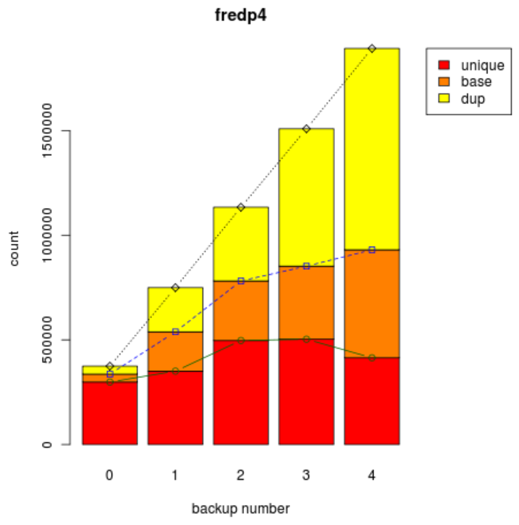
(a) exchange



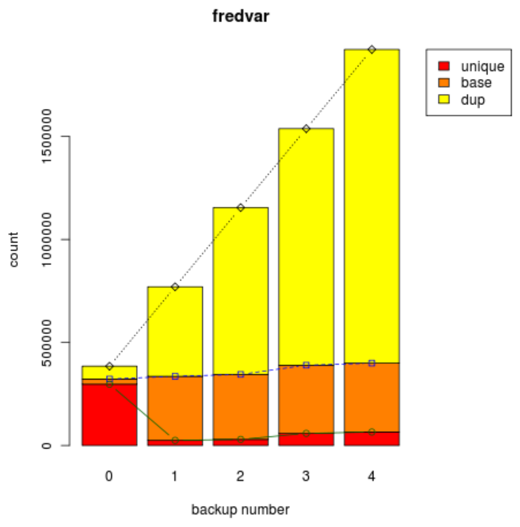
(b) exchange_is



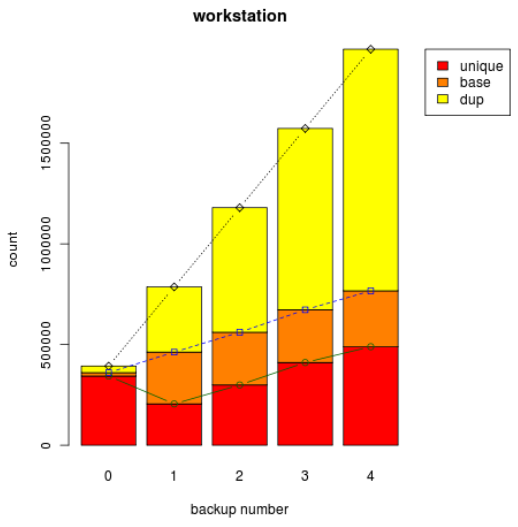
(c) exchange_mb



(d) fredp4



(e) fredvar



(f) workstation

Figure 7.0: Composition of segments for different datasets. Graphs show how the composition of segments changes over time as weekly full backups are deduplicated.

7.2 Description of the Experimental Datasets

Six different datasets shown in Table 7.1 were used in our evaluation. Due to privacy concerns, only the SHA1, size and the backup order of the segments were made available.[1] We refer to this data as *segment trace*.

The *exchange*, *exchange_is* and *exchange_mb* datasets are all Exchange server data encoded in different ways. The *fredp4* dataset contain a revision control system data and *fredvar* contain data from /var directory in the same machine. The *workstation* dataset contains data from the home directories of several users.

The segment trace is not of the entire original dataset. Unfortunately, the data presented to us were of the data that is truncated from a larger dataset. Although a truncated dataset is used, it does not effect the methodology presented in this paper.

[1] The actual data provided also contained compressed size of the segment, stream offset which represents where the segment is located in the original dataset. The compressed size is unused since the local compression of segments is not of interest in this study and stream offset is redundant information which can be deduced from the order and the size of the segments.

This is because we characterize the datasets based on how deduplication system processes them. There are only a specific range of data characteristics from deduplication system's point of view. And these truncated datasets are used to show where in this range a particular dataset may lay. We could easily have done same analysis with the synthetic traces since their characteristics cannot lay outside this range regardless of how we generate the trace.

The system parameters for data deduplication system is described in the Table 7.2. The cache size for the *content dictionary* is kept reasonably small to capture the effect of cache misses. Since the maximum raw throughput of the system is IO limited at 50MB/s, you can see that the throughput of different datasets vary from being close to the maximum throughput to about 50% of that throughput.

As mentioned earlier, there is a strong linear relationship between the compression ratio and the throughput. The sample correlation coefficient, $r_{ct} = -0.98$ (where $c = \textit{compression ratio}$ and $t = \textit{throughput}$), suggest that higher compression (lower compression ratio) results in higher throughput. While it may seem that compression ratio is the dominant factor in determining the throughput since the two are so highly correlated, the system for which the throughput numbers are reported are *in-line* deduplication systems where all segments must be compared against the content dictionary before it can be stored on the disk. Therefore, the elimination of duplicate segments not only results in dictionary update but also additional disk writes. In systems where data may already all be written to the disk before being looked up in dictionary may not perform in exactly the same manner. Therefore, we concentrate more on dictionary access pattern to analyze potential throughput concerns.

The segment trace contains 5 weekly full backup information of 6 different datasets and 5GB of each full backup. Therefore, all data presented in Table 7.1 are of the same size at 25GB. The variation in number of segments is due to variation in average segment size. Compression ratio provided in the table is compression due to data deduplication process only and does not include additional compression provided by local compression of segments using traditional data compression tools.

7.3 Data Characterization

An obvious and perhaps the most important data characteristic for data deduplication is amount of redundancy. Two datasets of equal redundancy would yield in similar compression assuming the content generation was done in a reasonable manner. However, they could perform very differently depending on the other aspects such as locality of segments and fragmentation due to elimination of duplicate segments. The difficulty in characterizing workload of data deduplication is that the characteristics is actually dependent on the contents. While access patterns and physical attributes such as size of files tends to follow a well known distributions in a large scale [72, 73], contents themselves are random in nature due to human factor and different encoding deployed by different applications.

7.3.1 Composition of Segments

Figure 7.0 shows how the composition of segments, as defined in Figure 7.1, changes over time for each dataset at a full backup granularity. The composition of the segments is quite different for all datasets. In the figure *backup number* represents accumulated number of full backups stored in the system. We make a few observations from the composition.

Observation 1 *Amount of duplication found within a single full backup is negligible compared to the duplication found across the full backups.*

This is an observation also made by other deduplication works on backup data [24]. This observation allows us to look at the changes in the composition of segments as results of deduplicating a full backup instance against other instances of full backup. While non-zero unique and base segments at backup number 0 in Figure 7.2d, Figure 7.1e and Figure 7.1f indicate that there exists some inter-duplication within a backup, the amount is negligible and we ignore its effect in this paper.

Observation 2 *For the base segments there are two distinctive case where number of base segments increase with number of fulls as in Figure 7.2b and Figure 7.2d and case where number of base segments stay steady as shown in Figure 7.2a, Figure 7.1e and Figure 7.1f.*

This observation can be made more pronounced in Figure 7.1a.

The number of base segments can only increase in the absence of the deletion. Once a segment becomes a base segment, it cannot become any other type of segment. Therefore, the number of base segment can only stay steady if there is no or very little amount of new base segments are created. We can conclude that in this kind of dataset, the unique segments stays unique segments while the same set of base segments keeps getting duplicated.

Conversely, the increase in number of base segments mean that the unique segments of previous backup number is getting converted to the base segments. This happens only when a newly generate data at backup number i still exists at backup number $i+1$. We assume that no base segments are generated within a full backup instance based on the observation 1. We call this rate of conversion *base generation rate*(bgr) which is bounded by the number of unique segments in previous backup number and is larger than or equal to 0.

We formally define bgr below.

Definition 2 (bgr) *Rate at which unique segments are converted to base segments.* $bgr_i = \frac{|B_{i+1}|-|B_i|}{|U_i|}$ *where the subscripts represent the backup number. We define bgr to be average value of* bgr_i, $bgr = \frac{\sum bgr_i}{max(backupnumber)}$.

Observation 3 *The number of unique segments either increase steadily as shown in Figure 7.2c and Figure 7.1f or stay relatively steady as in the rest of the figures in Figure 7.0. While the number of unique segments can decrease, it does not happen very often.*

The number of unique segments, $|U|$, at backup number i is increase by amount of unique segments generated by that particular full backup and is decreased by amount of unique segments at backup number $i-1$ converted to base segments[2] at backup number i. Since the amount of decrease in $|U_{i+1}|$ is $|U_i| \times bgr_i$, we also need to define a parameter to represent the increase in $|U|$, *unique segment ratio*(usr). This ratio does not represent the changes in the number of unique segments, $|U|$, since there is also conversion of segments from unique to base.

[2] Conversion to duplicate segments is also possible but is ignored due to observation 1.

Definition 3 (usr) *Relative amount of unique segments within a full backup.* $usr_i = \frac{|U_{i+1}|-|U_i|(1-bgr_i)}{|L_{i+1}|}$. *We define usr to be average value of* usr_i, $usr = \frac{\sum usr_i}{max(backupnumber)}$.

From the definition of *usr* it is obvious that we cannot simply determine the rate of unique segment generation from looking at the changes in $|U|$. However, since it is possible to determine the *bgr* from also looking at the changes in number of base segments, $|B|$ which in turn allow us to calculate *usr*. Since increase in $|B|$ is only possible due to *bgr*, if increase is very small, we can assume that $bgr \simeq 0$. This allows us to think that $usr_i = U_{i+1} - U_i$ which means that unique segments in backup number i no longer exists in backup number $i+1$ or else they would have become base segments. This pattern is most pronounced in Figure 7.1f. Intuitively, the pattern suggests the existence of a small and heavily updated *working set* within the file system as suggested by [74]. Similar analysis could applied to Figure 7.2c where almost the entire dataset is a working set.

Cases where $|B|$ actually increase with the backup number, *bgr* represents the rate at which new data becomes part of *non-working set.* It suggest that part of the working set becomes stable over time and becomes base on which future segments are deduplicated. Extreme case is shown in Figure 7.2d where the effect of *bgr* starts to outweigh the effect of *usr* and the number of unique segments actually decrease. This is the most desirable case for the data deduplication where amount of data to be store grows sub-linearly and therefore more scalable in terms of backups you can store. It will also be shown that the throughput also tends to be higher in these cases for similar compression.

Cases seen in Figure 7.2a and Figure 7.1e suggest that there exists only a small change between the backups. While this is also a desirable case for data deduplication, it may also be argued that for a storage system with such little activity, a longer period of incremental backups could provide similar performance.

The logical size, *usr* and *bgr* of a dataset completely describes the dataset as long as there is no deletion involved. Furthermore, they do not vary as much as *dcr* over time. Therefore a aggregated values of *usr* and *bgr* are much better choices of parameters to describe a dataset. They show how much of new data is generated ($|L| * \overline{usr}$), how much of that data will be changed again before the next full ($1 - bgr$). Together, they allow you to calculate the compression ratio at time $i + 1$ by following these steps.

1. $|B_{i+1}| = |U_i| * bgr_i + |B_i|$.

2. $|U_{i+1}| = |L_{i+1}| * usr + |U_i|(1 - bgr_i)$.

3. Calculate $|D_{i+1}|$ and scr from above two values.

7.3.2 Segment Run Length

In section 7.2, we showed that the compression ratio and the throughput of a deduplication system can be highly correlated. Figure 7.2 shows throughput vs. compression ratio of each backup instances. This graph together with Figure 7.3 shows that the compression ratio is indeed a most relevant factor in determining throughput. However, Figure 7.3 shows some skewness both in Residual vs. Fitted graph and Residual vs. Leverage graph. They suggest at other factors that affect throughput which are not simply white noise.

Figure 7.2 also shows a linear estimation line. One interesting observation is that there are some datasets which perform consistently better than the estimation. These are *exchange_is*, *fredp4* and *fredvar* datasets. On the other hand, *workstation* dataset performs consistently worse than expected.

To quantify this effect, we take look at the deviation of throughput numbers from the regression line as shown in Table 7.3. These numbers are not absolute deviation. These are average values of $actual\ throughput - expected\ throughput$ and removes the effect of the compression ratio from the throughput numbers. As explained in section 7.1, the *content dictionary* is the only place where this variation in performance due to content could occur. Note that all data are of the same size, written sequentially once. While *content store* actually stores less if more segments are deduplicated, this performance difference is likely to be linear to the amount of data stored. Since we have removed this linear factor and concentrate only on the deviation, the effect must be due to on content dictionary behavior.

To evaluate the content dictionary behavior, we first look at the sequential runs of segments. A longer sequence of a run represents a sequential access of content dictionary making it easier for the dictionary cache to be a hit. A *run* is simply a sequence of consecutive segments that are either unique or duplicate segments. We refer to them as

Dataset	**Average Deviation**	**Max Run Len.**	**Average Run Len.**	**Run Len. > 1000**
fredp4	1.532	207400	7.944	57%
exchange_is	1.375	392000	15.88	27%
fredvar	0.852	366200	28.88	76%
exchange	0.217	183100	30.53	64%
exchange_mb	-0.722	7490	18.6	13%
workstation	-3.260	5632	6.635	9%

Table 7.3: Average Deviation from the expected throughput and their corresponding run length information

a *content run* and *duplicate run* respectively when need to distinguish between them. Table 7.3 shows aggregated *run* data. We make following observations.

Observation 4 *The average run length varies little across the datasets and shows little correlation to the throughput. However, the maximum run length varies significantly across the datasets and show high correlation.*

The correlation coefficient of maximum run length and the average deviation is $r^2 = 0.63$. It is high enough that we can safely assume there is significant correlation between the two values. The correlation coefficient of average run length and the average deviation is only $r^2 = 0.002$. The obvious reason is that the mean values of run length does not represent anything physical when the significant portions of the data lay in outlier regions. The last column of Table 7.3 shows that the over 50% of the segments fall in a run length of over 1000. It is interesting to note that the *fredvar* and *exchange* datasets suffer in throughput even though they have the highest percentage of large runs. It leads us to our next observation.

Observation 5 *Datasets with high average run length due to many extremely large runs perform worse than cases where the run length are more evenly distributed.*

The skewed run length has a negative impact on performance not because of the content dictionary but the content store. While it is obvious that less amount to be stored result in high logical throughput as described earlier, the arrival rate of segments to the content store is also important. In a simple queuing model, the arrival rate at the content store is simply $(1-p)S$ where S is the throughput of the content dictionary and p is content dictionary hit probability. If p was uniformly distributed, that the arrival rate would simply be a function of compression ratio. However, as the skewed distribution of run length shows, p of each segments are highly correlated. The effect is bursty arrivals at the content sore resulting in high queue length and lower throughput as shown in Figure 7.4. It is shown that the effect of run length is less pronounced for datasets with higher compression ratio. The *all_miss* case is the worst case where every segment is sent to the segment store and *rand* is the best case where run length distribution is uniform.

The run length analysis shows that while the throughput is highly correlated with the compression ratio, higher run length typically results in higher throughput. Additionally, skewed run length which typically results from few very long runs, results in lower throughput especially when the compression ratio is low.

7.4 Applications

Above analysis of composition of segments allows users of deduplication systems to collect specific statistics to predict future storage requirements. In the world of deduplication where the physical storage space and the logical segment space does not match, it is difficult to provision storage space. Extracting *bgr* and *usr* from the deduplicated data allow users to better predict future storage requirements.

For the given six test datasets, we use *bgr* and *usr* of the first three fulls to predict the compression ratio of 4th and 5th full and compare them with the *dcr* approach.

For example, the $bgr_1 \sim 0.01$ and $usr_1 \sim 0.26$ for *workstation* dataset. Since $|L_i| \sim 393311$, we can predict future segment composition given backup number 2 information. Figure 7.5 shows the graph of the original workstation composition and the graph of predicted composition. Since the parameters were extracted from the backup number 1 and 2, the accuracy of prediction falls as the number of fulls increase. Of course, the accuracy of the prediction itself is also dependent on the dataset and the worst case error was observed in *exchange* dataset where the *scr* at backup 4 was off by just over 0.02($\sim$ 5% error). This is much better approach than simply observing the compression ratio. For *fredp4* dataset the *dcr* prediction is off in compression ratio by

datasets	**dcr**	**usr & bgr**
fredp4	35.99%	3.68%
exchange_is	9.20%	2.21%
fredvar	7.35%	0.21%
exchange	17.82%	5.42%
exchange_mb	0.14%	0.20%
workstation	3.61%	0.61%

Table 7.4: Compression ratio prediction error comparison

0.18 ($\sim$ 36% error).

Table 7.4 shows our approach outperforms the traditional *dcr* approach in all but exchange_mb dataset where the error is about the same.

Another potential application is evaluating the impact of merging two separately deduplicated data. Unless there is a reason to suspect a large similarity between the two datasets, parameters of each dataset can be superpositioned to predict the composition of combined dataset.

A more involved application would be to filter out the *working set* from being deduplicated. Various techniques exists to evaluate the working set of a storage system [75, 57] which can be used to identify *hot data* and pass only those deemed cold to the deduplication system. This would be especially useful for datasets with low *bgr* where same portions of the data are constantly changing. This portion can deemed unworthy of deduplication and is backed up separately.

The analysis of run length suggest that it maybe more beneficial for the through put if the inter-duplicate segments are ignored. That is we only evaluate duplicate segments generated between the full backups. These single or dual duplicate segments fragment the index table and potentially the contents on disk without providing any substantial gain in compression.

Last application maybe to adjust buffer sizes at each stage of deduplication based on the run length observed to minimized skewed arrival effect.

7.5 Limitations

Many deduplication systems allow the segments to be stored without duplicate detection for the better throughput [27, 19]. This does not cause correctness issue. However, it makes predicting performance even more difficult. In this study we assume that all duplicate segments must be identified and eliminated. There has also been attempts to make better decisions on identifying the data segments from the dataset [25, 24, 16]. We claim that the segments, regardless of how they are defined, define datasets in terms of data deduplication process. This claim assumes that different datasets will have statistically similar relative compression and performance regardless of the segmentation process. In other words, segmentation process affects different dataset in statistically similar way. For example, if dataset A gained compression ratio by 10% while losing throughput by 5% by using the new segmentation method, than dataset B is highly probable to gain in compression and lose in throughput by similar amount. While this assumption is actually verifiable, lack of datasets prevents us from providing strong statistical guarantee. However, previous analysis of segmentation algorithms [24] as well as our own tests indicate no evidence to reject our assumption.

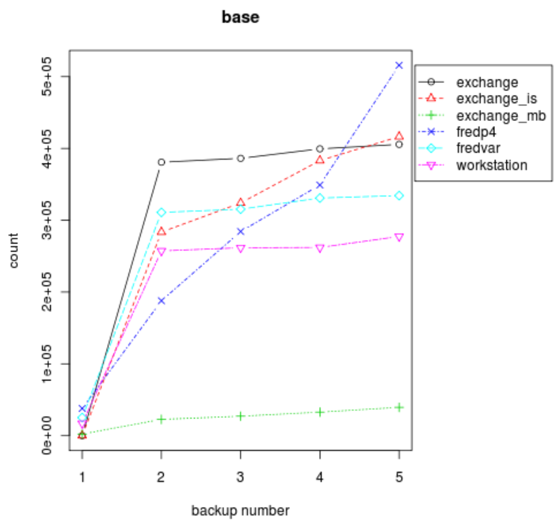
7.6 Conclusion

We have shown a framework for characterizing datasets by analyzing how different datasets behave within a data deduplication system. For the datasets presented here, we show that there are classes of datasets which behave similarly.

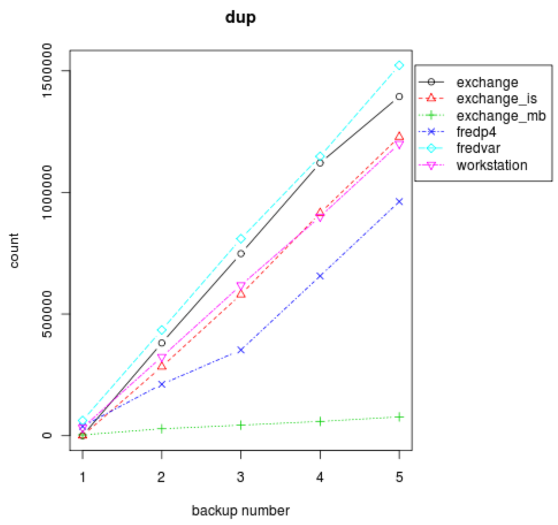
We also show that the changes in the number of base segments is more important in terms of scalable data compression than the simple compression ratio. More specifically, a positive bgr guarantees continuing reduction in the compression ratio resulting in sub-linear growth in the storage requirement. Intuitively, bgr represents life time of new data. High bgr means that newly created data are long lived and low bgr means that they are short lived. Since usr represents amount of new data created, their ratio tells you how much of the new data is transient and how much are more permanent. Generating more and more permanent data means more data to be deduplicated at each back up resulting in better compression ratio.

We also show that the segment run length has a fundamental effect on the dictionary lookup process both in terms of the cache behavior and queuing delays, which can counteract each other in terms of throughput. More specifically, the typical belief that high average run length results in better throughput is not true. In fact, it can lower the throughput due to skewed arrival rate the content store. However, high maximum run length tends to improve performance due to good caching effect of content dictionary. Simply put, one can estimate throughput based on the compression ratio using a C/R vs. throughput regression line for a given system and can further expect the throughput to be better or worse based on the run length information.

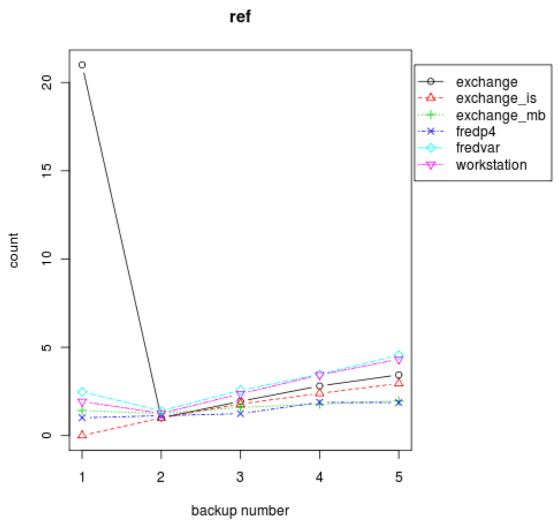
Lastly, we showed that we can increase the accuracy of compression ratio prediction by as much as 80%. Furthermore, we can accurately predict the composition of segments which allow us to determine the existence of frequently updated data which does not have to be deduplicated.



(a) Base segments



(b) Duplicate segments



(c) Average references

Figure 7.1: Comparison of data sets. Number of references are simply $|D|/|B|$. Therefore, *exchange_is* and *frep4* show lowest average number of references since their number of base segments increase with the number of duplicate segments. For all other segments, the number of base segments stay relatively steady indicating that same portion of the data is changing over and over again.

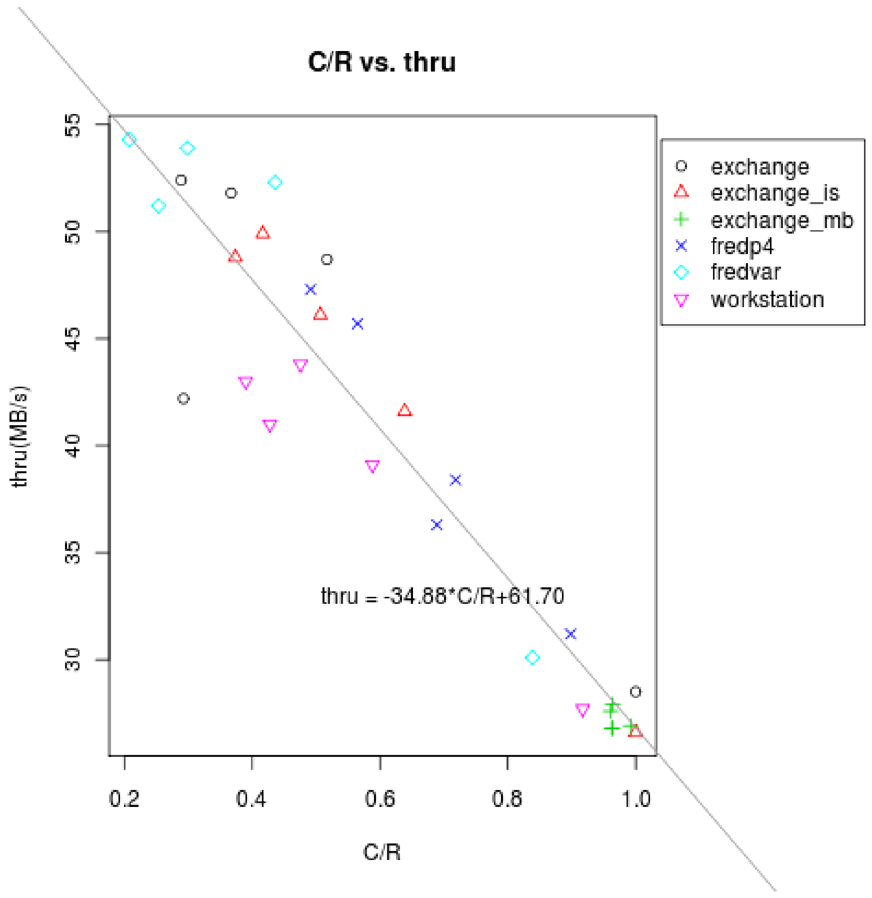


Figure 7.2: C/R vs. Throughput. The accumulated compression ratio of each dataset at each full backup is plotted against the throughput for that particular backup window. You can observe that the throughput increases as the accumulated compression ratio decreases even though every full backup is equal in size.

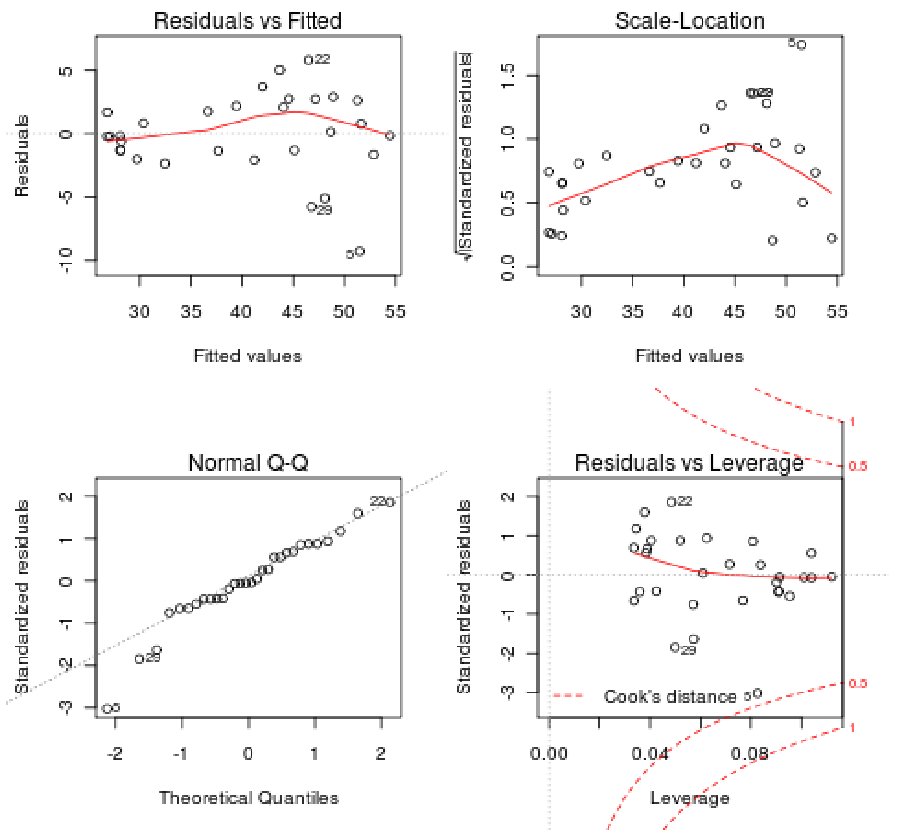


Figure 7.3: Analysis of C/R vs.throughput. The top left graph shows that while the regression is a good fit, data around throughput value of 40MB/s to 50MB/s tends to out perform expected throughput. Top right graph shows square root of residual error. The backup instance 5, 22 and 29 show a large deviation (from exchange, fredvar and workstation datasets respectively). The lower left graph shows that the residual errors are more or less normally distributed with the exception of 4, 21, 29 (from exchange, exchange_is and workstation datasets respectively). The lower right graph shows the Cook's distance. Backup instance 5 is less than 0.5 Cook's distance away and can be safely taken into evaluation.

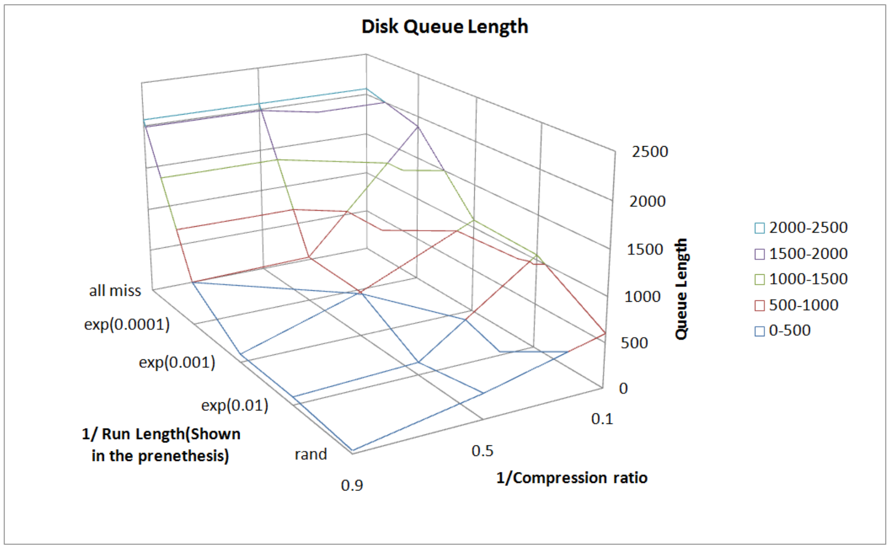
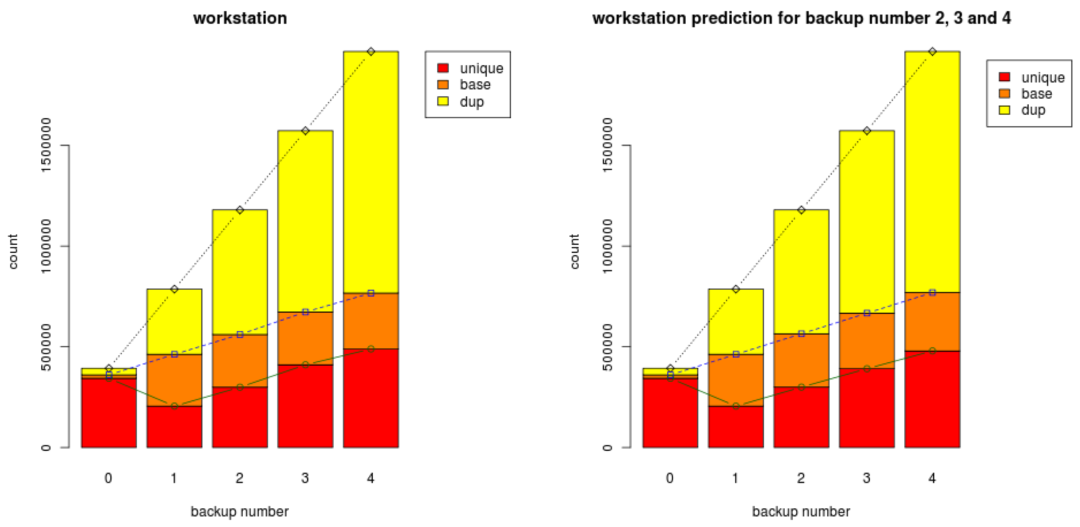


Figure 7.4: Run length vs. compression ratio vs. queue length.



(a) Real Data

(b) Predicted Data

Figure 7.5: The original segment composition of workstation and the predicted values.

Chapter 8

Related Work

8.1 Storage IO Workload Characterization

Characterizing the workload for storage subsystems started as early as 1965 in order to simulate computer system performance [76]. For almost two decades, the main focus of the characterization was on the arrival rate of IO requests [76, 77, 78]. Hence, a simple queueing model was used to describe storage system behavior [79].

At around the same time, synthetic benchmarks that takes record sizes, number of records, read rate, write rate and number of files became available [80, 45]. A plateau of works on measuring real workloads in production systems followed [81, 58, 10, 1, 82, 83, 84, 85, 86, 87]. Consequently, comprehensive and complex storage models [88, 89, 90, 91, 92] as well as storage benchmarks followed [93].

Importance of locality in sub storage systems was shown to be important in the 80's [94]. However, the efforts to quantify locality in storage systems have began only recently. Two different approaches exists. One approach relies on the fact that the autocorrelation of LBN sequence tends not to decay exponentially [95]. This effect is known as the *long memory* effect and can be quantified using *Hurst exponent* [96]. The other approach takes a more traditional approach of *Stack Distance* [97] and *Seek Distance* [98].

These new ways of describing workloads allows us to gain new insights into the workloads. However, previous works have failed to link these new characteristics to the way the workloads interact with storage systems rendering the insights to be somewhat interesting but less useful.

There exists another set of characterization that relies on extracting a set of components from the workload so that the workload can be synthetically generated [99]. Theses techniques typically rely on unintuitive parameters that cannot be meaning fully controlled. Therefore the application of these characteristics are limited to replaying a specific trace only.

8.2 Data Deduplication

The workload presented in typical deduplication systems paper are either from a privately accessible systems [23, 30, 100, 14, 19, 15] or a relatively small dataset of specific type in public domain [101, 24, 25]. However, either techniques allow a direct comparison of different systems if the dataset under evaluation is completely different or if it only covers a dataset of a particular characteristics.

There have been few work in trying to statistically characterize the input dataset to various benchmark programs [102, 103, 104]. However, these papers concentrate on generic sub-setting of dataset so that different characteristics of application is explored statistically hence lacking in accuracy when it comes to predicting exact performance variances.

Numerous works have also been applied in filesystem activities and its contents [105, 73, 105] with one paper using a simple statistical analysis to characterize a filesystem impression [106]. While these approaches all reveal some interesting intuitions of how we use storage systems, the scope is too generic to be neither accurate or simple enough to be applied in real systems. More specific workload analysis have been targeted to SSDs [74] and we use some of their approach in our analysis.

8.3 Storage Consolidation and Resource Management

Our experiments have revealed that storage workload and storage device models are highly inter-dependent. This confirms observations from other works [57, 2].

Early efforts to automatically manage storage resources in a virtualized environment such as *VectorDot* [69] ignored the workload and simply concentrated on utilization of each storage system. A workload from the over-loaded device is simply migrated to a less loaded device. This approach would fail to work in a heterogeneous storage environment since the resulting performance is unknown. Furthermore, the goal is not to simply ensure that no storage system suffers from over-utilization but to minimize the performance degradation caused by resource sharing across all workloads.

Other work has focused on highly accurate latency prediction of workloads across storage devices but this has come at the cost of practicality by requiring extensive off-line experimentation and modeling. For instance, Wang et al. [57] developed CART models off-line to develop workload-storage models. In contrast to their work, Romano aims to be learn storage behavior quickly using online calibration without resorting to intrusive off-line methods.

Earlier work in this area include Basil [60] and Pesto [61] which propose creating separate workload and device models and using heuristics to perform load balancing. The underlying assumption is the independence of the two classes of models. A commercial product [53] based on Pesto is now available. Whereas Basil and Pesto have very good average behavior, they suffer from severe corner case model inaccuracies. For example, it was shown in their work that the prediction error can be more than 10 times larger than the latency it predicts for sequential workloads. These can lead to bad data movement recommendations. To address these shortcomings, Pesto applies aggressive cost-benefit analysis to prune out potentially bad moves. The inadvertent result of this pruning to deal with model inaccuracies is that many highly beneficial moves are also left out.

Romano proposes to deal with these problems by creating a model capable of determining its own confidence of prediction using *prediction interval* [107]. Furthermore, Romano is capable of quantifying the effect of the workload characteristics and their interactions for a given data store resulting in much more accurate and robust modeling.

Chapter 9

Conclusion and Discussion

9.1 Summary

In this thesis, we have shown that utilizing statistical tools allow us to analyze and design storage systems in a workload dependent manner. Rather than trying to find one or few points in the workload space that are representative, quantitative description of the workload and its interaction with the system provides not only insight into the complex and stateful system but also allow us to design a system that is more robust and efficient.

To quantify the interaction between the workload and the system using synthetic workloads, we have shown that it is better to use a single benchmark with better controlled input vector than to use different benchmark programs. One technique called PB method was shown to generate an efficient input vector of size linear to the number of parameters with an assumption that all parameters have a monotonic effect on the performance. Such dramatic decrease in the input vector size allows user to repeat experiments to increase confidence of the experiment.

It is also shown that 2 to 4 input parameters are capable of covering the majority of storage system's operational space. This counter intuitive result shows that we can further reduce the size of the input vector, allowing us to run more exhaustive experiments to gain more that the first order effects.

We used such technique to create a storage performance model. A full-factorial linear regression is performed using only 4 parameters. The resulting model is surprisingly accurate for static workloads. More importantly, the model is well behaved. We show that misprediction falls closely to the target confidence level.

The main benefits of the proposed model is ease with which the model could be updated with the performance data monitored at runtime, ability to trade the confidence interval with statistical significance and generality of the method.

Based on the model we have designed a load balancing mechanism, called Romano, for virtualized environment with heterogeneous workloads and storage systems. We have shown that Romano can outperform previous systems with same goals by up to 80% in prediction accuracy. This increased accuracy results in 78% reduction in maximum latency observed and 82% reduction in variance while load balancing.

We have shown that the performance differences between data stores cannot be described with a single number. More specifically, we have shown that the storage performance must be described in terms of the workload.

The last contribution follows the second contribution. Since the performance of storage systems effectively change with the workload, the load balancing problem is no longer a bin-packing problem as previously believed. Therefore we present a probabilistic approach to finding a pseudo-optimal mapping of workloads to the data stores. It is important to mention that probabilistic approach is only used to find the final state of the system and the moves are actually made greedily to speed up the convergence and minimize the number of moves that has to be made.

We believe that Romano not only provides means for more efficient load balancing but also in many other areas such as QoS management, power management and performance tunning in tiered storage.

If the system-workload interaction is simple enough for a given metric, we can gain a much more precise result by analytical model of underlying interaction. We demonstrate such model of compression ratio by analyzing how the data deduplication system interacts with the backup workloads. By simply analyzing how the unique segments are generated and converted to duplicate segments, we were able to improve the prediction accuracy by as much as 80% compared to simple method using *average data change rate*. Furthermore, because the throughput of the deduplication systems linearly depends on the compression ratio, we were able to get insights into the potential future throughput as well.

In this dissertation, we have evaluated techniques to parametrize workload characteristics in terms of how it affects the system behavior. We have shown that the correct parameterization simplifies the system response model allowing easier online updates of the model. These models are capable of predicting future behavior of the system allowing system administrators to simulate system changes without actually making changes. Furthermore, deterministic nature of the models allow programs to simulate changes and optimize the system for a given goal.

9.2 Future Work

9.2.1 Characterizing Dynamic Behavior of the Workload

In this work, we have characterized static workload characteristics and observed how it is varied over time. The interaction between the system and the static workload is modeled which is simply applied to time series of of the characteristics to estimate the performance. We assume that by carefully choosing workload characterization parameters the effect of statefullness allowing a memoryless model to predict the system performance. However, in reality we know that this does not always work. For example, the garbage collection pattern in solid state disks [108] or content addressable storage [109] cannot be described using memoryless model. Furthermore, the workload themselves have shown to have a long memory process properties [96] and does not render itself easily to such models.

We have initially evaluated means to capture time domain properties as well as spatial domain properties by evaluating entropy rates. The technique is similar to *PQRS* [57] but allows dynamic characterization at run time.

The goal would be to able to generate workloads of some duration given a set of characteristics parameters.

9.2.2 Locality Characterization

One of the key performance determinant in a storage system is how the workload interacts with its cache. With reducing cost of SDRAM and NAND flash, cache sizes of over 1/1000th of underlying storage system is becoming viable. We have found that the block accessed at time t have a very low probability of being accessed again before $t + \delta t$ for some δt. However, it is highly likely that the block close to the block access at time t will be accessed with in δt. Based on the observation, we are evaluating a technique that allows us to predict the effectiveness of the data prefetching through cache block size control.

References

[1] Liang Wang. Workload configurations for typical enterprise workloads. Technical report, Microsoft, 2009. `http://blogs.msdn.com/b/tvoellm/archive/2009/05/07/useful-io-profiles-for-simulating-various-workloads.aspx`.

[2] N Park, W Xiao, K Choi, and DJ Lilja. A statistical evaluation of the impact of parameter selection on storage system benchmarks. In *7th IEEE International Workshop on Storage Network Architecture and Parallel I/O*, SNAPI'11. IEEE, 2011.

[3] Nohhyun Park, Irfan Ahmad, and David J Lilja. Romano: autonomous storage management using performance prediction in multi-tenant datacenters. In *Proceedings of the Third ACM Symposium on Cloud Computing*, 2012.

[4] Nohhyun Park and David J Lilja. Characterizing datasets for data deduplication in backup applications. In *Workload Characterization (IISWC), 2010 IEEE International Symposium on*, 2010.

[5] Youngjin Nam, Guanlin Lu, Nohhyun Park, Weijun Xiao, and David HC Du. Chunk fragmentation level: An effective indicator for read performance degradation in deduplication storage. In *High Performance Computing and Communications (HPCC), 2011 IEEE 13th International Conference on*, 2011.

[6] S.B. Vaghani. Virtual machine file system. *ACM SIGOPS Operating Systems Review*, 44(4):57–70, 2010.

[7] Karan Gupta, Reshu Jain, Ioannis Koltsidas, Himabindu Pucha, Prasenjit Sarkar, Mark Seaman, and Dinesh Subhraveti. Gpfs-snc: An enterprise storage framework

for virtual-machine clouds. *IBM Journal of Research and Development*, 55(6):2–1, 2011.

[8] Steven R Soltis, Thomas M Ruwart, and Matthew T O'keefe. The global file system. In *NASA CONFERENCE PUBLICATION*, pages 319–342, 1996.

[9] VMware, http://pubs.vmware.com/vsphere-51/index.jsp. *VMware vSphere 5.1 Documentation.*

[10] I. Ahmad. Easy and efficient disk i/o workload characterization in vmware esx server. In *Workload Characterization, 2007. IISWC 2007. IEEE 10th International Symposium on*, IISWC '07. IEEE, 2007.

[11] Seagate. *SCSI commands reference manual*, rev. a edition, 2006.

[12] C. Clark, K. Fraser, S. Hand, J.G. Hansen, E. Jul, C. Limpach, I. Pratt, and A. Warfield. Live migration of virtual machines. In *Proceedings of the 2nd conference on Symposium on Networked Systems Design & Implementation*, NSDI '05. USENIX Association, 2005.

[13] A. Mashtizadeh, E. Celebi, T. Garfinkel, and M. Cai. The design and evolution of live storage migration in vmware esx. In *Proceedings of the 2011 USENIX Annual Technical Conference*, USENIX ATC'11. USENIX Association, 2011.

[14] Lior Aronovich, Ron Asher, Eitan Bachmat, Haim Bitner, Michael Hirsch, and Shmuel T. Klein. The design of a similarity based deduplication system. In *Proceedings of SYSTOR 2009: The Israeli Experimental Systems Conference*, SYSTOR '09, New York, NY, USA, 2009. ACM.

[15] Benjamin Zhu, Kai Li, and Hugo Patterson. Avoiding the disk bottleneck in the data domain deduplication file system. In *Proceedings of the 6th USENIX Conference on File and Storage Technologies*, Berkeley, CA, USA, 2008. USENIX Association.

[16] D.R. Bobbarjung, S. Jagannathan, and C. Dubnicki. Improving duplicate elimination in storage systems. *ACM Transactions on Storage (TOS)*, 2(4):448, 2006.

[17] Athicha Muthitacharoen, Benjie Chen, and David Mazières. A low-bandwidth network file system. *SIGOPS Oper. Syst. Rev.*, 35:174–187, 2001.

[18] Dirk Meister and André Brinkmann. Multi-level comparison of data deduplication in a backup scenario. In *Proceedings of SYSTOR 2009: The Israeli Experimental Systems Conference*, SYSTOR '09, New York, NY, USA, 2009. ACM.

[19] Mark Lillibridge, Kave Eshghi, Deepavali Bhagwat, Vinay Deolalikar, Greg Trezise, and Peter Camble. Sparse indexing: large scale, inline deduplication using sampling and locality. In *Proceedings of the USENIX conference on File and storage technologies*, Berkeley, CA, USA, 2009. USENIX Association.

[20] LL You, KT Pollack, and DDE Long. Deep Store: An archival storage system architecture. In *Data Engineering, 2005. ICDE 2005. Proceedings. 21st International Conference on*, pages 804–815, 2005.

[21] M.A. Smith, J. Pieper, D. Gruhl, and L.V. Real. IZO: applications of large-window compression to virtual machine management. In *Proceedings of the 22nd conference on Large installation system administration conference*, pages 121–132. USENIX Association, 2008.

[22] Keren Jin and Ethan L. Miller. The effectiveness of deduplication on virtual machine disk images. In *Proceedings of SYSTOR 2009: The Israeli Experimental Systems Conference*, SYSTOR '09, New York, NY, USA, 2009. ACM.

[23] A.T. Clements, I. Ahmad, M. Vilayannur, J. Li, I. VMware, and MIT CSAIL. Decentralized deduplication in san cluster file systems. In *Proceedings of the USENIX Annual Technical Conference*, 2009.

[24] E. Kruus, C. Ungureanu, and C. Dubnicki. Bimodal Content Defined Chunking for Backup Streams. In *Proceedings of the Eighth USENIX Conference on File and Storage Technologies*, 2010.

[25] K. Eshghi and H.K. Tang. A framework for analyzing and improving content-based chunking algorithms. *Hewlett-Packard Labs Technical Report TR*, 30, 2005.

[26] Nagapramod Mandagere, Pin Zhou, Mark A Smith, and Sandeep Uttamchandani. Demystifying data deduplication. In *Proceedings of the ACM/IFIP/USENIX Middleware '08 Conference Companion*, Companion '08, New York, NY, USA, 2008. ACM.

[27] D. Bhagwat, K. Eshghi, D.D.E. Long, and M. Lillibridge. Extreme binning: Scalable, parallel deduplication for chunk-based file backup. In *Proceedings of the IEEE International Symposium on Modeling, Analysis, and Simulation of Computer and Telecommunication Systems (MASCOTS)*, 2009.

[28] Burton H. Bloom. Space/time trade-offs in hash coding with allowable errors. *Commun. ACM*, 13:422–426, 1970.

[29] Ricardo Koller and Raju Rangaswami. I/o deduplication: Utilizing content similarity to improve i/o performance. In *Proceedings of the 8th conference on file and storage technologies (FAST '10)*, 2010.

[30] Sean Rhea, Russ Cox, and Alex Pesterev. Fast, inexpensive content-addressed storage in foundation. In *Proceedings of the USENIX Annual Technical Conference*, 2008.

[31] J. Bonwick, M. Ahrens, V. Henson, M. Maybee, and M. Shellenbaum. The Zettabyte File System. In *FAST 2003: 2nd Usenix Conference on File and Storage Technologies*, 2003.

[32] M. Dillon. The Hammer Filesystem, 2008.

[33] Robin L Plackett and J Peter Burman. The design of optimum multifactorial experiments. *Biometrika*, 33(4):305–325, 1946.

[34] Joshua J Yi, David J Lilja, and Douglas M Hawkins. A statistically rigorous approach for improving simulation methodology. In *High-Performance Computer Architecture, 2003. HPCA-9 2003. Proceedings. The Ninth International Symposium on*, pages 281–291. IEEE, 2003.

[35] Biplob K Debnath, David J Lilja, and Mohamed F Mokbel. Sard: A statistical approach for ranking database tuning parameters. In *Data Engineering Workshop,*

2008. ICDEW 2008. IEEE 24th International Conference on, pages 11–18. IEEE, 2008.

[36] Ronald A Fisher. Design of experiments. *British Medical Journal*, 1(3923):554, 1936.

[37] James V Stone. *Independent component analysis.* Wiley Online Library, 2004.

[38] Vassilios N Christopoulos, David J Lilja, Paul R Schrater, and Apostolos Georgopoulos. Independent component analysis and evolutionary algorithms for building representative benchmark subsets. In *Performance Analysis of Systems and software, 2008. ISPASS 2008. IEEE International Symposium on*, pages 169–178. IEEE, 2008.

[39] Lieven Eeckhout, Rashmi Sundareswara, JJ Yi, David J Lilja, and Paul Schrater. Accurate statistical approaches for generating representative workload compositions. In *Workload Characterization Symposium, 2005. Proceedings of the IEEE International*, pages 56–66. IEEE, 2005.

[40] D.J. Lilja. *Measuring computer performance: a practitioner's guide.* Cambridge Univ Press, 2005.

[41] Peter M Chen and David A Patterson. Storage performance-metrics and benchmarks. *Proceedings of the IEEE*, 81(8):1151–1165, 1993.

[42] Jens Axboe. Fio-flexible io tester. Technical report, Freecode, 2008. `http://freecode.com/projects/fio`.

[43] Henk Vandenbergh. Vdbench users guide, 2008.

[44] Jeffrey Katcher. Postmark: A new file system benchmark. Technical report, Technical Report TR3022, Network Appliance., 1997. `http://www.netapp.com/tech_library/3022.html`.

[45] Barry Wolman and Thomas M Olson. Iobench: a system independent io benchmark. *ACM SIGARCH Computer Architecture News*, 17(5):55–70, 1989.

[46] William D Norcott and Don Capps. Iozone filesystem benchmark, 2006. `www.iozone.org`.

[47] Russell Coker. Bonnie++, 2001.

[48] David Plonka, Archit Gupta, and Dale Carder. Application buffer-cache management for performance: Running the world's largest mrtg. In *LISA*, pages 63–78, 2007.

[49] Jacques Hadamar. *An Essay on: He Psichology of Invention in the Mathematical Field*, volume 107. DoverPublications. com, 1954.

[50] Pauliina Ilmonen, Klaus Nordhausen, Hannu Oja, and Esa Ollila. A new performance index for ica: properties, computation and asymptotic analysis. In *Latent Variable Analysis and Signal Separation*, pages 229–236. Springer, 2010.

[51] Gabor J Szekely and Maria L Rizzo. Hierarchical clustering via joint between-within distances: Extending ward's minimum variance method. *Journal of classification*, 22(2):151–183, 2005.

[52] F. Hayashi. *Econometrics*. Princeton University Press, 2000.

[53] VMware. Resource management with VMware DRS. `http://vmware.com/pdf/vmware_drs_wp.pdf`, 2006.

[54] M. Seltzer, D. Krinsky, K. Smith, and X. Zhang. The case for application-specific benchmarking. In *Hot Topics in Operating Systems, 1999. Proceedings of the Seventh Workshop on*, pages 102–107. IEEE, 1999.

[55] J. Sievert. Iometer: The I/O performance analysis tool for servers, 2004.

[56] Zhaobin Liu, Bo Jiang, Zixiang Zhao, and Yunhan Jiang. Modeling and simulation of self-similar storage i/o. In *Proceedings of the 3rd International Conference on Advances in Grid and Pervasive Computing*, GPC'08. IEEE, 2008.

[57] M. Wang, K. Au, A. Ailamaki, A. Brockwell, C. Faloutsos, and G.R. Ganger. Storage device performance prediction with cart models. In *Proceedings of the 12th*

International Symposium on Modeling, Analysis, and Simulation of Computer and Telecommunications Systems, MASCOTS'04. IEEE, 2004.

[58] S. Kavalanekar, B. Worthington, Qi Zhang, and V. Sharda. Characterization of storage workload traces from production windows servers. In *Workload Characterization, 2008. IISWC 2008. IEEE International Symposium on*, IISWC '08. IEEE, 2008.

[59] V. Tarasov, S. Kumar, J. Ma, D. Hildebrand, A. Povzner, G. Kuenning, and E. Zadok. Extracting flexible, replayable models from large block traces. In *Proccedings of the 10th USENIX conference on File and storage technologies*, FAST'12. USENIX Association, 2012.

[60] Ajay Gulati, Chethan Kumar, Irfan Ahmad, and Karan Kumar. BASIL: Automated IO load balancing across storage devices. In *Proceedings of the 8th USENIX conference on File and Storage Technologies*, FAST'10. USENIX Association, 2010.

[61] Ajay Gulati, Ganesha Shanmuganathan, Irfan Ahmad, Carl A. Waldspurger, and Mustafa Uysal. Pesto: Online storage performance management in virtualized datacenters. In *Proceedings of the 2nd ACM Symposium on Clound Computing*, SOCC'11. ACM, 2011.

[62] J.D.C. Little. A proof for the queuing formula: L= λ w. *Operations research*, pages 383–387, 1961.

[63] I. Ahmad, A. Gulati, and A. Mashtizadeh. vic: Interrupt coalescing for virtual machine storage device io. In *Proceedings of the 2011 USENIX Annual Technical Conference*, USENIX ATC'11. USENIX Association, 2011.

[64] Duncan Epping and Frank Denneman. *VMware vSphere 5 clustering: technical deepdive*. VMware, 2011.

[65] David R. Merrill. Storage economics: Four principles for reducing total cost of ownership. Technical report, Hitachi Data Systems, 2009. `http://www.hds.com/assets/pdf/four-principles-for-reducing-total-cost-of-ownership.pdf`.

[66] Nik Simpson. Building a data center cost model. Technical report, Burton Group, 2010. `http://www.burtongroup.com/Research/DocumentList.aspx?cid=49`.

[67] T. Wood, P. Shenoy, A. Venkataramani, and M. Yousif. Black-box and gray-box strategies for virtual machine migration. In *Proceedings of the 4th conference on Symposium on Networked Systems Design & Implementation*, NSDI '07. USENIX Association, 2007.

[68] M. Nelson, B.H. Lim, and G. Hutchins. Fast transparent migration for virtual machines. In *Proceedings of the 2005 USENIX Annual Technical Conference*, USENIX ATC'05. USENIX Association, 2005.

[69] A. Singh, M. Korupolu, and D. Mohapatra. Server-storage virtualization: integration and load balancing in data centers. In *Proceedings of the 2008 ACM/IEEE conference on Supercomputing*, SC'08. IEEE Press, 2008.

[70] S. Kirkpatrick, C.D. Gelatt Jr, and M.P. Vecchi. Optimization by simulated annealing. *science*, 220(4598):671–680, 1983.

[71] V. Granville, M. Krivánek, and J.P. Rasson. Simulated annealing: A proof of convergence. *Pattern Analysis and Machine Intelligence, IEEE Transactions on*, 16(6):652–656, 1994.

[72] Alma Riska and Erik Riedel. Evaluation of disk-level workloads at different time scales. *SIGMETRICS Perform. Eval. Rev.*, 37:67–68, 2009.

[73] Andrew W. Leung, Shankar Pasupathy, Garth Goodson, and Ethan L. Miller. Measurement and analysis of large-scale network file system workloads. In *USENIX 2008 Annual Technical Conference on Annual Technical Conference*, Berkeley, CA, USA, 2008. USENIX Association.

[74] G. Soundararajan, V. Prabhakaran, M. Balakrishnan, and T. Wobber. Extending SSD Lifetimes with Disk-Based Write Caches. In *Proceedings of the USENIX Conference on File and Storage Technologies*. FAST, 2010.

[75] S. Lee, K. Ha, K. Zhang, J. Kim, and J. Kim. FlexFS: A Flexible Flash File System for MLC NAND Flash Memory. In *Proceedings of the USENIX Annual Technical Conference, San Diego, CA*, 2009.

[76] George K Hutchinson and John Norris Maguire. Computer systems design and analysis through simulation. In *Proceedings of the November 30–December 1, 1965, fall joint computer conference, part I*, AFIPS '65 (Fall, part I). ACM, 1965.

[77] Kurt Fuchel and Sidney Heller. Consideration in the design of a multiple computer system with extended core storage. In *Proceedings of the first ACM symposium on Operating System Principles*, SOSP '67. ACM, 1967.

[78] A.L.N. Reddy and P. Banerjee. An evaluation of multiple-disk i/o systems. *Computers, IEEE Transactions on*, 38(12):1680–1690, 1989.

[79] Leonard Kleinrock. *Queueing systems. volume 1: Theory.* Wiley-Interscience, 1975.

[80] Arvin Park, Jeffrey C Becker, and Richard J Lipton. Iostone: a synthetic file system benchmark. *ACM SIGARCH Computer Architecture News*, 18(2):45–52, 1990.

[81] Chris Ruemmler and John Wilkes. Unix disk access patterns. In *Proceedings of the USENIX Winter Conference*. USENIX Association, 1993.

[82] Drew S Roselli, Jacob R Lorch, Thomas E Anderson, et al. A comparison of file system workloads. In *Proceedings of the General Track: 2000 USENIX Annual Technical Conference*, ATC '00. USENIX Association, 2000.

[83] Maria Calzarossa and Giuseppe Serazzi. Workload characterization: A survey. *Proceedings of the IEEE*, 81(8):1136–1150, 1993.

[84] Evgenia Smirni and Daniel A Reed. Workload characterization of input/output intensive parallel applications. In *Computer Performance Evaluation Modelling Techniques and Tools*, pages 169–180. Springer, 1997.

[85] Maria Calzarossa, Luisa Massari, and Daniele Tessera. Workload characterization issues and methodologies. In *Performance Evaluation: Origins and Directions*, pages 459–482. Springer, 2000.

[86] Martin F Arlitt and Carey L Williamson. Web server workload characterization: The search for invariants. *ACM SIGMETRICS Performance Evaluation Review*, 24:126–137, 1996.

[87] Alma Riska and Erik Riedel. Disk drive level workload characterization. In *USENIX Annual Technical Conference, General Track*, ATC '06. USENIX Association, 2006.

[88] James A Anderson. A memory storage model utilizing spatial correlation functions. *Kybernetik*, 5(3):113–119, 1968.

[89] F Brichet, Jim Roberts, Alan Simonian, and Darryl Veitch. Heavy traffic analysis of a storage model with long range dependent on/off sources. *Queueing Systems*, 23(1-4):197–215, 1996.

[90] George P Copeland and Setrag N Khoshafian. A decomposition storage model. In *ACM SIGMOD Record*, volume 14, pages 268–279. ACM, 1985.

[91] Eric Freeman and David Gelernter. Lifestreams: A storage model for personal data. *ACM SIGMOD Record*, 25(1):80–86, 1996.

[92] Ilkka Norros. A storage model with self-similar input. *Queueing systems*, 16(3-4):387–396, 1994.

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

[94] John K Ousterhout, Herve Da Costa, David Harrison, John A Kunze, Mike Kupfer, and James G Thompson. A trace-driven analysis of the unix 4.2 bsd file system. In *Proceedings of the tenth ACM symposium on Operating systems principles*, SOSP '85. ACM, 1985.

[95] Peter J Denning and Stuart C Schwartz. Properties of the working-set model. *Communications of the ACM*, 15(3):191–198, 1972.

[96] María Engracia Gomez and Vicente Santonja. A new approach in the modeling and generation of synthetic disk workload. In *Modeling, Analysis and Simulation of Computer and Telecommunication Systems, 2000. Proceedings. 8th International Symposium on*, pages 199–206. IEEE, 2000.

[97] Richard L. Mattson, Jan Gecsei, Donald R. Slutz, and Irving L. Traiger. Evaluation techniques for storage hierarchies. *IBM Systems journal*, 9(2):78–117, 1970.

[98] Cory Fox, Dragan Lojpur, and A-IA Wang. Quantifying temporal and spatial localities in storage workloads and transformations by data path components. In *Modeling, Analysis and Simulation of Computers and Telecommunication Systems, 2008. MASCOTS 2008. IEEE International Symposium on*, pages 1–10. IEEE, 2008.

[99] K. Sreenivasan and A. J. Kleinman. On the construction of a representative synthetic workload. *Commun. ACM*, 17(3):127–133, mar 1974.

[100] U. Manber et al. Finding similar files in a large file system. In *Proceedings of the USENIX winter technical conference*, 1994.

[101] Mark W. Storer, Kevin Greenan, Darrell D.E. Long, and Ethan L. Miller. Secure data deduplication. In *Proceedings of the 4th ACM international workshop on Storage security and survivability*, StorageSS '08, New York, NY, USA, 2008. ACM.

[102] Lieven Eeckhout, Hans Vandierendonck, and Koen De Bosschere. Quantifying the impact of input data sets on program behavior and its applications. *Journal of Instruction-Level Parallelism*, 5:1–33, 2003.

[103] Joshua J. Yi, David J. Lilja, and Douglas M. Hawkins. A statistically rigorous approach for improving simulation methodology. In *in Proceedings of the Ninth International Symposium on High Performance Computer Architecture (HPCA-9)*, 2002.

[104] Wei Chung Hsu, Howard Chen, Pen Chung Yew, and Dong-Yuan Chen. On the predictability of program behavior using different input data sets. In *Proceedings of the Sixth Annual Workshop on Interaction between Compilers and Computer Architectures*, INTERACT '02, Washington, DC, USA, 2002. IEEE Computer Society.

[105] J.R. Douceur and W.J. Bolosky. A large-scale study of file-system contents. *ACM SIGMETRICS Performance Evaluation Review*, 27(1):70, 1999.

[106] Nitin Agrawal, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau. Generating realistic impressions for file-system benchmarking. *Trans. Storage*, 5:16:1–16:30, December 2009.

[107] C. Chatfield. Calculating interval forecasts. *Journal of Business & Economic Statistics*, pages 121–135, 1993.

[108] Nitin Agrawal, Vijayan Prabhakaran, Ted Wobber, John D Davis, Mark S Manasse, and Rina Panigrahy. Design tradeoffs for ssd performance. In *USENIX Annual Technical Conference*, pages 57–70, 2008.

[109] Cristian Ungureanu, Benjamin Atkin, Akshat Aranya, Salil Gokhale, Stephen Rago, Grzegorz Calkowski, Cezary Dubnicki, and Aniruddha Bohra. Hydrafs: A high-throughput file system for the hydrastor content-addressable storage system. In *FAST*, pages 225–238, 2010.