Intelligent Block Level I/O Workload Characterization for a Temporal and Spatial Locality Aware Workload Generator

**A THESIS
SUBMITTED TO THE FACULTY OF
UNIVERSITY OF MINNESOTA
BY**

Keerthi Palanivel

**IN PARTIAL FULFILLMENT OF THE REQUIREMENTS
FOR THE DEGREE OF
MASTER OF SCIENCE**

David J. Lilja

June 2014

Acknowledgements

The author is extremely fortunate in having the assistance of several talented academicians during the conduct of research reported in this thesis. Notably, Professor David J. Lilja, who supervised this work and contributed several key ideas herein, proved a very valuable source of encouragement and support. Also, Professor David Du assisted very substantially in providing a good research direction, and guidance regarding various algorithms and storage system intricacies. Professor Carl Sturtivant is remembered for his help with implementation of efficient algorithm used in stack distance calculations. Nohhyun Park who started this work had done a great job in understanding the initial challenges, ideas and steps to take, and his support is greatly remembered. Additionally, the generous support of Seagate Technology to this work is sincerely appreciated. I would like to thank Dave Anderson, Gary Moorhead, Mike Miller and Norman Binger for their continual support throughout the project, without their support this activity would not likely have been possible. Finally I would like to thank my parents for always providing me with proper motivation, support and guidance.

Dedication

This thesis is dedicated to my parents

Abstract

Performance of a system is a function of the system properties, and the workload seen by the system. One of the best ways to improve performance in systems is to tune or design the system based on the input workload. Localities in workloads greatly dictate the benefits one can extract from various cache hierarchies of the system stack. However, existing synthetic workload generators fail to reproduce traces that are a good representative of the original application, in terms of temporal and spatial localities. Additionally, existing workload generators are not flexible, and cannot handle cases that mimic changes in application behavior. Hence a probabilistic workload generator framework that produces synthetic trace with similar characteristics and locality as the original application, and has the support to accept or tune various workload parameter values to mimic existing or predicted workloads is presented. Apart from that, this workload generator has integration with a replay engine to issue trace IOs to a real system, or a storage simulator. Microsoft Research Traces were used for validating the tool, and the results show with up to 90% confidence that the ordering of synthetic trace is similar to the real trace. This tool can be used to study workloads in various environments like VM, cloud, database etc., and perform system optimizations or load studies.

Table of Contents

List of Tables

List of Figures

Chapter 1

Introduction

1.1 Motivation, challenges and solution

The performance of an application highly depends on the workload that it is running. Most of today's workloads are IO centric. The workload keeps evolving with changing times. In current times, cloud and big data workloads are becoming a common place, and such applications that require huge external storage have necessitated the need for external storage infrastructure. In such a situation processors, storage systems, and other peripherals evolve to support the changing trend. Delimitrou et al [19] identified that existing benchmarks are not suitable for cloud environments, and that existing workload generators lack the ability to reproduce desired temporal and spatial locality.

Storage systems are one of the major bottlenecks in today's computer system and increasing its efficiency would help to improve overall application performance. Al-Kiswany et al. [20] identified that large scale (peta scale) computing will cause performance and scalability bottleneck, and suggested application optimized storage system design. Achieving such designs are possible after carefully understanding the characteristics of the workload seen by the system. When an application issues IOs (read/write commands), they get converted into block IO commands, which then get issued to the drive (as SCSI or iSCSI or SATA commands). In this work, the main focus is the block IO layer. The application behavior might not be completely visible at the block layer, though, due to the absorption of temporal locality due to higher cache layers. Study by Wong and Wilkes [21] shows the transformation of temporal and spatial locality by the caches. Workloads determine which caching and migration policies to use. Hence, studying the workload characteristics in terms of read-write ratio, access sizes, access locality etc. are highly important. Otherwise it might lead to storage system design inefficiencies and over provisioning.

Workloads also depend on application configuration. This means that even for a given application, the behavior will change based on settings. Hence, if one uses synthetic workload generator, then such changes should be factored in [22].

Additionally, locality in the application greatly affects the response time of an IO request. Higher temporal and spatial locality translates to good response times, and better cache behavior. Request sizes also play an important role in response times. If an application issues reads for large records, then reads would be much quicker than the case when it issues small reads. This is because there would be more spatial locality in the workloads, and effective prefetching schemes could be used. In general, achieving good locality in workloads, and having good cache behavior is becoming increasing important with peta-scale and exa-scale computing.

The size of buffer cache or the writing schemes also affect the workload seen by the storage layer. In case of Linux, there is a common buffer-cache to cache data from the underlying devices, and which is common to all underlying file systems and block devices. In Linux the size of buffer might vary from 512, 1024, 2048, 4096 to 8192 bytes [27]. Whenever a read command is issued, the file system would check its buffer cache to see if the data item is cached and up to date, else it will try to evict one of the buffer's data and this new data would be added. Here, we can also see that modification of buffers or writes would lead to disk writes and increased latency. In case of a bursty workload, if the file system issues the read/write commands at the same rate without buffering, then the disk controller queue would be saturated, and there might be more failures.

Workload characterization is performed at various levels of system hierarchy; for example, operating system layer, storage IO layer, network IO layer etc. In each case workload characteristics could be described using certain metrics or visually using histograms or other plots. Storing and capturing disk IO traces is one method of reproducing workload behavior. However, storage space required for today's traces are too high (100s of GBs). And as the number of applications or its versions change, the total number of traces to store also increases tremendously. Hence, in order to tackle this problem many people use techniques like storing a part of the trace and replicating it multiple times, or they resort to sampling the trace. There are also a few workload generators that claim to produce the same trace characteristics as the original application. However, careful analysis reveals that many of these workload generators like FIO, IOMETER, IOBench etc do not produce the same stack distance, block distance or cache properties as the original trace.

Hence, a novel methodology to artificially generate trace that is statistically similar to the original trace in terms of temporal and spatial locality, read write ratio, request size distributions and arrival pattern is proposed. Initially the trace is characterized to extract all the properties of the application. Then the correlations existing between various dimensions of the workload are studied, and a model is fit to the data. Then a workload with similar statistical distribution as the real trace was produced using the proposed methodology. Inspired by biological algorithms, the novel method of using simulated annealing to achieve stack distance convergence was tried. Comparing the real trace against the synthetic trace validated this methodology, and by observing the p-value associated with the Spearman's correlation index. This tool can be used to study workloads in various environments like VM, cloud, database etc., and perform system optimizations or load studies.

1.2 Thesis contributions and organization

The primary contributions of this thesis are as follows:

1. *A comprehensive workload characterization is carried out for various enterprise workloads.*

   Characterization of workloads is done, and important patterns in workloads like long-range dependency, stack distance behavior, and block distance behavior are observed. Workload is considered to be a set of n tuples, where its elements are set of characterization values. Workload dimensions that are time dependent like inter-arrival time are shown to have bad auto-correlation for enterprise workloads, suggesting a bursty and random nature.
2. *New techniques are developed to reproduce temporal and spatial locality in synthetic workload generator.*

   Localities in workloads are shown to directly affect the caching properties. A novel method of using simulated annealing combined with other statistical tools to reproduce temporal and spatial locality is suggested. The research done shows that majority of existing synthetic workload generators do not reproduce desired localities.

3. *A robust synthetic workload generator, which can reproduce workload behavior with about more than 90% statistical confidence is developed*
   The proposed methodology and tool can be used to imitate real workloads with high degree of statistical confidence, and provides a convenient and easy to use user-interface.
4. *Phase detection techniques, entropy and burstiness analysis is performed*
   Characterization work is extended to detect phases in workloads, and study the entropy of the workloads to detect patterns. Also, burstiness of the workload is studied, and a methodology to recreate them synthetically is devised.

The rest of the thesis is organized as follows:

- Chapter 2 is the background where existing workload generator tools are compared and contrasted, and related works are presented
- Chapter 3 talks about the characteristics of a workload, how they help to make storage management decisions.
- Chapter 4 discusses phase detection methodology, and explains how not all dimensions of workload matter for clustering
- Chapter 5 is model based trace generation, and here auto-correlation studies, and other modeling techniques are discussed
- Chapter 6 discusses details of workload generator
- Chapter 7 is the conclusion and future work

Chapter 2

Background

2.1 Synthetic Workload Generator Benchmarks

Synthetic workload generators and benchmarks play a crucial role in studying the performance of a storage subsystem. Today's complex systems have a lot of interactions between file system, kernel, external devices, network etc., and this makes it a daunting task to account for these minute details and come up with realistic benchmarks and storage system simulators. Additionally, there are some tasks, which are performed asynchronously in real systems [1], and this could not be properly captured in the existing benchmarks as well. There are a lot of synthetic trace generators and storage system benchmarks, and each of these is different, and has different features and simulation capabilities. In current scenario, no single benchmark can suite the needs to perform all possible simulations or benchmarking.

Finding the right parameters that describe a workload or application is a challenging task, and most of the current workload generators focus mainly on some common features like operation type, block size, throughput etc. Design of experiments can help to find the right parameters. If one chose to use design of experiments concept like performing a full factorial exploration of the design space to see which combinations characterize the applications in best way, then it might be very time consuming. Plackett and Burman designs [23] might be a better choice in this case, if one wants to save time and at the same time arrive at such combinational results in a statistically significant way. Synthetic workloads have become popular for a lot of different applications like grid computing [3], database applications, high performance transaction systems etc.

To understand the use cases of synthetic workloads, let us consider one example in which a system in pre-production stage needs to be tested, and one needs to emulate transactions that the system could potentially experience during production phase. Here, say for load testing, the developers need to ensure that they test the system with workload that represents a scenario of high load. Finding the right workload in this situation would help them to perform diagnosis and fix the bugs in their prototype or initial design. Also, scalability and capacity planning is something that should be carefully dealt with in order to minimize design costs, and achieve maximum utility. Again, here using the right trace could help to find the correct parameter settings and capacity requirements.

2.2 Existing Workload Generator Tools

2.2.1 Flexible IO Tester (fio)

Flexible IO tester (fio) is an IO workload generator tool that allows the user to perform workload setups and workload generation based on available parameters [26]. It can issue its IO requests using one of many synchronous and asynchronous IO APIs, and can also use various APIs which allow many IO requests to be issued with a single API call. “fio” allows tuning of file size, file offset, delay between issuing IO requests etc. “fio” has synchronous and asynchronous modes to issue IO requests. In synchronous mode the operating system make sure that any information that is cached in memory has been saved to disk and introduces a significant delay. Fio produces important results like bandwidth, completion latency and IOPS required for benchmarking the storage system. Fio can emulate a variety of queue depths, and allows different access patters like random or sequential reads/writes or a mix of those. It also provides different types of IO engines. Fio also allows bypassing access to the file-system mounted on the device by using the “-direct” parameter. Fio is good for studies where one needs to perform studies on storage subsystem to understand how the system would respond during variation in block sizes, queue depths or varying the delay values.

For example, if one wants to generate:
60% Read and 40% Write; 100% 4k block size, then one can use the command:

```
fio --filename=/dev/sdx --direct=1 --rw=randrw --refill_buffers --norandommap --
randrepeat=0 --ioengine=libaio --bs=4k --rwmixread=60 --iodepth=16 --numjobs=16 --
runtime=60 --group_reporting --name=example
```

2.2.2 Iometer

Iometer is another tool used to produce load on storage subsystem, and to achieve performance testing [16]. One of the good features about Iometer is that it is easy to use, and can be used to quickly determine storage system performance. However, Iometer fails to produce realistic workload conditions for example a mix of multiple block sizes. It is a good tool however for simulating desktop workloads.

Iometer allows creation of random test file sizes, and provides some recommendations for parameter settings. IOPS is one of the common metrics that Iometer uses for determining performance. And similar to fio, one can specify the read write mix to be used. Inspite of the ease of use, this tool significantly lacks the ability to mimic realistic workloads, due to its limited parameter setting capabilities.

2.2.3 Vdbench

Vdbench is an enterprise storage benchmark and workload generator that is implemented in Java, and requires a Java runtime environment [31]. This benchmark allows control over workload parameters such as i/o rate, file sizes, thread count, transfer sizes, volume count, volume skew, read/write ratios, read and write cache hit percentages, random or sequential workloads or any combination of the two. Additionally, one does not need to run this program as the root, as long as the user has read and write access to the output directories.

Vdbench helps the user to define execution parameters, host parameters, storage definition and workload definitions. The workload parameters that it allows users to define are read-write percentages, data transfer size, percentage of skew the workload receives from the total IO rate, access type etc. For example, If one wants to issue random reads of 4K at the rate for 10 ios/second, then once can use the command:

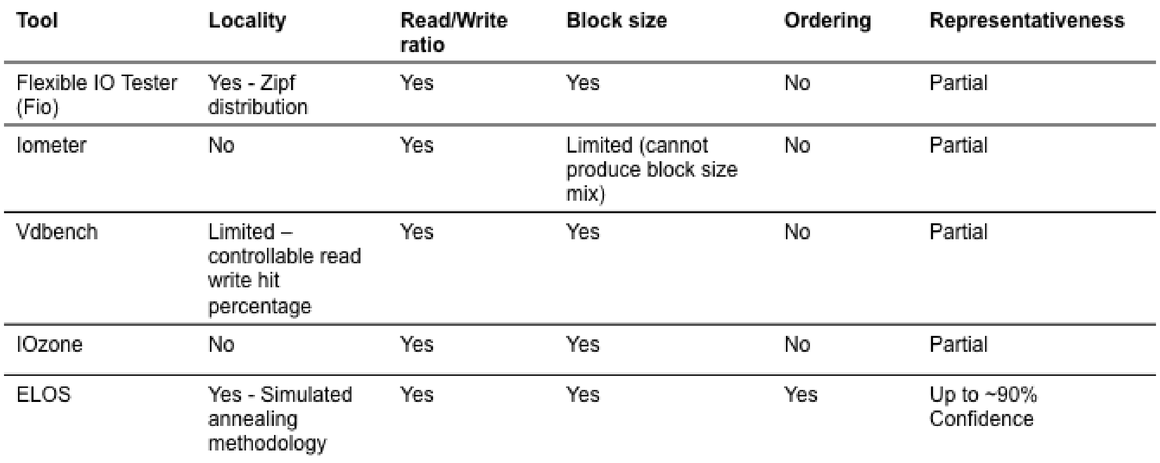
```
sd=sd1,lun=/dev/rdsk/c0t0d0s0
wd=wd1,sd=sd1,xfersize=4096,readpct=100
rd=run1,wd=wd1,iorate=100,elapsed=10,interval=1
```

2.2.4 IOzone

IOzone is another benchmark that could be used for studying disk IO performance [32]. IOzone is compatible with Linux. IOzone allows to benchmark the filesystem performance, and to measure IOs for files of various sizes. With IOzone one can see more detailed information about read, write, and rewrite. IOzone is great at detecting areas where file IO might not be performing as well as expected. This tool helps to gauge the system level performance when one tries to read and write to the storage system.

Table 1 compares some of the existing workload generator tools. From the table we can see that Flexible IO tester use Zipf distribution to achieve desired locality. However, not all locality distributions follow the Zipf law. Hence, this could lead to errors in modeling locality values. Thus, the localities in the final synthetic trace could deviate significantly from what was desired. Vdbench has a parameter to tune the read write hit percentages, and this helps to control the locality to some extent. Some workload generators like Iometer have limited control over block size, and cannot generate block size mix. Ordering is not maintained in most of the workload generators, and are partially representative of the original trace in terms of locality and ordering.

Table 1: Comparing different workload generator tools



Tool	Locality	Read/Write ratio	Block size	Ordering	Representativeness
Flexible IO Tester (Fio)	Yes - Zipf distribution	Yes	Yes	No	Partial
Iometer	No	Yes	Limited (cannot produce block size mix)	No	Partial
Vdbench	Limited – controllable read write hit percentage	Yes	Yes	No	Partial
IOzone	No	Yes	Yes	No	Partial
ELOS	Yes - Simulated annealing methodology	Yes	Yes	Yes	Up to ~90% Confidence

2.3 Related Work

There are various studies done to generate synthetic workloads and a lot of ongoing research to find good representative workloads that are statistically similar to the real application workloads [2]. Bahga et al. have proposed cloud-computing benchmark that could take workload parameters and generate synthetic cloud computing workloads [4]. This benchmark could be used for performance evaluation of cloud computing applications. Ganger [5] in mid 90s realized that generating representative workloads is a challenging task, and indicated that identifying the right workload characteristics is important. He also suggested the framework to validate synthetic workloads. Workload generation is also important in parallel processing systems, and Pfneiszl et al. [6] identified that synthetic workloads don't always match the real systems. Sometimes, it is important to generate workloads that can overload the system. Such workloads are required for load balancing experiments. Mehra et al. [7] describe a workload generator system that could be used in load balancing experiments. Eeckhout et al. [8] in their paper describe how more detailed statistical profiles can be obtained and how the synthetic trace generation mechanism should be designed to generate required benchmark traces. This leads to accurate performance prediction.

Some work has been done to characterize disk IO workloads; Irfan Ahmad [9] performed implementation of disk IO workload characterization using online histograms in a VMware ESX server. One of the main reasons for using synthetic traces is because there is a scarcity of publicly available trace repositories, and especially block IO traces and production server traces are scarce. Kavalanekar et al. [10] provide set of characterizations for the available traces, and list important statistics associated with the traces, and its analysis. They realize that TPC, Filebench, DBT, Postmark, AM-Utils etc could be used for server storage analysis. Gulati et al. [11] studied the impact of storage consolidation in virtualized environments. Riska et al. [12] performed disk drive level workload characterizations, and they observed common characteristics across all traces are in terms of idleness and burstiness.

Not just in drive level, but peta-scale IO workloads also require characterization. It would be hard to store such large workload traces due to memory constraints, and even while development of such systems that see peta-scale workloads, memory management and other such issue become important. Carns et al. [13] developed “Darshan” and IO characterization tool that can shed light on the IO behavior of applications at extreme scale. Delimitrou et al. [19] proposed a novel methodology for synthetic trace generation. They used hierarchical state transition diagrams to probabilistically generate IOs.

2.4 Tracing

2.4.1 Basics

Tracing involves capturing information using a profiling tool, and provide a summary of events that happened. Hence, a trace is a list of events that occurred when a process was executed. Traces are generally chronologically ordered, and show time stamps of each of the events. Trace capturing tools exist, which allow capturing of various traces, and they could be set to collect varying levels of details. Traces could be fed into a real system or fed into a simulation environment. However, it is important to understand that tracing could perturb the system under consideration, and can affect the ongoing processes. Tracing is a good method to perform debug and analysis of the system, and it is important to understand that sometime the trace generated might not provide all information to reach a conclusion. Consider an example, when one is using a timer to collect traces every fixed interval of time, and based on how the system is set up if a high priority interrupt occurs which impairs the trace generation for a short interval of time, or if there was some error in the system, then the trace collected would not be very accurate. Sometimes, multiple smaller events might occur which could not even be captured by tracing, and there are only certain portions of the system stack that could be traced.

2.4.2 IO trace

Traces provide valuable information about workloads encountered by the system. And as mentioned earlier performance of a system highly depends on the parameters set and its optimization is affected by the benchmark or input trace used while performing this process. Having traces is good for performance evaluation of existing systems, however traces are static, and cannot mimic cases when a system undergoes scaling. In such a scenario synthetic workloads prove beneficial. Many of the synthetic benchmarks described before allow user to describe the workloads. Some tools have more tunable parameters than the others, and the user needs to carefully choose to see if it fits their needs, and if the tool allows them to describe the workload with accuracy. One can perform static or dynamic tracing at various layers of the system stack. One example showcasing the difference between static and dynamic tracing and its associated trace points are shown in Figure 1 [15].

Trace-points could be placed in various locations as we can see from figure 1. The main focus here is the block-tracing framework. Block IOs get issued to block IO controllers, which eventually reach the storage system underlying. One of the famous tools for block trace capturing and replaying is blktrace, which can capture lot of different events like IO merges, request queues, IO split or bounces. This tool has three major components: Kernel patch, blktrace and blkparse. Blktrace uses the files from debug file system and hence must have the mount point set up in the mount directory /sys/kernel/debug.

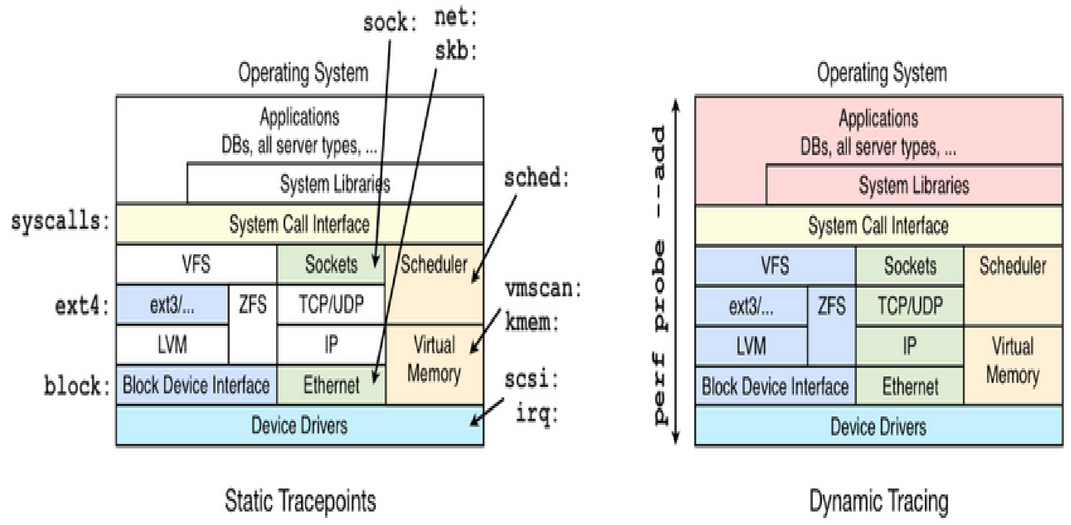


Figure 1: Static versus Dynamic Tracing (Source: [15])

2.5 Workload Profiles

Different usage patterns result in different types of workloads, and hence various applications have their own unique pattern of workload (which is subject to change). Some of the real world workloads patterns as done by a Dell study [28] are shown in the table 2 below, and this was discussed in [5].

Table 2: Workload Descriptions (source: [28])

Application	Block Size in Bytes	% read/write	% Rand/Seq	I/O Performance Metric
Web File Server	4KB, 8KB, 64KB	95%/5%	75%/25%	IOPS
Database Online Transaction Processing (OLTP)	8KB	70%/30%	100%/0%	IOPS
Exchange Email	4KB	67%/33%	100%/0%	IOPS
OS Drive	8KB	70%/30%	100%/0%	IOPS
Decision Support Systems (DSS)	1MB	100%/0%	100%/0%	IOPS
File Server	8KB	90%/10%	75%/25%	IOPS
Video on Demand	512KB	100%/0%	100%/0%	IOPS
We Server Logging	8KB	0%/100%	0%/100%	MBPS
SQL Server Logging	64KB	0%/100%	0%/100%	MBPS
OS Paging	64KB	90%/10%	0%/100%	MBPS
Media Streaming	64KB	98%/2%	0%/100%	MBPS

Chapter 3

Characteristics of a Workload

3.1 Common Block IO Workload Characteristics

Block IO workloads have some unique characteristics that are listed below:

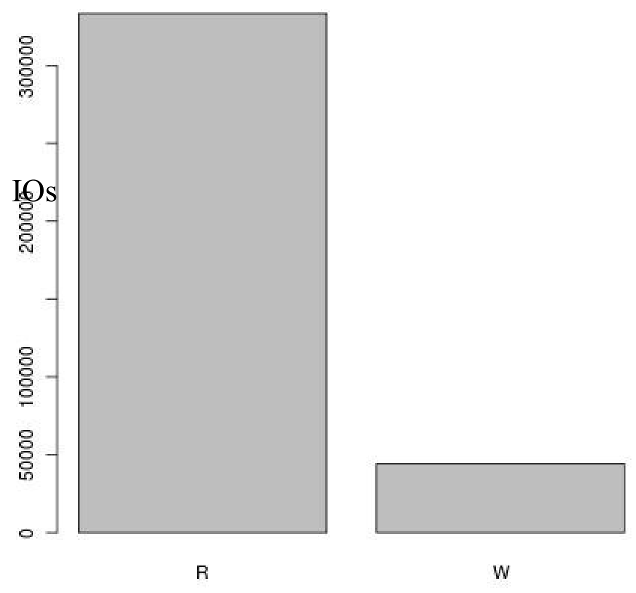
1. Request type
2. Access pattern
3. Request size
4. Arrival time
5. Inter-arrival time
6. Burstiness
7. Block distance
8. Stack distance

Hence, a workload (w) is described completely if there exists a set 'W' such that W = {r, p, s, t, τ, s, b, S}; where r = request type, p= access pattern, l =request size, t=arrival time, τ = inter-arrival time, s = burstiness, b=block distance, S= stack distance. Workload characteristics can be considered as a tuple, containing elements such that one can find all statistical information about the workload using this set. Details of how different workload characteristics and how they help to make storage management decisions are explained below.

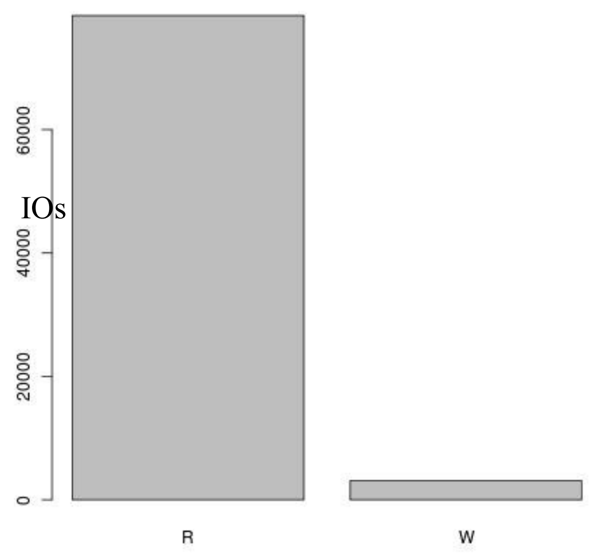
3.1.1 Request type

For Block IO workloads, the two request types are read and write. Read-write ratios are an important characteristic of a workload/application. Understanding this ratio would help to efficiently design and optimize a storage system. Database applications are usually dominated by writes whereas webserver applications are dominated by reads. For other applications this the read/write ratio might vary. Writes in general have higher latency and are more expensive, knowing this ratio is very crucial to optimize the system.

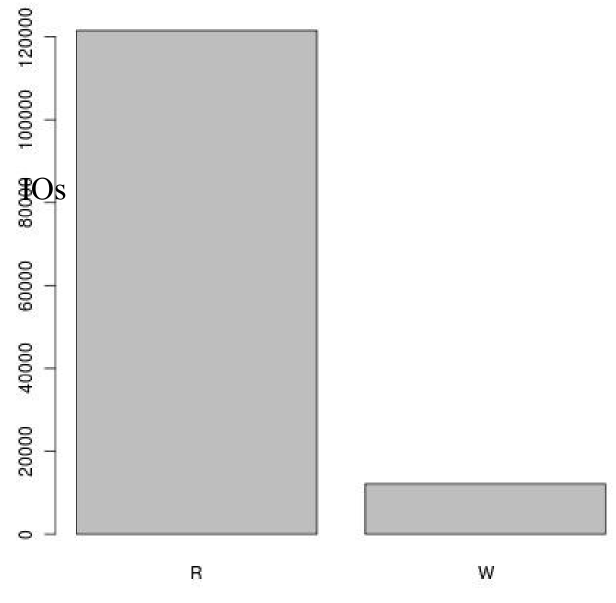
Lot of storage management decisions depend, on this ratio, for example, In applications that demand very low response times, it might not be a good idea to use RAID5 or RAID6 error recovery mechanisms as writes are amplified and on the contrary other mechanisms like incorporating a layer of non-volatile storage to temporarily hold writes or other error recovery mechanisms could be used. Backup systems on the other hand would behave differently, and response times are not very critical in such applications. The read-write ratios of four Seagate customer applications are shown in figure 2 below:



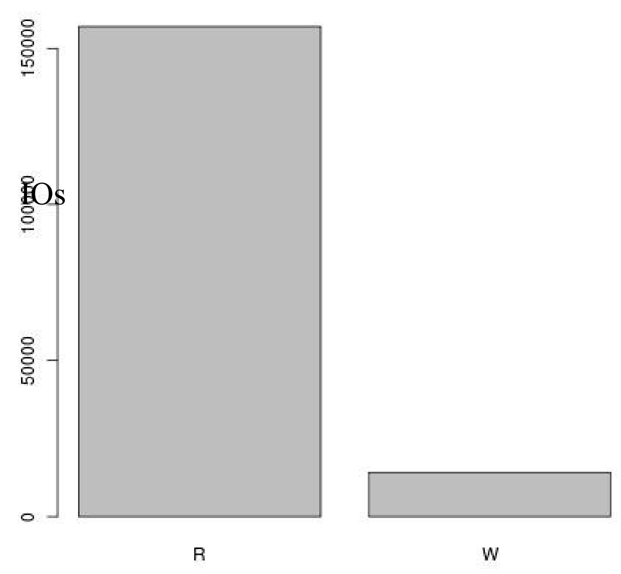
(a)


IOs
0
20000
40000
60000
R
W

(b)


IOs
0
20000
40000
60000
80000
100000
120000
R
W

(c)



(d)

Figure 2: Read write profiles of four Seagate customer traces (a) build (b) home (c) project (d) email. Here x-axis represents read/write, and y-axis represents number of IOs. Here R= reads, and W= writes.

3.1.2 Access pattern (Sequential or Random)

Some applications have high spatial locality and have a sequential access behavior (for example: a logging application where writes are sequential and also backup server applications). On the other hand, there might be applications that have a random access pattern (for example- database type applications). In case an application is highly random, then the underlying storage device should support this. Access patterns help in making storage management decisions regarding data placement (in HDDs, SSDs, PCMs etc.) and retrieval. Solid State Drives have a very high random read performance, so using SSDs in the underlying storage system would be an intelligent decision. Also, SCSI devices and SAS devices are faster than SATA devices. Intelligent data placement would also help convert random reads to sequential reads. For example, an application writes randomly to a disk, then the mappings could be changed, and the apparently random workloads could be stored sequentially in the disk. This way during reads, the response time would be faster. However, Solid State Drives have bad large write performance, as large amount of data migrations are required. Another important thing to remember is that large sequential reads are also sometimes susceptible to fragmentation (internal and external); fragmentation here could refer to logical/physical disk fragmentation (data might be present in different disks) and when the logical order of data placement does not follow physical order (which could happen in case of Solid State Drives due to different logical to physical mapping). Phase Change Memories (PCMs) can be used in cases when data retention is very crucial and in spite of any interruptions like power loss, data could still be recovered. Hence, knowing the access pattern is very crucial.

3.1.3 Request size and data set size

Request size is the amount of bytes requested by an application from the storage or memory. Some applications operate on large data sets, whereas some others deal with smaller data sets. This property of the application is important to know which considering which caching scheme to use, and to characterize an application. Consider an application whose data set is small enough such that it fits in the cache, or maybe consider another scenario where the application writer changed the application to intelligently fit all the data in the cache and modularized the application. In such a scenario, the read and write performance would be very high, and the storage system should be robust enough to have a fast write or read (to and from the cache). Prefetching could also be effectively used in such a scenario if the application shows locality. Another important feature that a storage system developer might consider is the temporal locality in workloads. In many cases, the application might repeatedly access certain memory locations, and in such cases it would be intelligent to keep the data item in the cache for some time before eviction. Adaptive replacement cache would make use of this property and effectively combine the benefits of LRU and MFU policies.

3.1.4 Arrival time

The time at which a request reaches the disk IO subsystem is referred to as the arrival time. The rate at which requests arrive at the disk subsystem is the arrival rate.

CDF of different types of job arrival times are shown in figure 3.

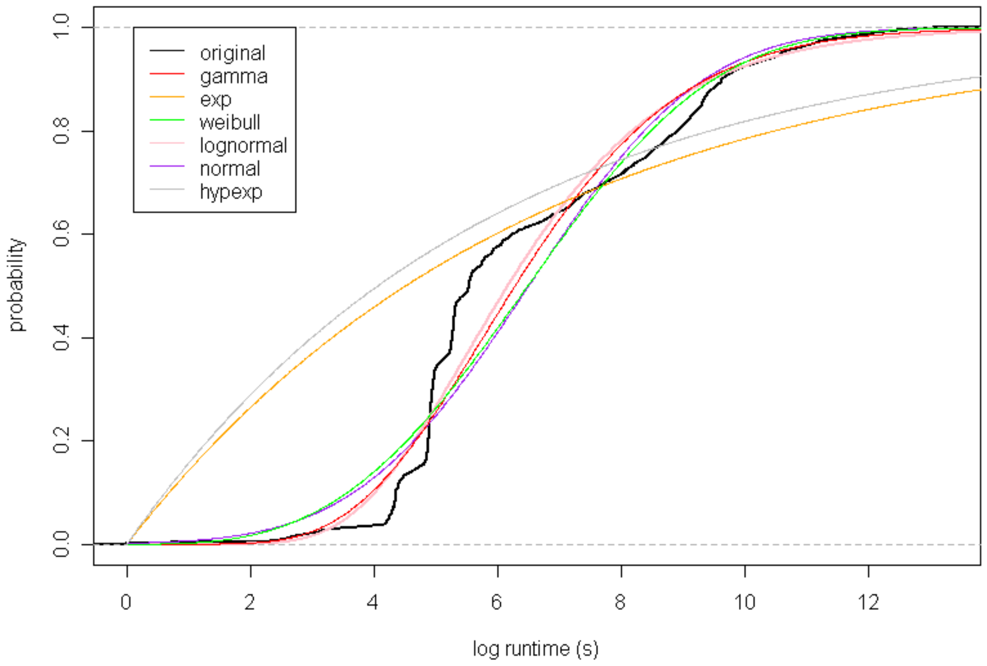


Figure 3: CDF of job runtimes (source: [14])

3.1.5 Inter-arrival time

The time between two consecutive requests is called as the inter-arrival time, and it is a workload parameter. Inter-arrival times are usually modeled by Poisson distributions. If X is the random variable indicating the time between two requests, then let F(t) be a Poission process, and λ be the rate parameter then:

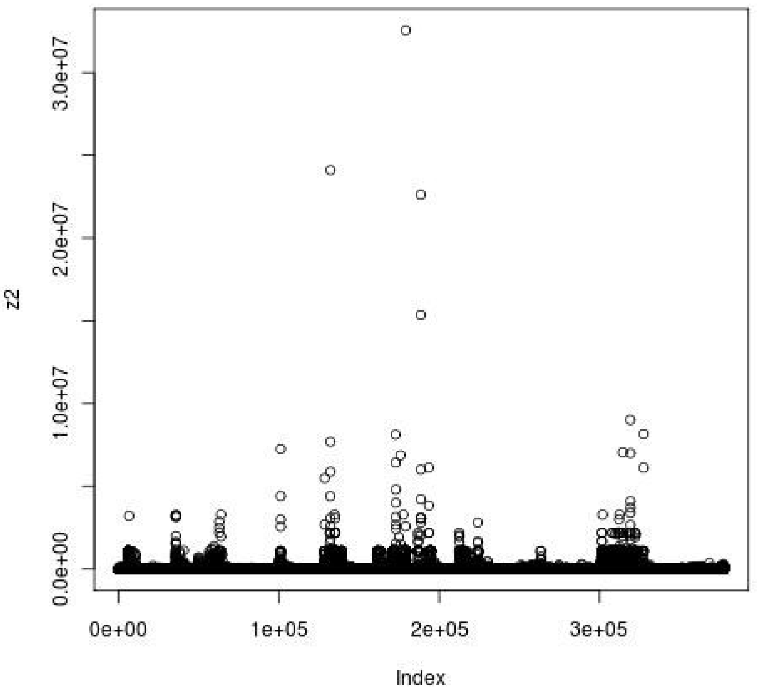
$$\mathbf{F_x(t) = P(X <= t)}$$

$$\mathbf{F_x(t) + e^{-\lambda t} = 1}$$

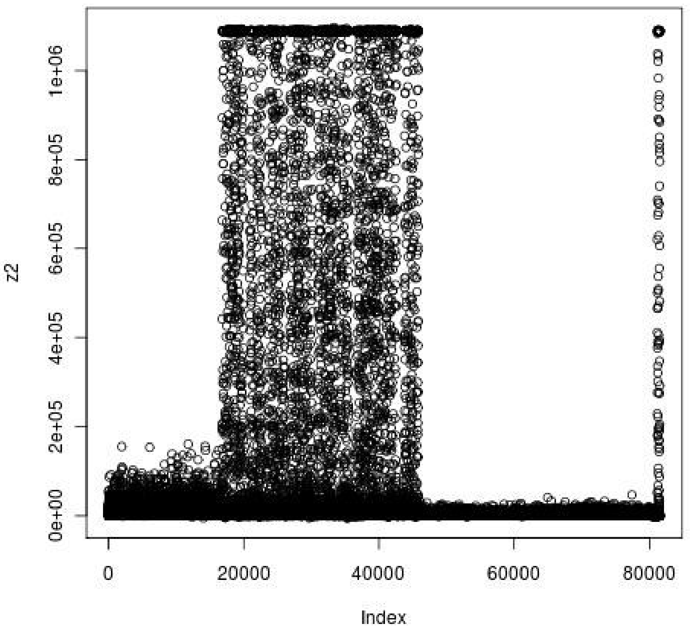
$$\mathbf{F_x(t) = 1 - e^{-\lambda t}}$$

$$\mathbf{f_x(t) = d/dt\ (F_x(t)) = \lambda e^{\lambda t}}$$

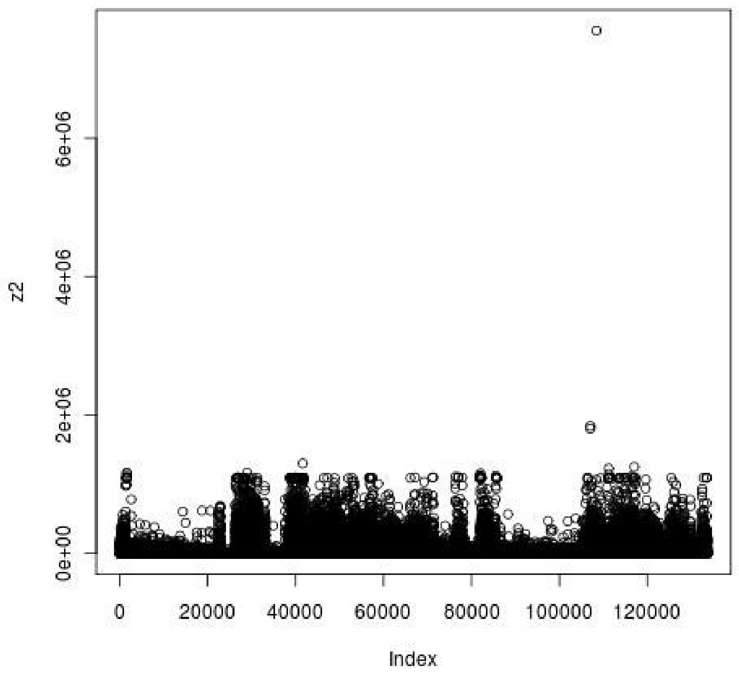
Poisson distributions are used when arrival patterns are unknown, and Poisson process is memory-less; hence if no arrival has occurred by time *t*, the distribution of the remaining waiting time is the same as it was originally. But inter-arrival times can also take on other types of distributions. Inter-arrival times of four of Seagate customer traces were plotted, which could be seen from figure 4.



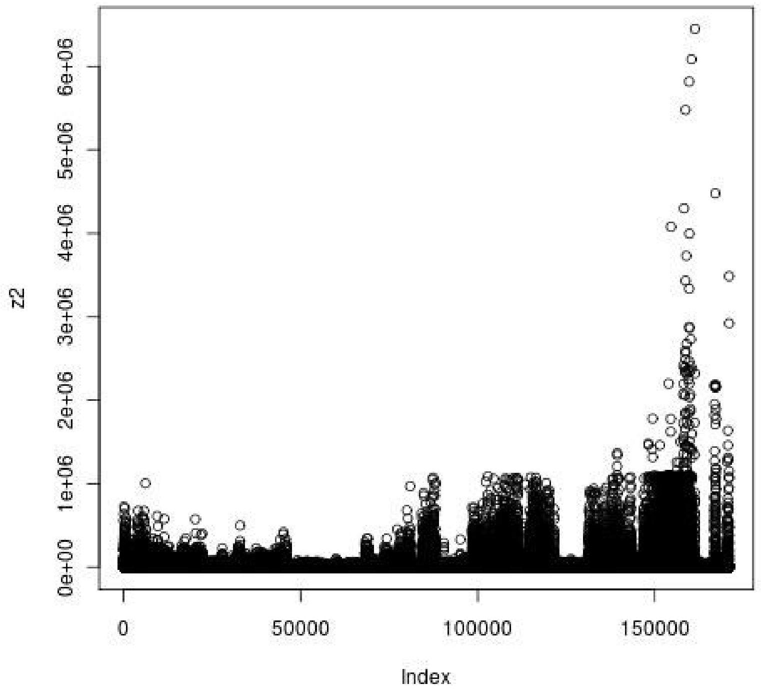
(a)


1e+06
8e+05
6e+05
4e+05
2e+05
0e+00
z2
0
20000
40000
60000
80000
Index

(b)


6e+06
4e+06
2e+06
0e+00
z2
0
20000
40000
60000
80000
100000
120000
Index

(c)



(d)

Figure 4: Inter-arrival times of four Seagate customer block traces. (a) build trace (b) home (c) project (d) email. Here the y-axis represents the inter arrival time, and x-axis represents the trace elements.

In figure 4, the higher the y-axis value (inter-arrival time), the longer is the time between issuing two IOs. And additionally, increasing x-axis indicates increasing time. Here, Trace 1, shows phases of longer and shorter inter-arrival times, whereas Trace 2 shows a long phase of low inter-arrival times followed by a long phase of high inter-arrival times. Whereas, Trace 3 shows bursty behavior. Trace 4 shows a phase of high bursts, and then followed by an idler phase as time progresses.

3.1.6 Burstiness

Depending on the application, the storage system might see a bunch of IOs in certain times of the day as compared to other times. For example, in case of share trading applications, the time during opening and closing would see most IOs, and the IO activity would slow down in the afternoon and one might not see any activity during the night. In such cases, it is very important to identify this behavior to properly scale the storage system or fulfill requests without failing. The period where the storage system sees peak IOs is the bottleneck or the phase that one needs to worry about. During times like night or during less activity, background activities like backup are scheduled. During peak times if lots of reads are issued, then one need to ensure that the cache size is large enough and proper caching policies in the controller are used. Whereas if the period of peak activity sees mostly writes, then then proper writing policies and hardware to ensure failure tolerance should be used (for example using a layer of non-volatile memory like PCM as cache to support or better handle latency and failure during power loss). Entropy could be used to study burstiness in workloads. Section 5.2 has detailed analysis.

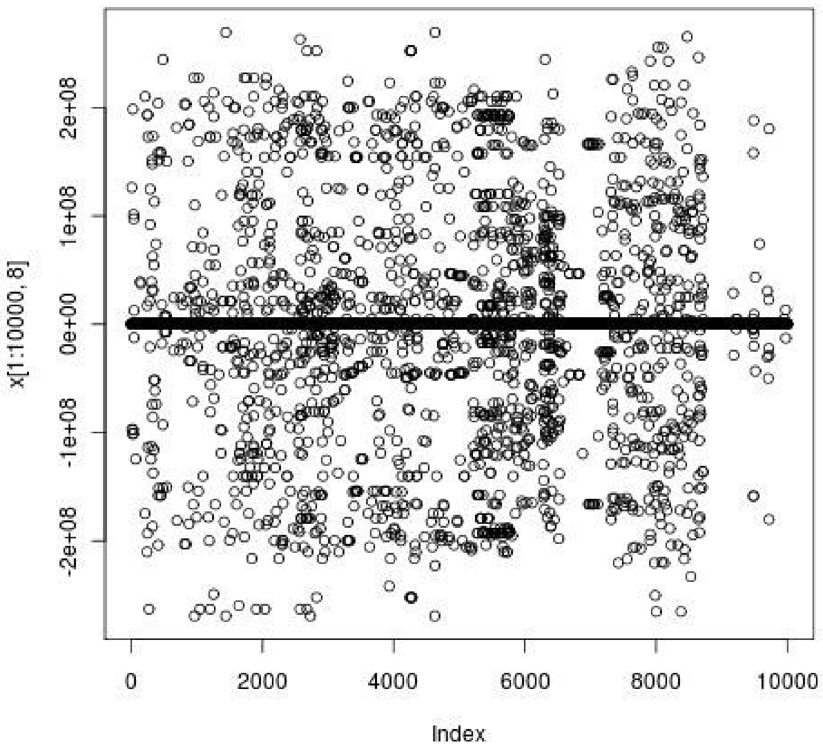
3.1.7 Block distance (Spatial Locality)

Block distance in case of Block IO workloads is the distance between two consecutive LBNs accessed. Spatial locality states that the likelihood of accessing a data is higher if the data near it was referenced just before. High spatial locality generally implies good performance. SSDs have a logical to physical mapping table for data layout, and hence spatial locality here might not mean higher performance. However pre-fetching algorithms generally tend to identify sequential access patterns nicely, and tend to improve performance this way. To understand the calculation of block distances one can refer to figure 5. Note here, the negative sign for block distance just means that the next block number accessed was smaller than the current access. The absolute value of the block distance is more significant, as it indicates how close or far the accesses were. Also, the physical layout of the disk drive or SSD would determine the performance impact of various block distances.

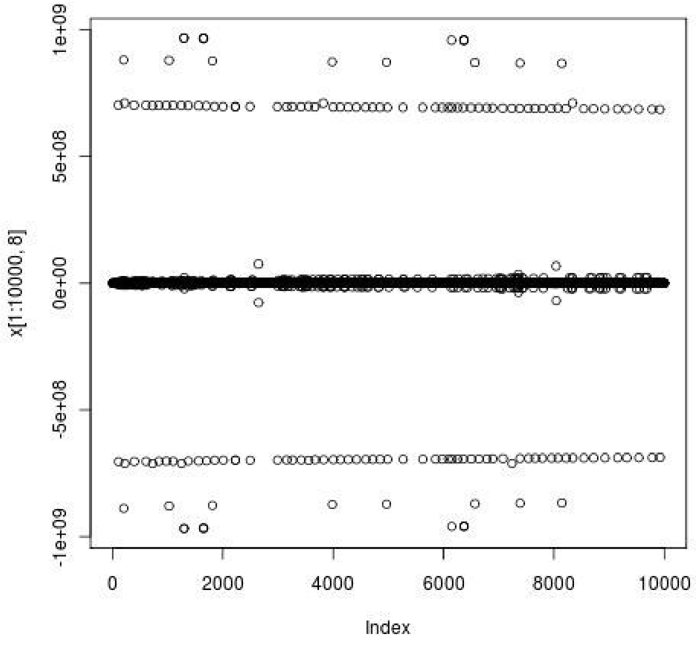
	A	B	C	D
Block number	**100**	**1000**	**1200**	**500**
Block distance	**100**	**900**	**200**	**-700**

Figure 5: Calculation of block distance is shown. Let A, B, C and D be the input requests. Then the block distance is given by the difference of two consecutive LBN numbers. Here, the negative value of block distance just means that the next block number accessed was smaller than the current one.

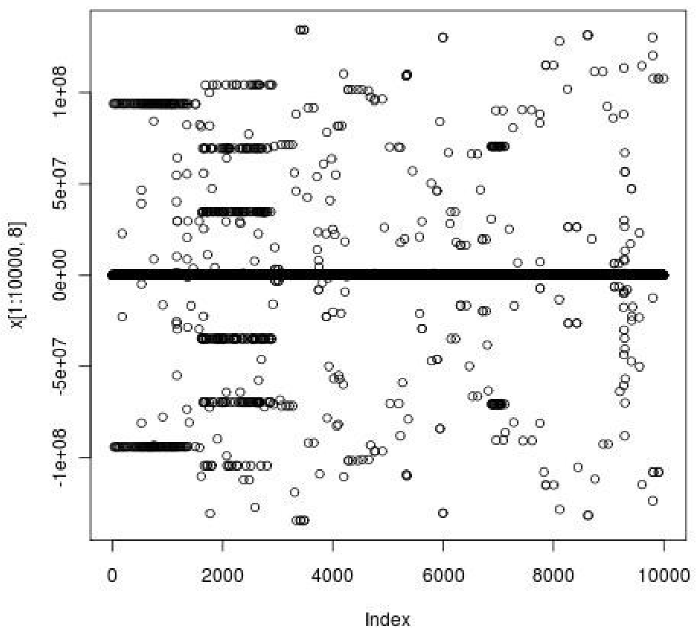
The block distances of the four Seagate customer traces used are shown in figure 6. The more the spread along y-axis, the more is the randomness in the trace access pattern. For example, home trace seems to have a more regular access pattern as compared to build trace.



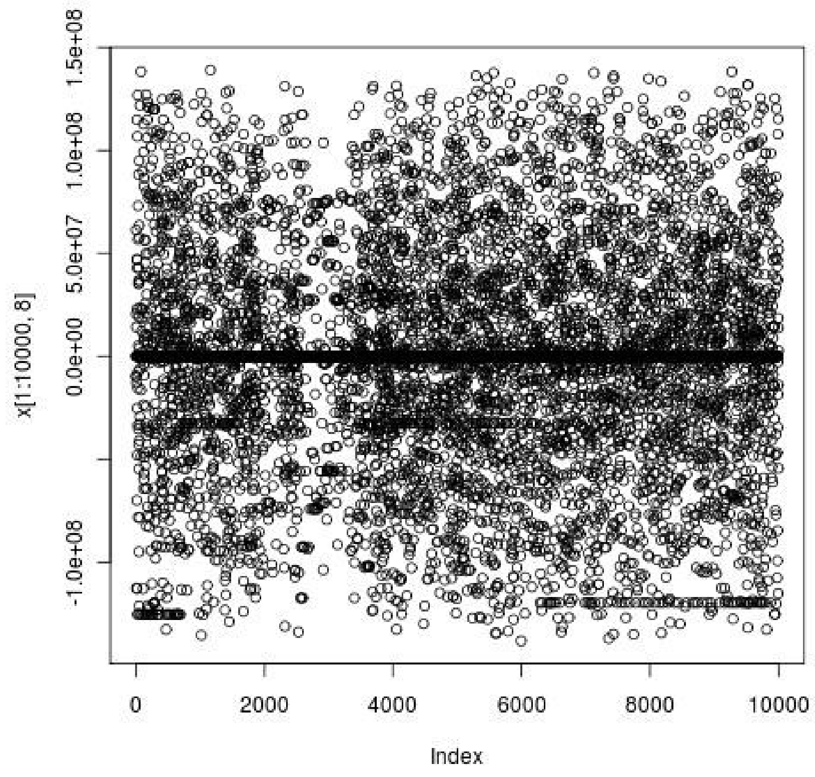
(a)


1e+09
5e+08
0e+00
-5e+08
-1e+09
x[1:10000, 8]
0
2000
4000
6000
8000
10000
Index

(b)


1e+08
5e+07
0e+00
-5e+07
-1e+08
x[1:10000, 8]
0
2000
4000
6000
8000
10000
Index

(c)

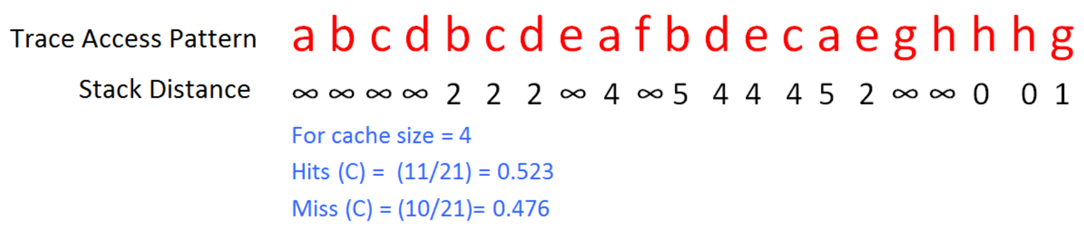


(d)

Figure 6: Block distances of four Seagate customer block traces. (a) build trace (b) home (c) project (d) email. Here the y-axis represents the block distances, and x-axis represents the trace elements.

3.1.8 Stack distance:

Memory is one of the major bottlenecks in today's technology and caches play a major role in eliminating the latency observed while accessing memory. Workload of an application highly dictates how the system would perform, and the accesses pattern of the application determines the way cache will be used. Temporal locality is a measure of when a particular memory location will be referenced in the future. Stack distance could be efficiently used to calculate the temporal locality of a trace. Stack distance or re-use distance is the number of unique elements accessed between two consecutive accesses to the same element [18]. Stack distance helps to quantify the temporal locality present in a trace. The temporal locality in an application helps to study the cache behavior, given the cache size, replacement policy and associativity of the cache. For a LRU cache, cache misses will only happen if stack distance is greater than or equal to the cache size. Refer to figure 7 for an example of stack distance calculation. Here, stack distance of infinity means that there was no previous reference of the trace element. Cache hits and misses could be calculated for an LRU cache, as shown in the figure 7.



Trace Access Pattern	a	b	c	d	b	c	d	e	a	f	b	d	e	c	a	e	g	h	h	h	g
Stack Distance	∞	∞	∞	∞	2	2	2	∞	4	∞	5	4	4	4	5	2	∞	∞	0	0	1

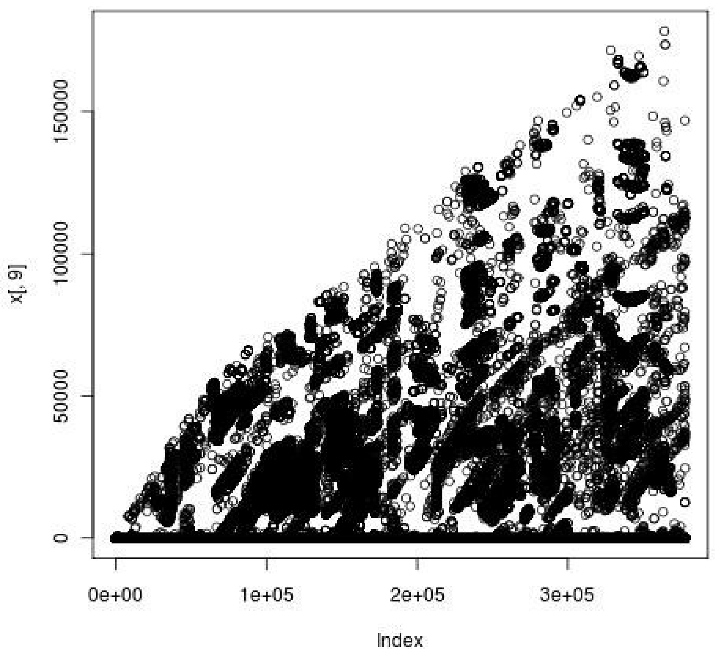
For cache size = 4

Hits (C) = (11/21) = 0.523

Miss (C) = (10/21)= 0.476

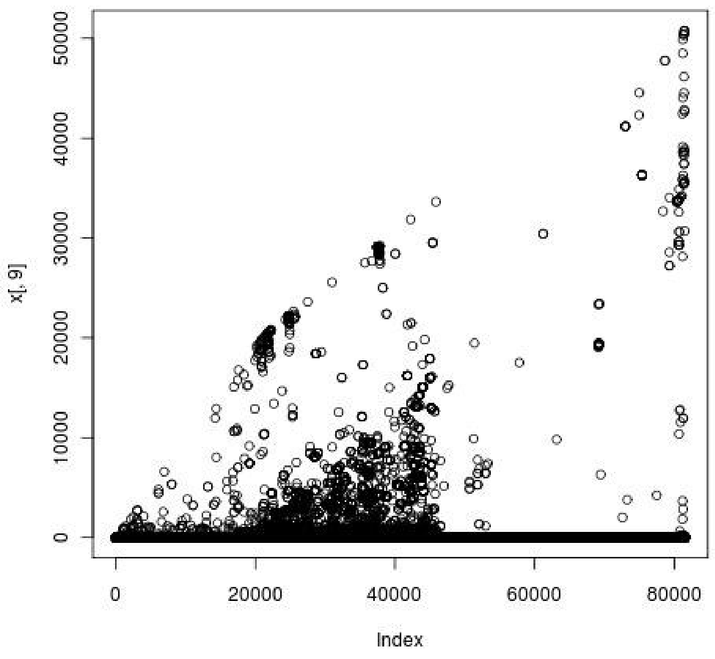
Figure 7: Stack distance calculation for a sample trace access pattern. The hits and misses calculated here are for a LRU cache.

The stack distance values of the four Seagate customer traces were plotted, and the results are shown in figure 8. We can see that trace 1 and trace 4 have a lot of temporal reuse. Trace 2 has low stack distance values, indicating that it would have good cache performance for LRU caches, given that the data set is small for the applications. Trace 3 shows a pattern in stack distances indicating that there are some processes that happen at regular intervals of time.



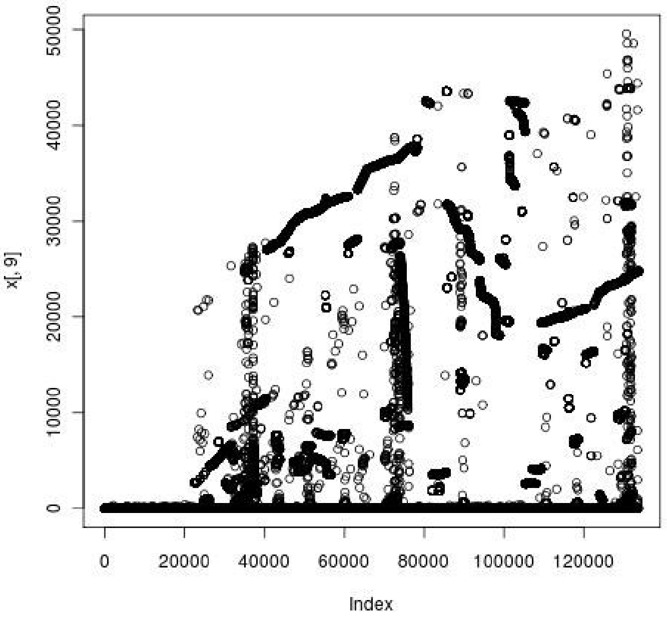
x[, 9]
0
50000
100000
150000
0e+00
1e+05
2e+05
3e+05
Index

(a)

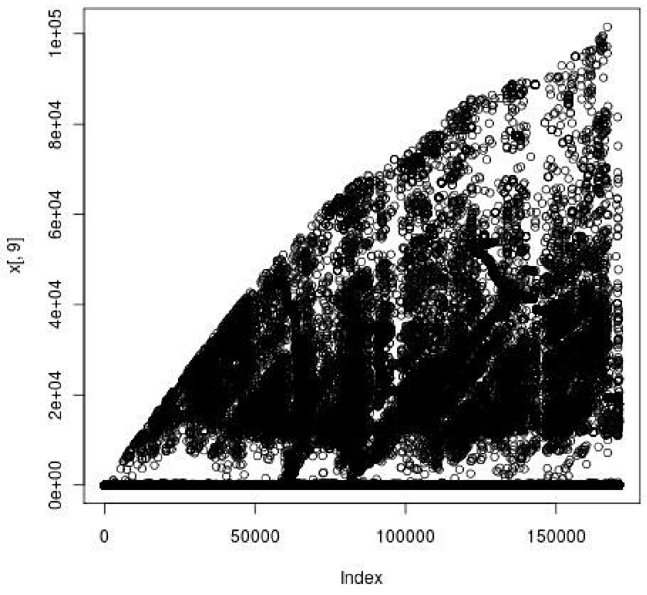


x[, 9]
0
10000
20000
30000
40000
50000
0
20000
40000
60000
80000
Index

(b)



(c)



(d)

Figure 8: Stack distances of four Seagate customer block traces. (a) build trace (b) home (c) project (d) email. Here the y-axis represents the stack distances, and x-axis represents the trace elements.

(i) Traditional Stack Distance Calculation:

Traditionally, stack distance as the name suggests, are calculated using LRU stacks. The time complexity of this algorithm is O(n^2). Initially, as stack is initialized, and as we iterate through an input IO sequence, the element is checked if it is already present in the stack, if it does exist, then it is moved to the top of the stack. Otherwise, the element is just inserted into the top of the stack. The stack distance of the element, if it already exists in the stack would be the number of places it had to move to reach the top of the stack. Otherwise, the stack distance would be infinity. To better understand, this consider, the example shown in figure 9, below. We can see here that the input sequence is given by "a b c d b", and elements are inserted into the top of the stack until the fourth access to d, after this point, the next access is to element b which is already present in the stack. Hence the stack distance of b in position 5 is "2".

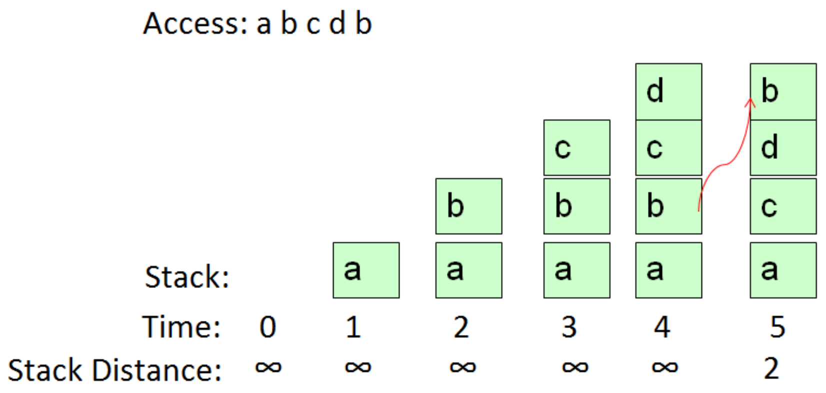


Figure 9: Traditional LRU stack distance calculation

(ii) ***Interval Based Stack Distance Calculation***:

Traditional stack distance calculation algorithm is O(n^2) algorithm, as discussed before. There are various O(nlgn) algorithms proposed by researchers. Here, one such nlgn algorithm is proposed. This algorithm is easy to use, and is highly parallelizable. In this algorithm, consider a trace say A, with n elements.

Let H be a hash table data structure. All the elements of the hash table are initialized to infinity (which was represented as '-1'). The algorithm starts by going through every element of the trace and finding if the trace element was already observed in the hash table or not, and every time this check is done, the value of the hash value associated is set to be the current iteration index. This way, if an element is repeated then its value will be the previously seen index. This can help us calculate the stack distance of the element. Additionally, if there are repeats then one should add the current element to a list of elements sorted in terms (say L) of increasing end times. Another list is created in the terms of increasing start times (L_r). Then AVL tree is constructed from the list of increasing end times. Stack distance can be calculated now by iterating through every element of L_r. Then stack distance is: *Current Index – Previous index from hash table – number of elements in left sub-tree of AVL Tree – 1,* if repeat exists. Else Stack distance is infinity (in this case the index will not exist in the list L). After calculating the stack distance from the list, the current element is removed from the tree, and the AVL tree is rebalanced. AVL tree operations are of the order of lgn and for n elements, the time complexity should be nlgn. And if there are N repeats in the trace with n elements, and N could be such that N<<n, so the complexity is further reduced. The details of the stack distance calculation algorithm are given below, and in figure 10.

Stack Distance Calculation Methodology:

Let $A = \{a_1, a_2, a_3 \ldots a_n\}$ be elements of a trace

Let D_i be the stack distance of element a_i

Let $S_{im} = \{a_i, a_j, a_k, \ldots a_m \mid a_i = a_j = a_k, \ldots a_m \wedge i \neq j \neq k \ldots \neq m\}$

Let $|S_{im}|$ is the repeat number (cardinality)

Let H be the Hash table

$H = \{h_{a1} \ldots h_{an} \mid h_{ai}$ is hash lookup for element $a_i\}$

if D_i is stack distance for element ai then

$D_i = i - h_i - |s_{ihi}| - 1$ if $h \geq 0$

$D_i = \infty$

```
Let L = list of intervals
for each $a_i$ in A
          If $h_i \geq 0$
              add an interval i-hi to L
          update H to i
Create $L_r$ = list of intervals sorted with increasing start times
Create a AVL tree (T) from L based on increasing end times
for each element in $L_r$
          Stack distance = i – hi – number of elements in left sub-tree (of AVL T) – 1
          Remove current element from tree and rebalance
```

Figure 10: Algorithm for stack distance calculation

Chapter 4
Phase Detection

Identification of phase boundaries is important to reuse important information associated with the phase. This information could be the configuration, initialization values, resource utilization metrics, failure rates or metrics, error correction needs, storage needs, bandwidth requirements, operation mix, QoS requirements, system load information etc. There are various methodologies that could be used for phase detection, and some of the approaches used are shown in the following subsections.

4.1 Approaches

4.1.1 ***Clustering***: Clustering could be used to divide an input trace into different phases. Phase boundaries could be described by the cluster boundaries. The behavior of the cluster could be used to perform optimizations. Clusters might be repeating after a particular time, and it helps to study the pattern of certain activities portrayed by the application trace. Clusters are easy to understand using a plot or by looking at the clusters visually. Once we find the clusters, then it becomes easier to find a representative workload using the original trace. This is important, as trace storage could be expensive. The enterprise storage traces could be as huge as few gigabytes or terabytes. In that case finding a representative workload for a subset of that trace becomes tricky. Clustering also makes outlier detection easy. Outliers are defined as clusters which show some anomalous behavior, or which are far away from other clusters. Sometimes certain clusters might represent error prone IOs, and in this case, identifying the properties of the cluster becomes important. Clustering is used in various fields of science and engineering to group data points similar to each other. Cluster sizes can be defined, and by iteration optimal number of clusters can be obtained. There are a lot of clustering algorithms, and k-means is one of the simplest unsupervised algorithms that compute clusters without prior knowledge about the data points. This algorithm initially choses k centroids associated with k clusters, and tries to assign data points to the cluster with minimum distance from its centroid. The algorithm tries to minimize the objective function.

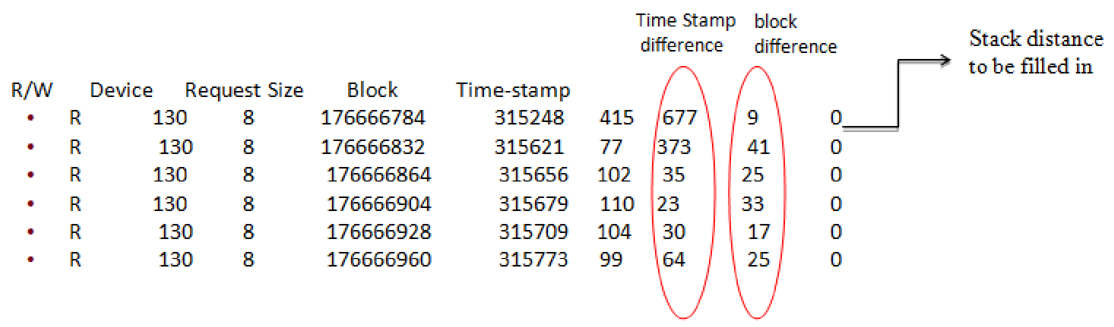
$$J = \sum_{j=1}^{k} \sum_{i=1}^{n} \left\| x_i^{(j)} - c_j \right\|^2$$

where, $\left\| x_i^{(j)} - c_j \right\|^2$ is the square of distance between the data point $x_i^{(j)}$ and the cluster centre c_j, for n data points from their respective cluster centers.

If one wants to do phase detection and identification using k-means clustering then one needs to follow the steps for experimentation and validation described below:

1. Create clusters for a training set (initial trace data)
2. Store the cluster information created using this training set
3. Use rest of the trace as validation set – and modify the clusters when necessary
4. As new data comes in, it is easy to cluster them
5. We can detect phases in workloads like backup phase, garbage collection, anomalies etc.

K-means clustering was done on Seagate customer traces and MSR Traces. The essential trace elements are shown below:



R/W		Device	Request Size	Block	Time-stamp		Time Stamp difference	block difference	
•	R	130	8	176666784	315248	415	677	9	0
•	R	130	8	176666832	315621	77	373	41	0
•	R	130	8	176666864	315656	102	35	25	0
•	R	130	8	176666904	315679	110	23	33	0
•	R	130	8	176666928	315709	104	30	17	0
•	R	130	8	176666960	315773	99	64	25	0

Stack distance to be filled in

This trace was modified to include columns for stack distance and block distance. One of the challenges was to come up with a good metric to classify the input IO data points. For example, one can use block size or requested LBN number to do the clustering. It is also a good idea to come up with a metric that accounts for multiple metrics, and takes a weighted average for classification. In some cases, only one-dimensional or two-dimensional clustering might be required to observe useful characteristics. Figure 11 and 12 show one such example. Here, we can see clear divisions in the input data points. Since, only one dimension was used, the demarcation seen is clear. As expected, same input trace, when cluster using different metrics, produce different clusters. To further understand the behavior, clustering of MSR Cambridge hm1 trace was done. Here, going from Figure 13 to Figure 14, we can see how cluster behavior changes. Figure 13, we clustered using one dimension, whereas Figure 14 was clustered using three dimensions block number, request size and inter-arrival time.

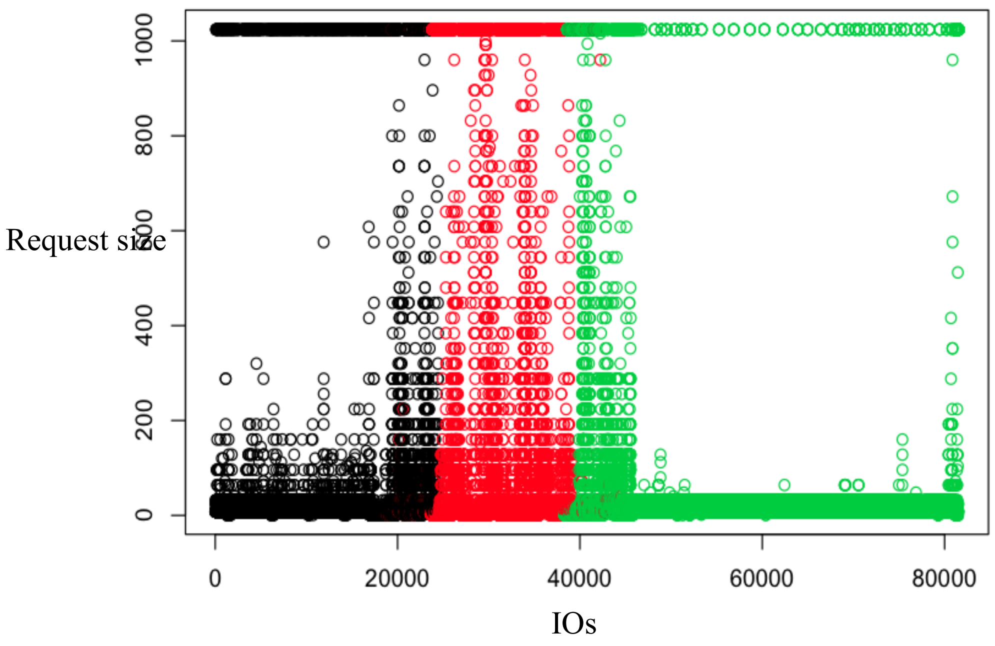


Figure 11: Seagate customer trace classified mainly in terms of request size (highest weight). Three clusters were requested. Here the black data points represent clusters with high request sizes. Red data points represent data point with request sizes in the mid-level, mostly between 200 to 1000. The green clusters represent data points with low request sizes.

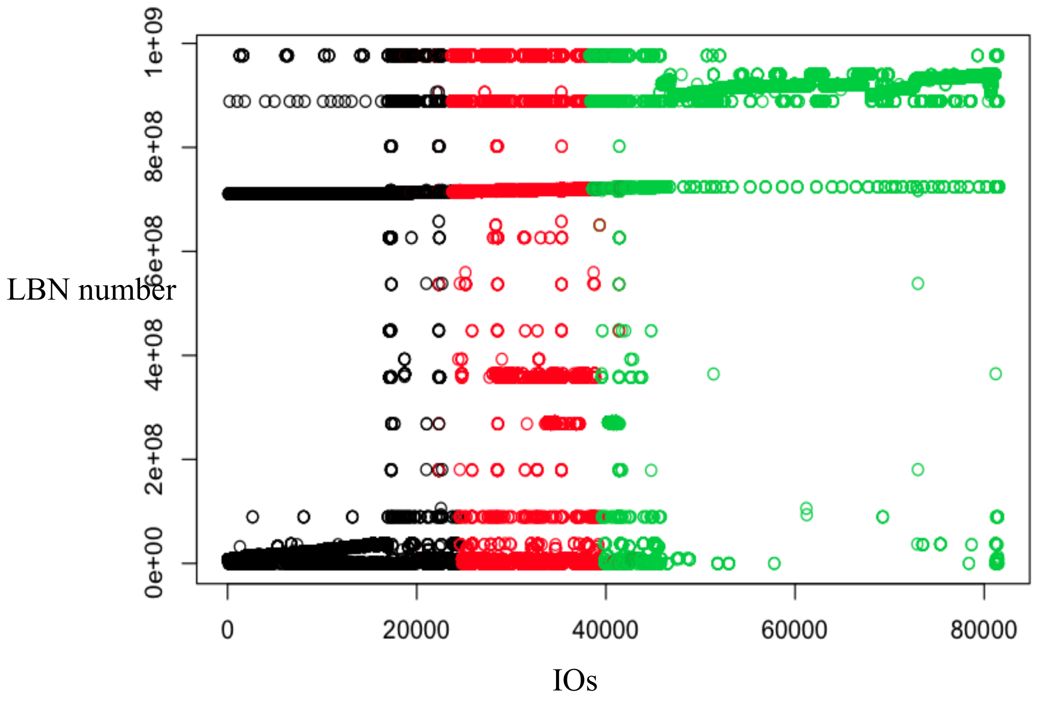


Figure 12: Seagate customer trace classified in terms of requested LBN. Three clusters were requested. Here we can see that black clusters represent data points with low LBN numbers; red clusters represent data points with LBN numbers in the mid-range, and green clusters represent data points with high LBN numbers.

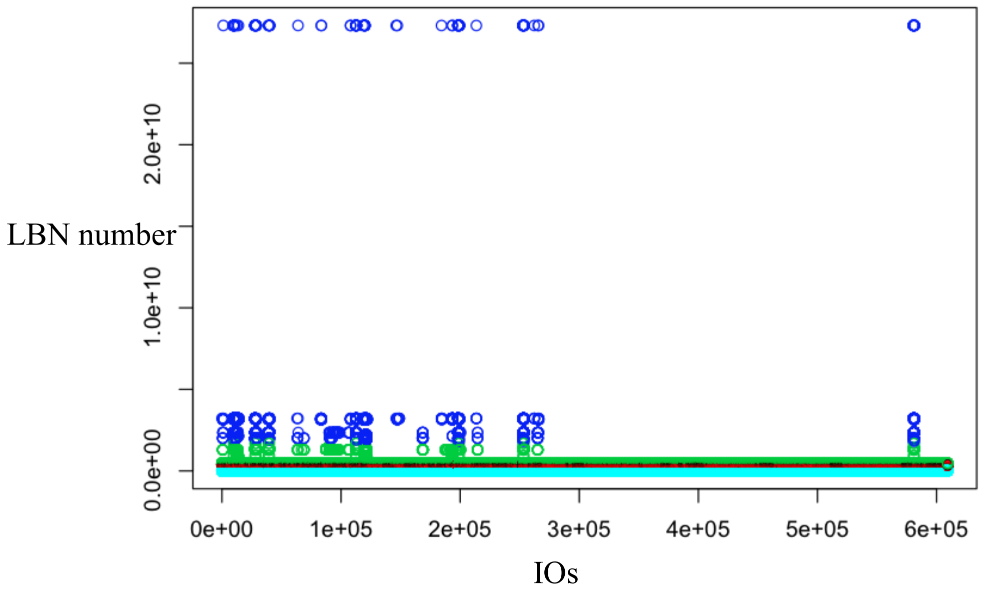


Figure 13: MSR Cambridge trace hm_1 classified in terms of LBN numbers

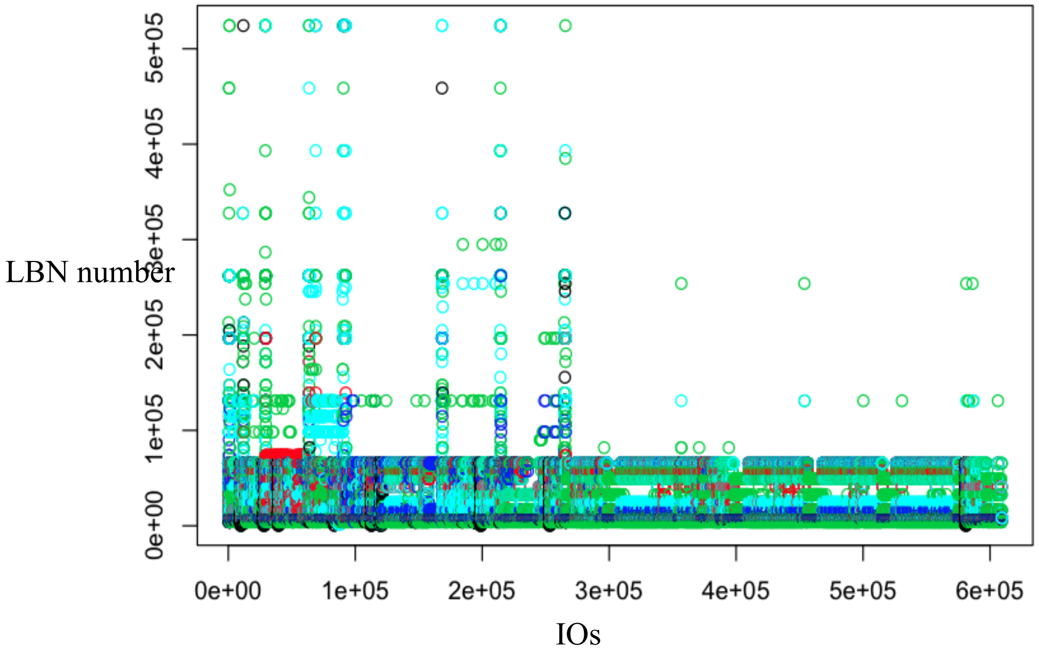
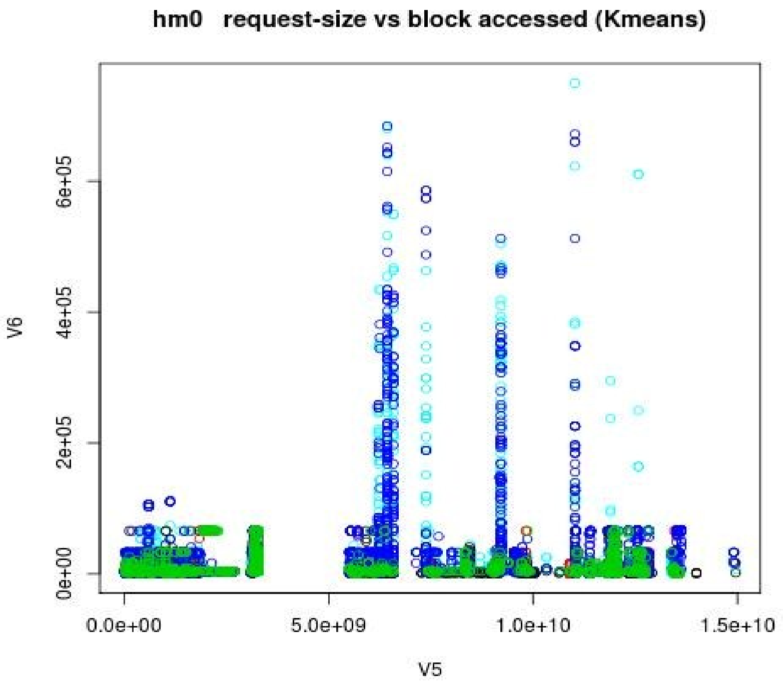
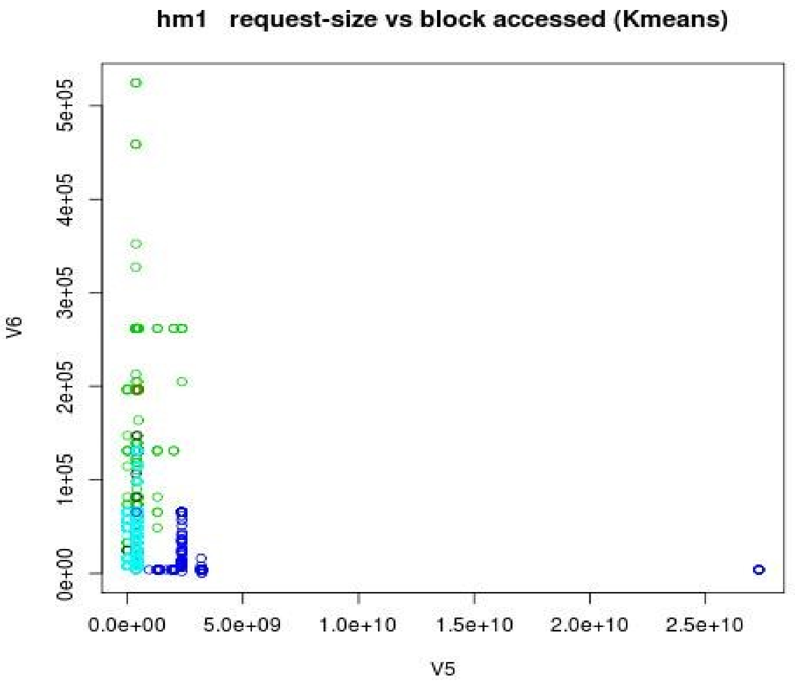


Figure 14: MSR Cambridge hm_1 trace classified in terms of LBN number, request size and inter-arrival time

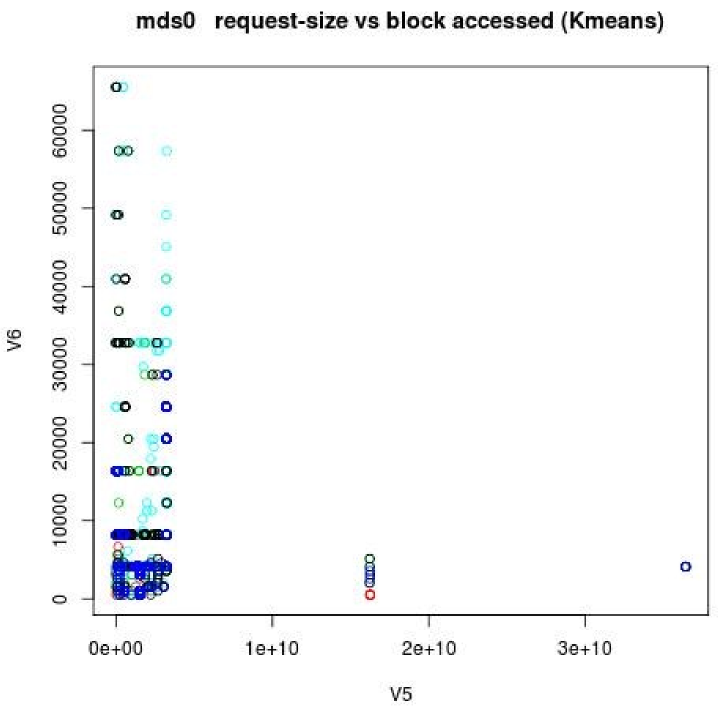
MSR Cambridge traces were then clustered using three dimensions block size, request size and inter-arrival time. The 2 dimensional plots can be seen in Figure 15.


hm0 request-size vs block accessed (Kmeans)
V6
V5
0e+00
2e+05
4e+05
6e+05
0.0e+00
5.0e+09
1.0e+10
1.5e+10

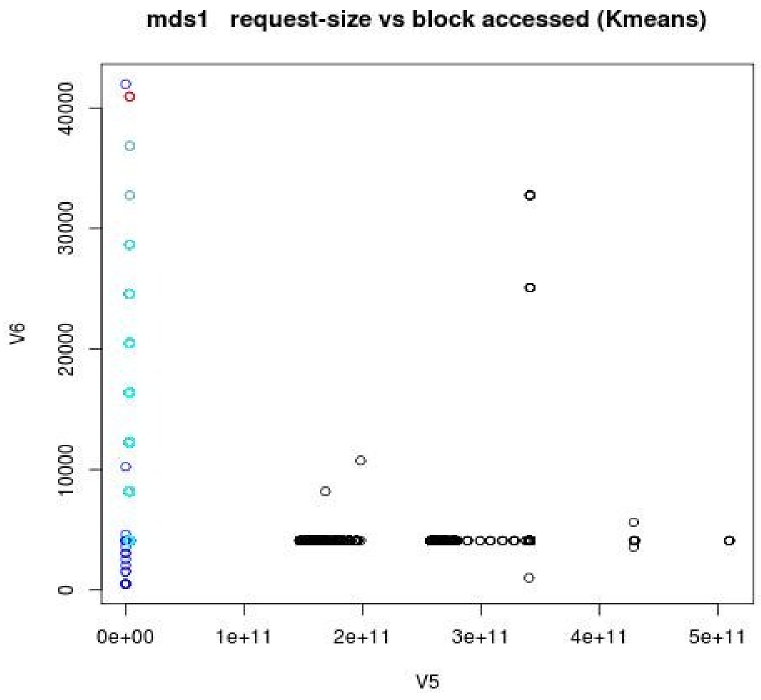
(a)


hm1 request-size vs block accessed (Kmeans)
V6
V5
0e+00
1e+05
2e+05
3e+05
4e+05
5e+05
0.0e+00
5.0e+09
1.0e+10
1.5e+10
2.0e+10
2.5e+10

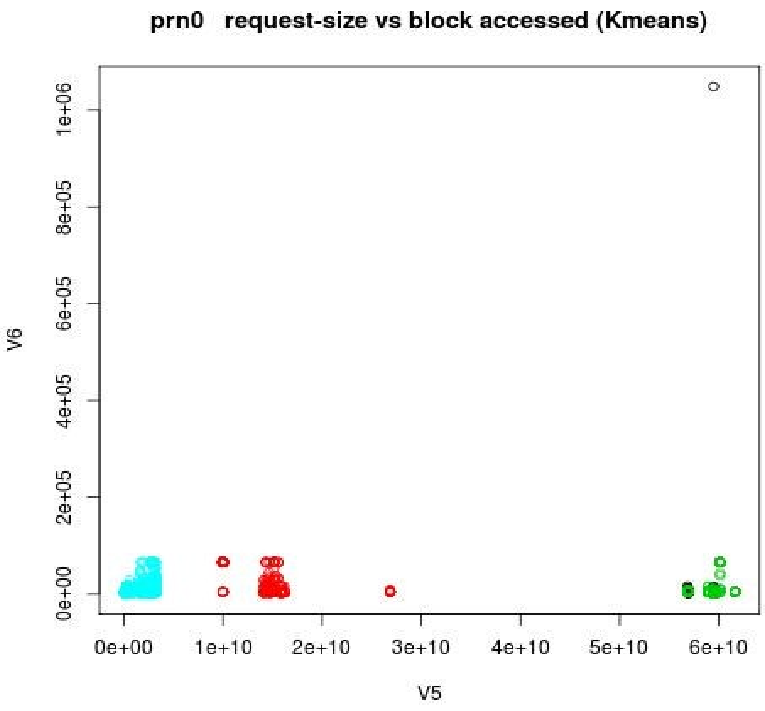
(b)


mds0 request-size vs block accessed (Kmeans)
V6
V5
0
10000
20000
30000
40000
50000
60000
0e+00
1e+10
2e+10
3e+10

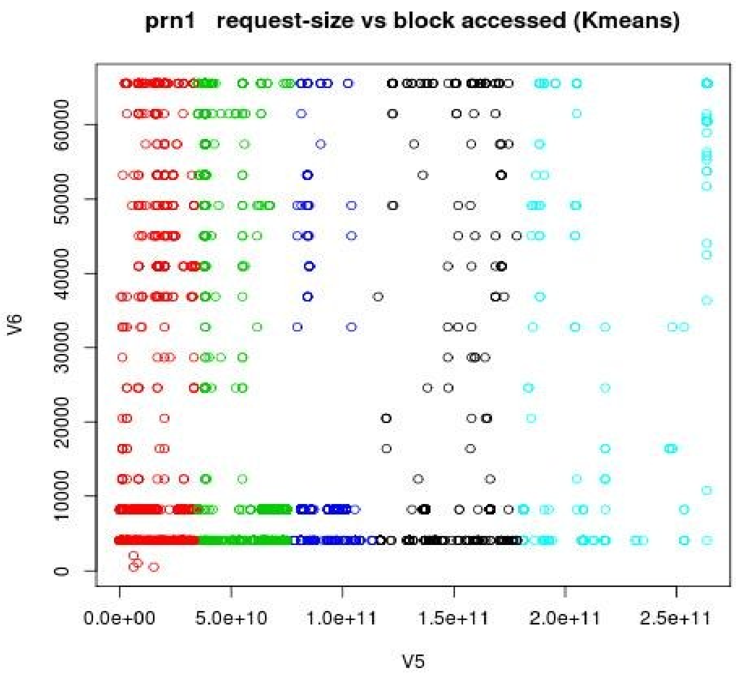
(c)


mds1 request-size vs block accessed (Kmeans)
V6
V5
0
10000
20000
30000
40000
0e+00
1e+11
2e+11
3e+11
4e+11
5e+11

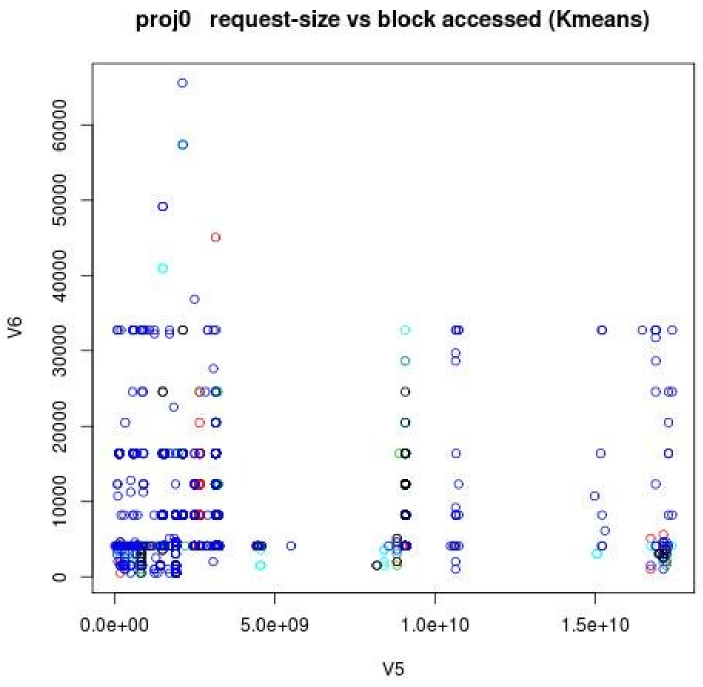
(d)


prn0 request-size vs block accessed (Kmeans)
V6
V5
0e+00
2e+05
4e+05
6e+05
8e+05
1e+06
0e+00
1e+10
2e+10
3e+10
4e+10
5e+10
6e+10

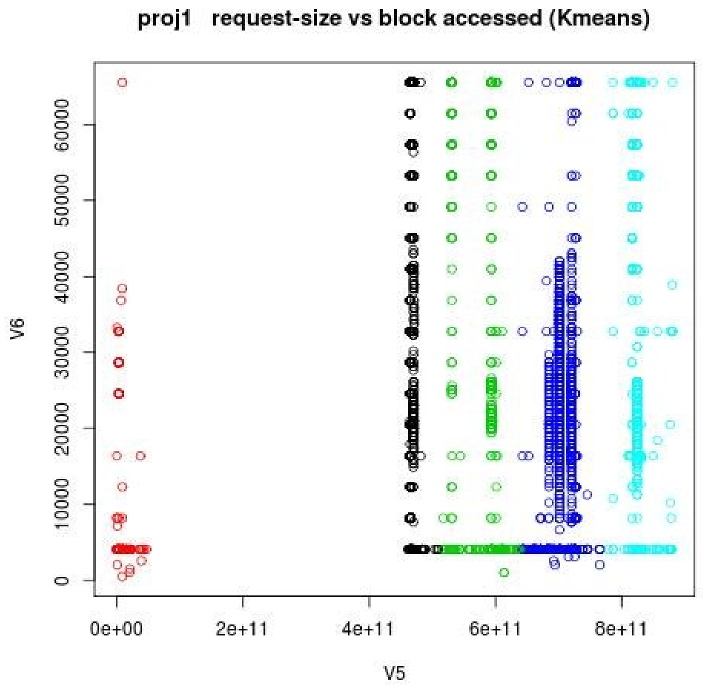
(e)


prn1 request-size vs block accessed (Kmeans)
V6
V5
0
10000
20000
30000
40000
50000
60000
0.0e+00
5.0e+10
1.0e+11
1.5e+11
2.0e+11
2.5e+11

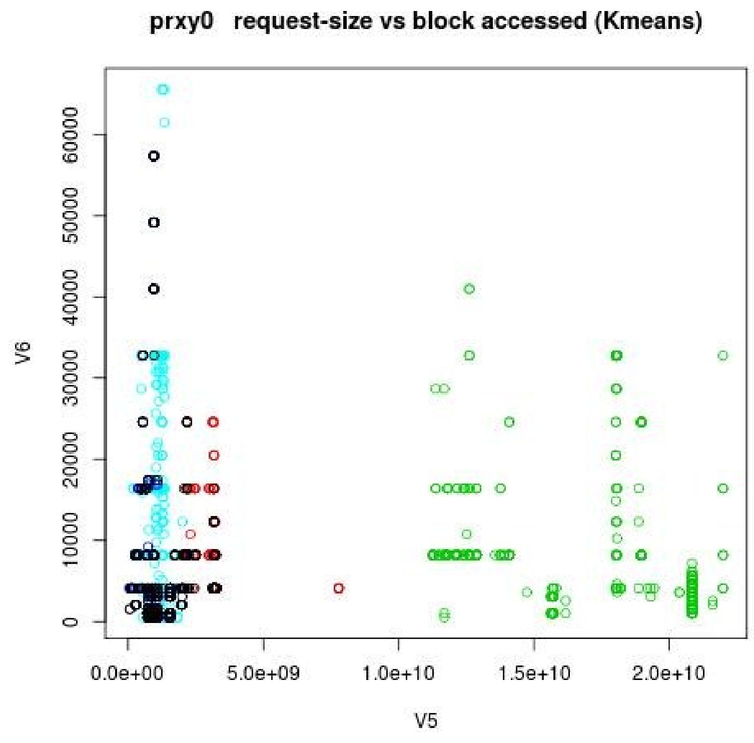
(f)


proj0 request-size vs block accessed (Kmeans)
V6
V5
0
10000
20000
30000
40000
50000
60000
0.0e+00
5.0e+09
1.0e+10
1.5e+10

(g)


proj1 request-size vs block accessed (Kmeans)
V6
V5
0
10000
20000
30000
40000
50000
60000
0e+00
2e+11
4e+11
6e+11
8e+11

(h)



(i)

Figure 15: K-means clustering done on MSR Cambridge traces using three workload dimensions namely block number, request size and inter-arrival time. For simplicity, five clusters were used, and the plot here shows two workload dimensions i.e. request size and block accessed. The plots are for MSR traces (a) hm0 (b) hm1 (c) mds0 (d) mds1 (e) prn0 (f) prn1 (g) proj0 (h) proj1 (i) prxy0. Note: Here V5 is 5th dimension of MSR trace, and V6 is 6th dimension of the trace.

In Figure 15, wherever the clusters are clearly demarcated as in case of prn0 traces, it means that there is not much dependency in the trace between request size, inter-arrival time, and block accessed. And in this specific case, request size and inter-arrival times seem to have uniform distribution.

4.1.2 ***Multi-clustering*****:** Another method is to use multiple clustering algorithms to classify the input data set into various clusters. Different clustering methods available are:

(i) *Hierarchical clustering*: Hierarchical clustering is a method of cluster analysis where hierarchies of clusters are built. It could be agglomerative or divisive. In agglomerative clustering the initial number of clusters is equal to the number of data points, and starts processing, the clusters are merged based on similarities. In Divisive clustering, all data points initially belong to one cluster, and as data is processed, they get divided into multiple smaller clusters. Drawback of agglomerative methods is that the memory usage could be proportional to the square of the number of groups in the initial partition [29].

(ii) *K-means clustering*: K-means clustering uses a data point to describe cluster representatives, and the decision of whether to include the IO/data point into the cluster is made by calculating the distance of data point from the centroid of each cluster, and including it in the cluster which it is closest to. This method is highly influenced by cluster centers.

(iii) *Model based clustering*: Clustering could be done based on some predefined models or distributions and type of clustering is good for large data sets. In this type of clustering, no initial clusters exist.

After performing different types of clustering on the same data set, a majority algorithm is used to reach to a final classification based on majority votes and to set class labels for each of the points. Figure 16 shows majority based clustering process.

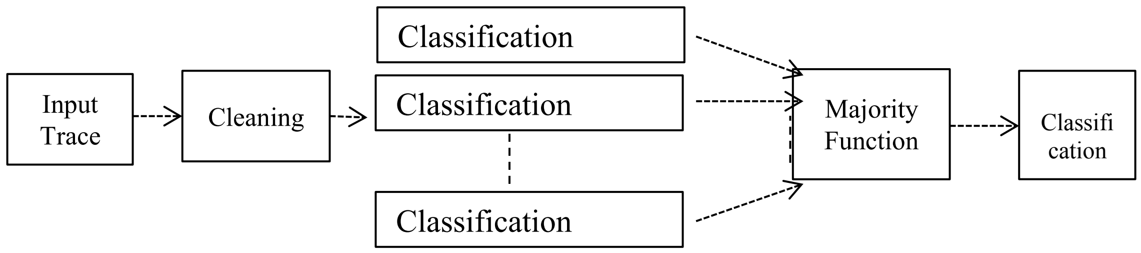


Figure 16: Majority votes based clustering methodology. Here, the input trace is provided to the system, and then cleansing of the trace file is done. After these stages, different classification methodologies are tried out, and using a majority function, the data points

are grouped into the clusters that receive the majority points or votes.

4.1.3 ***Model Based*****:** Yet, another method to perform phase detection is to try fitting various models on the input data points. Some of the data points belonging to a particular model might follow a particular distribution. There could be various models existing in an input trace. This methodology can be used to build models, and create a repository of earlier seen workload patterns. Hence, in case of a learning algorithm the prediction can be improved over time. Request size of one of the production traces was modeled, and consecutive sequences of IOs fitting a model were suggested to belong to the same phase. Whenever next set of IOs did not fit the previous distribution, a new phase would be initialized, and the process will be repeated.

Figure 17 shows one such model which was easily fit in R programming language, for a subset of the email workload trace. The idea here was to iterate through elements of the trace until the model fit shows significant increase in errors. In this case, we can call these IO sequences as a phase. We can see lot of such phases repeat throughout the trace.

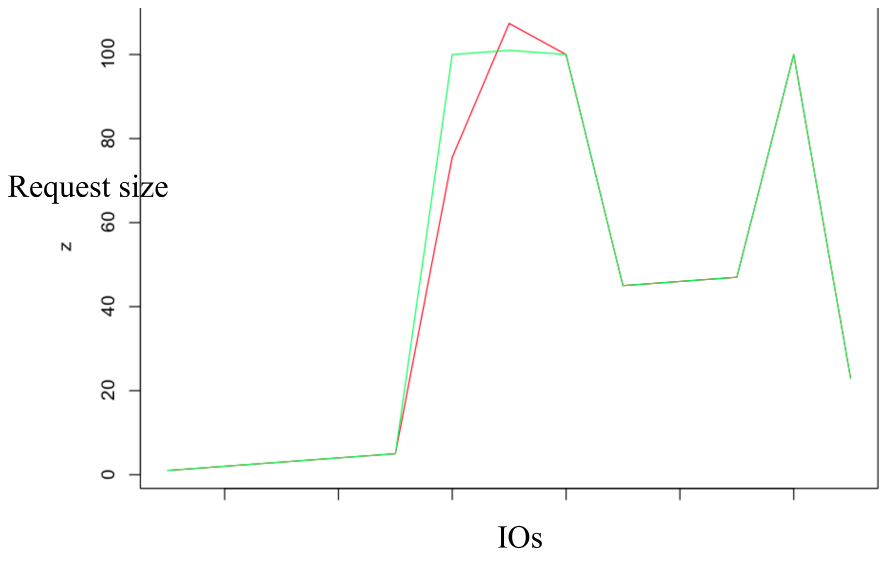


Figure 17: Model fitting for one of the Seagate Traces representing email workload

4.2 Summary

Phase detection was done using clustering, multi-clustering and model based clustering approaches. Multi-clustering is complex, but provides a good distribution for phases, as a majority vote based partitioning is done. One of the major observations here was that not all workload dimensions played an important role while clustering. For some traces like prn0, the block access pattern was the dominating factor, and other dimensions were more or less uniformly distributed.

Chapter 5
Model Based Trace Generation

Disk IO workloads for some applications like networking, email-server, production etc. show long-range dependence in their autocorrelation [1,2,3]. These traces show some level of self-similarity for certain dimensions and are sometimes fractal. Such traces pose design problems to a cache designer, as the access pattern has long-range dependency. If the cache is big enough then a LRU caching policy might suffice, otherwise in order to optimize cache and disk performance, fractal prefetching could be used [24].

5.1 Hurst Phenomenon and Similarity

Disk workloads show long-range dependence in autocorrelation, and hurst exponent helps to understand the changing autocorrelation in time series. Hurst exponent (H) helps to understand long-range dependence. The workload is said to be similar if H value is between 0.5 and 1, and for values of $H > 1$ and $H < 0.5$, there is no similarity. We can observe that the autocorrelation decays gradually in case of similar workloads. If one divides up the trace with the boundaries separated by similar IO traffic, then one can call those phases to be similar. Here, one of the production traces and MSR Cambridge traces are used to find the trend of autocorrelation function. The results are shown in figure 18 and figure 19. We can see that request size, read or write, and the blocks accessed show autocorrelation function that does not decay rapidly. And this proves that there is some long-range dependence in the workloads. On the contrary inter-arrival time shows less autocorrelation, and this indicates that the workload is bursty.

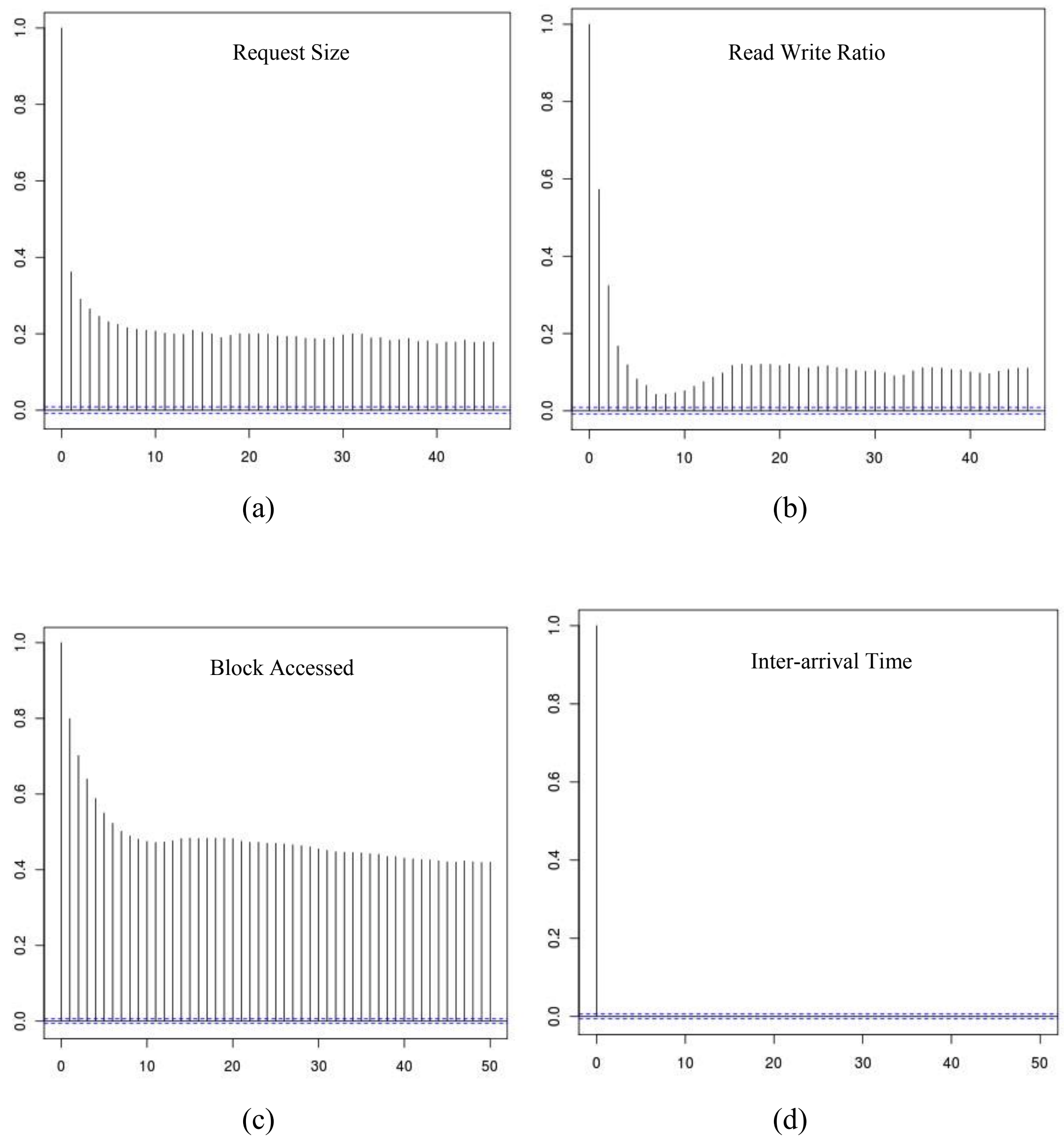
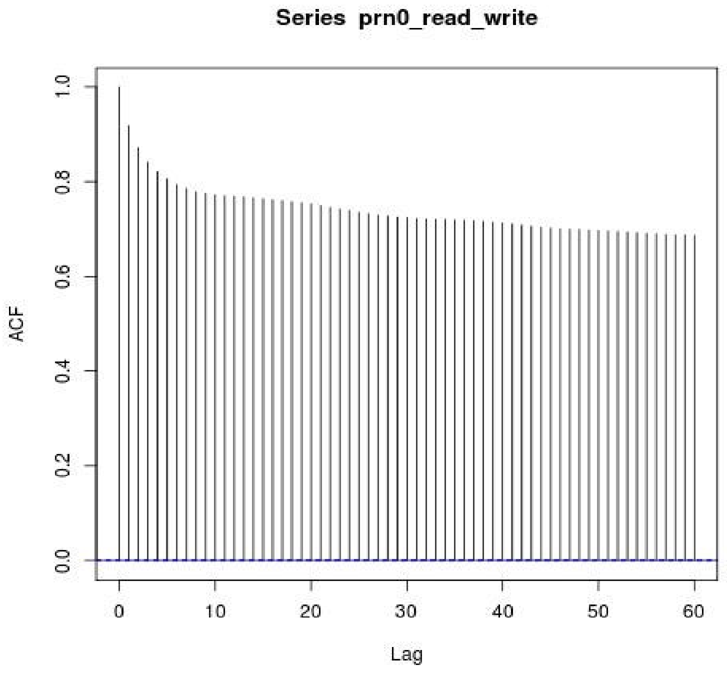


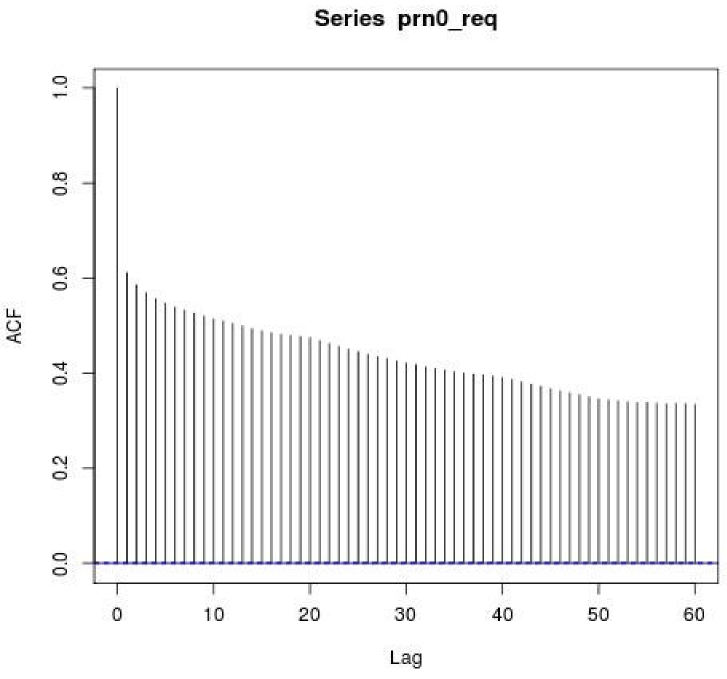
Figure 18: Autocorrelation functions in one of the Seagate Customer email trace (a) show autocorrelation in request size (b) autocorrelation in read write ratio (c) autocorrelation in block-accessed (d) autocorrelation in inter-arrival time. Here the y-axis represents the auto-correlation, and x-axis represents the trace index in log scale.

Autocorrelation functions for various MSR Traces are shown in figure 19 below



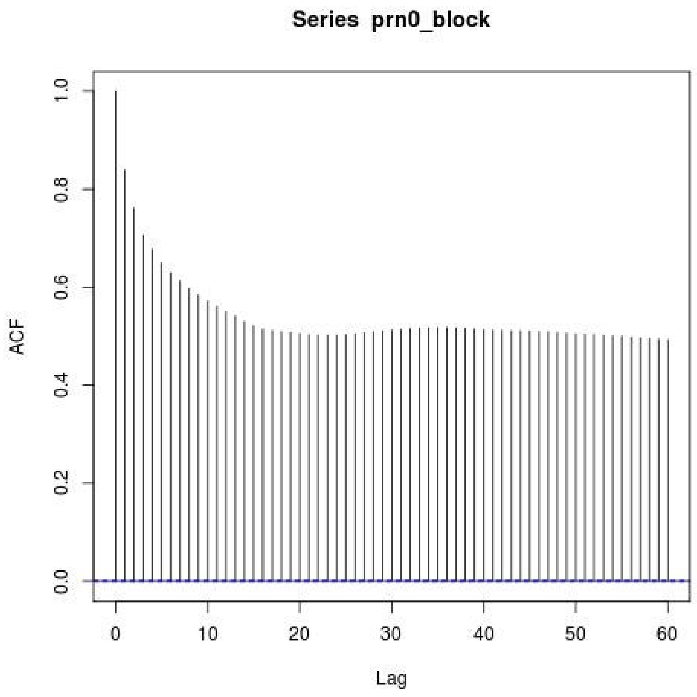
Series prn0_read_write
ACF
Lag
0.0
0.2
0.4
0.6
0.8
1.0
0
10
20
30
40
50
60

(i)

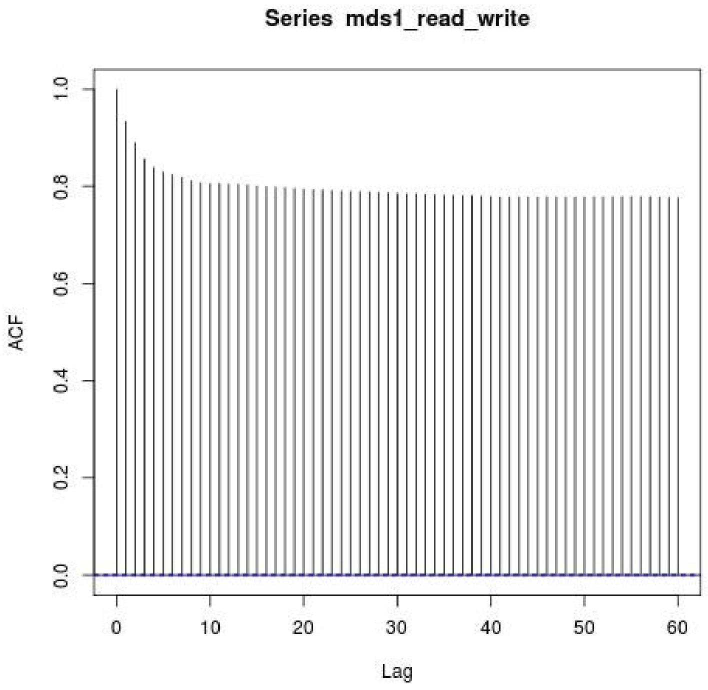


Series prn0_req
ACF
Lag
0.0
0.2
0.4
0.6
0.8
1.0
0
10
20
30
40
50
60

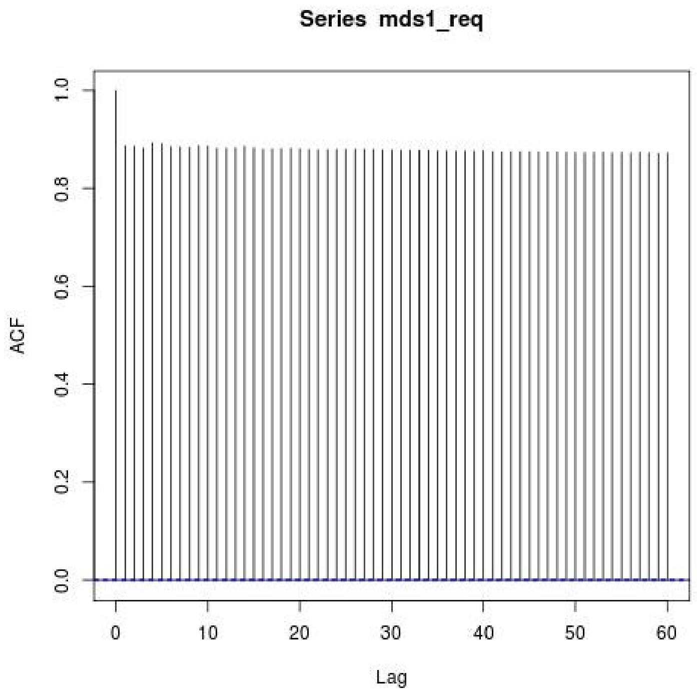
(ii)


Series prn0_block
ACF
1.0
0.8
0.6
0.4
0.2
0.0
0
10
20
30
40
50
60
Lag

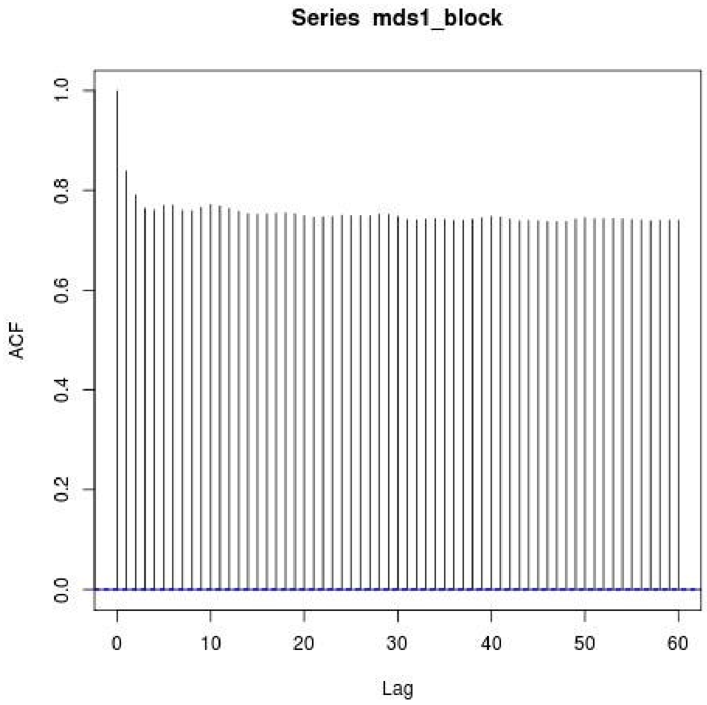
(iii)


Series mds1_read_write
ACF
1.0
0.8
0.6
0.4
0.2
0.0
0
10
20
30
40
50
60
Lag

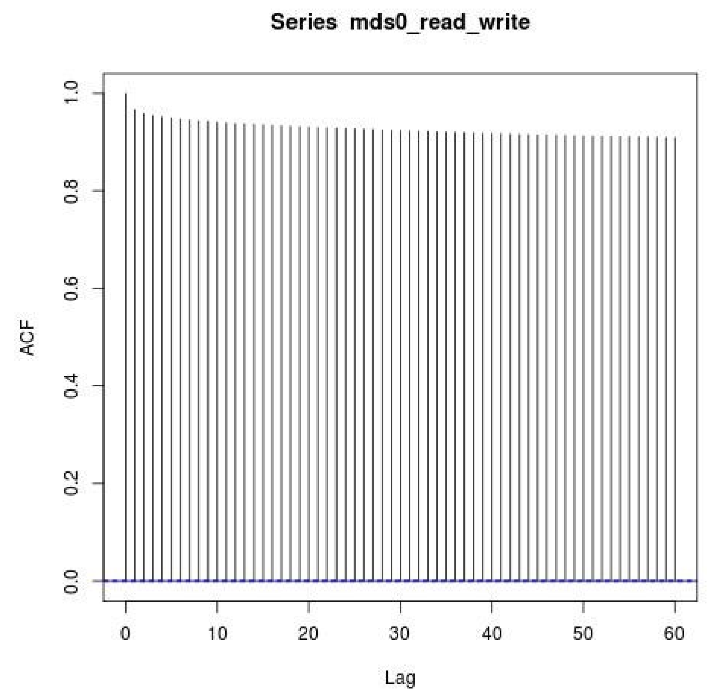
(iv)


Series mds1_req
ACF
Lag
0.0
0.2
0.4
0.6
0.8
1.0
0
10
20
30
40
50
60

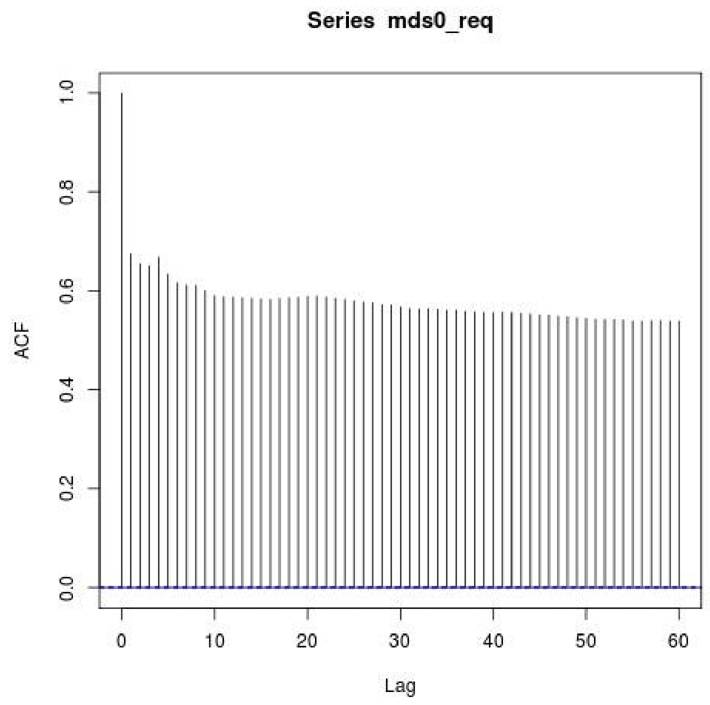
(v)


Series mds1_block
ACF
Lag
0.0
0.2
0.4
0.6
0.8
1.0
0
10
20
30
40
50
60

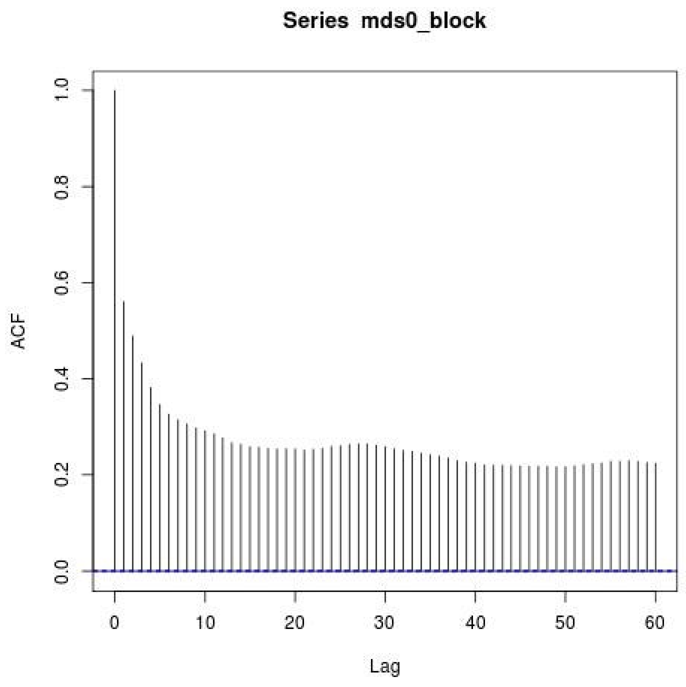
(vi)


Series mds0_read_write
ACF
Lag
0.0
0.2
0.4
0.6
0.8
1.0
0
10
20
30
40
50
60

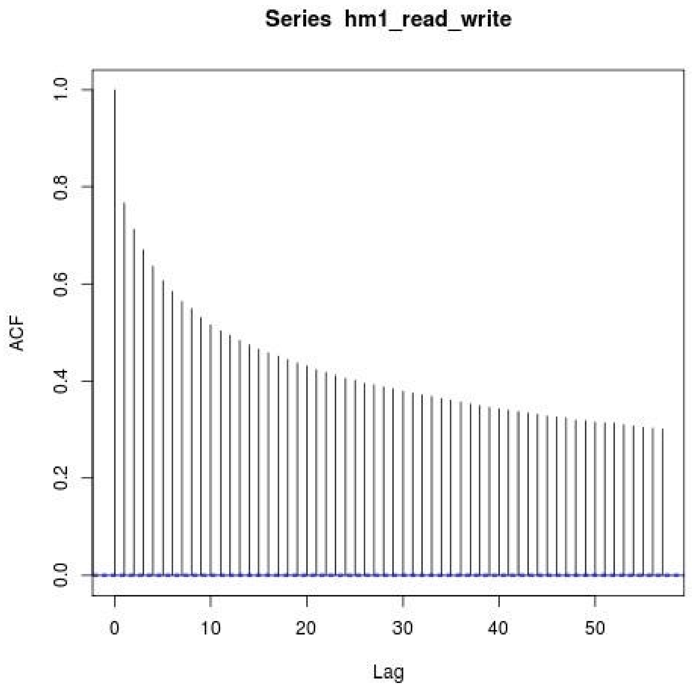
(vii)


Series mds0_req
ACF
Lag
0.0
0.2
0.4
0.6
0.8
1.0
0
10
20
30
40
50
60

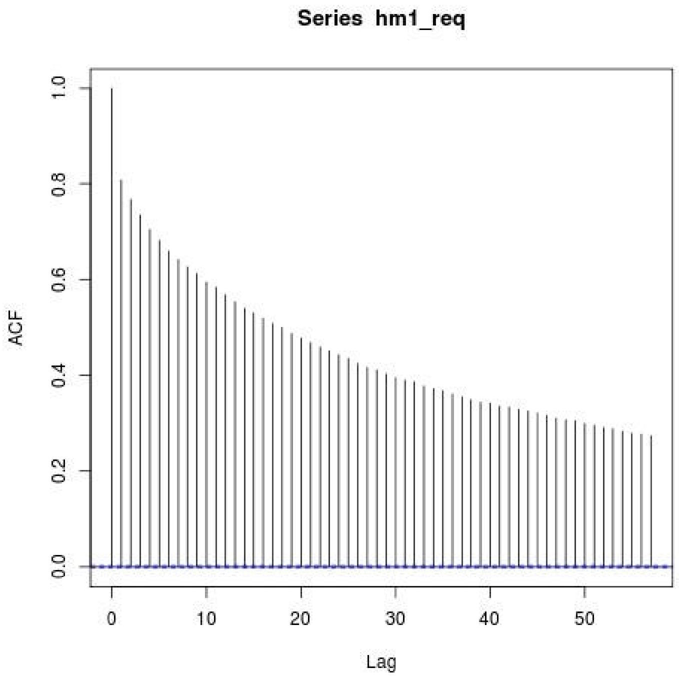
(viii)


Series mds0_block
ACF
Lag
0.0
0.2
0.4
0.6
0.8
1.0
0
10
20
30
40
50
60

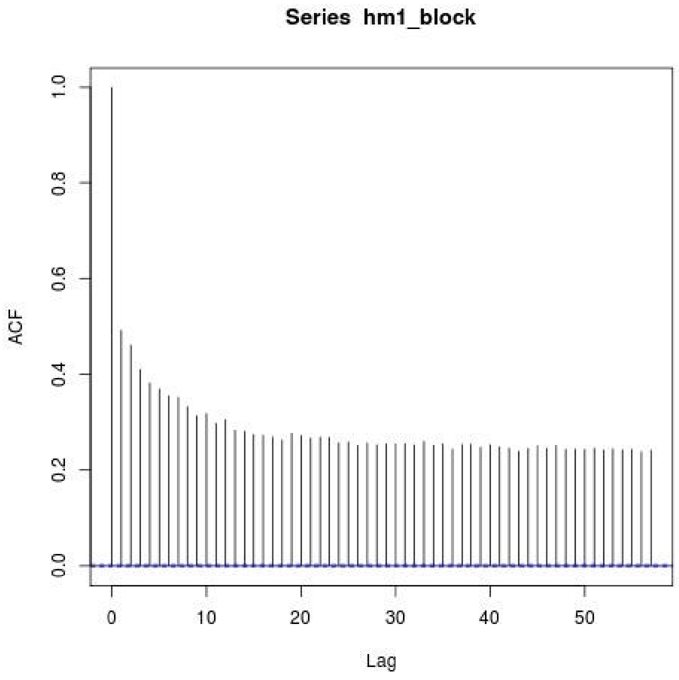
(ix)


Series hm1_read_write
ACF
Lag
0.0
0.2
0.4
0.6
0.8
1.0
0
10
20
30
40
50

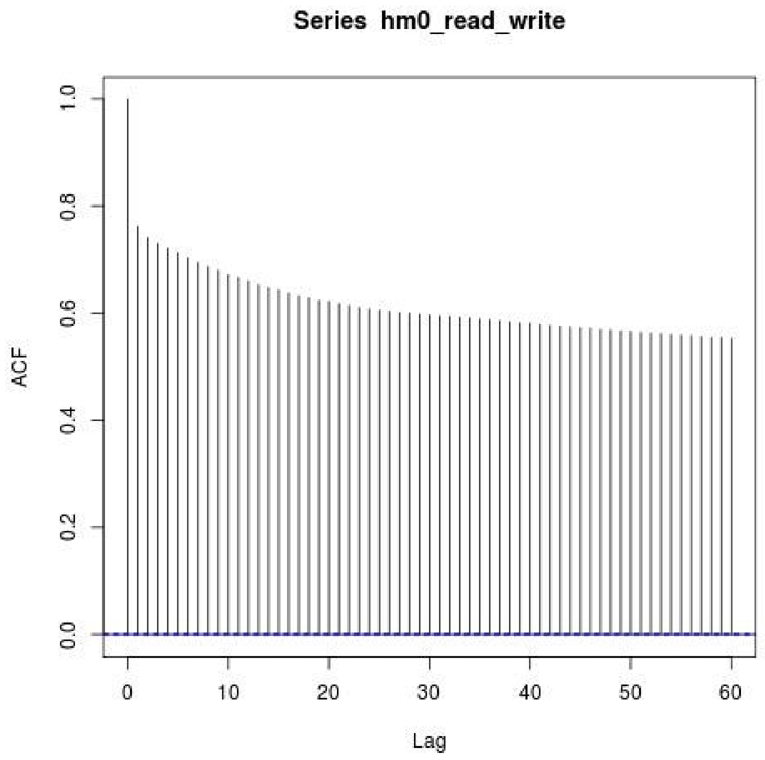
(x)


Series hm1_req
ACF
Lag
0.0
0.2
0.4
0.6
0.8
1.0
0
10
20
30
40
50

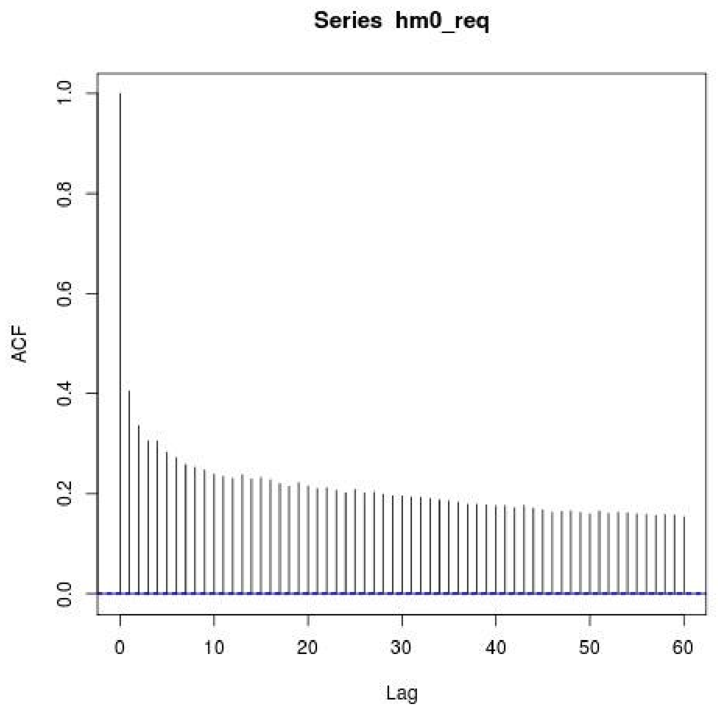
(xi)


Series hm1_block
ACF
Lag
0.0
0.2
0.4
0.6
0.8
1.0
0
10
20
30
40
50

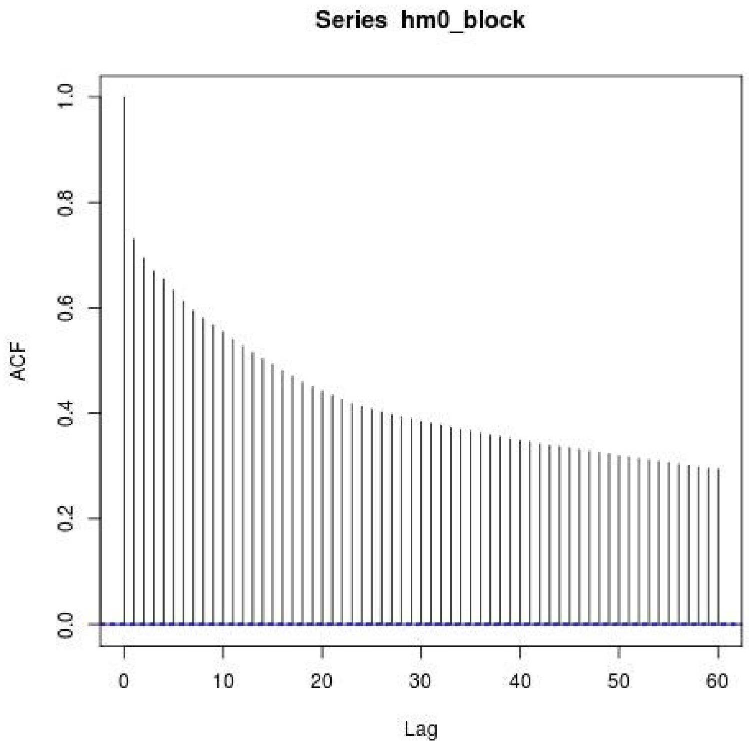
(xii)


Series hm0_read_write
ACF
Lag
0
10
20
30
40
50
60
0.0
0.2
0.4
0.6
0.8
1.0

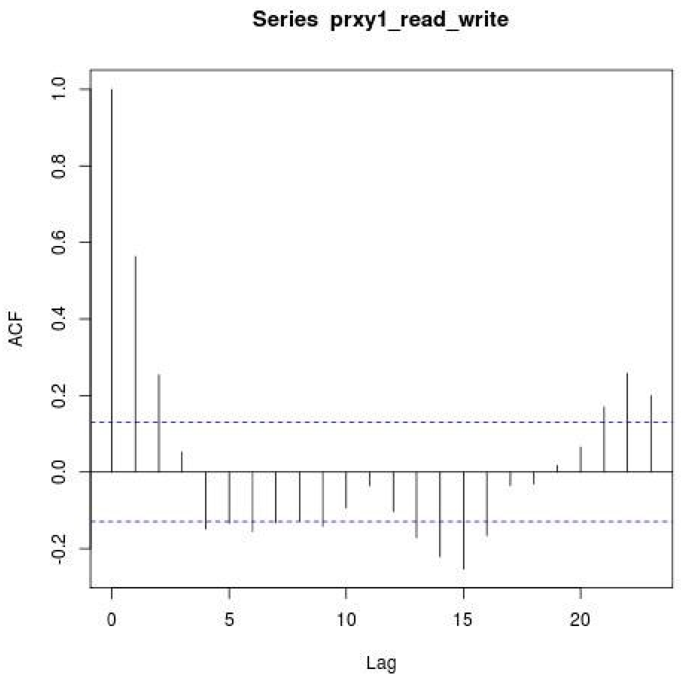
(xiii)


Series hm0_req
ACF
Lag
0
10
20
30
40
50
60
0.0
0.2
0.4
0.6
0.8
1.0

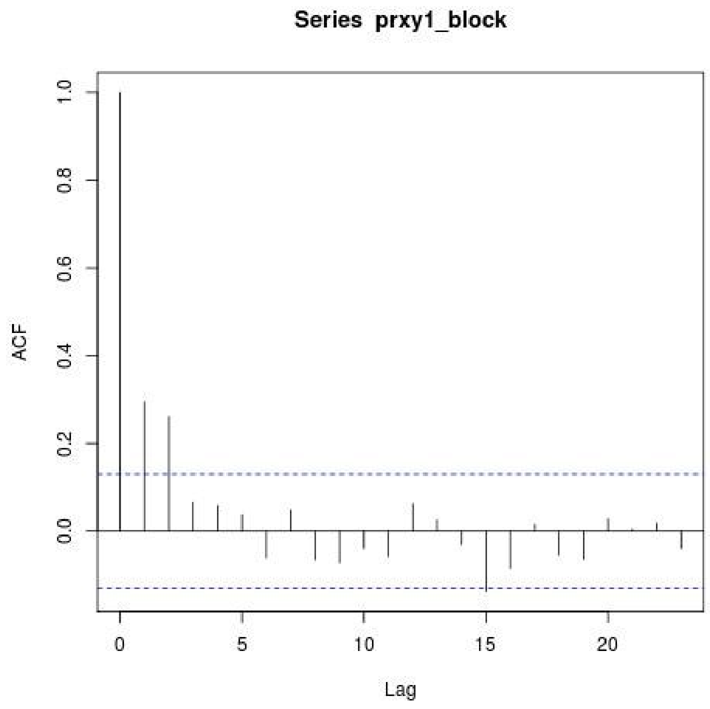
(xiv)


Series hm0_block
ACF
Lag
0.0
0.2
0.4
0.6
0.8
1.0
0
10
20
30
40
50
60

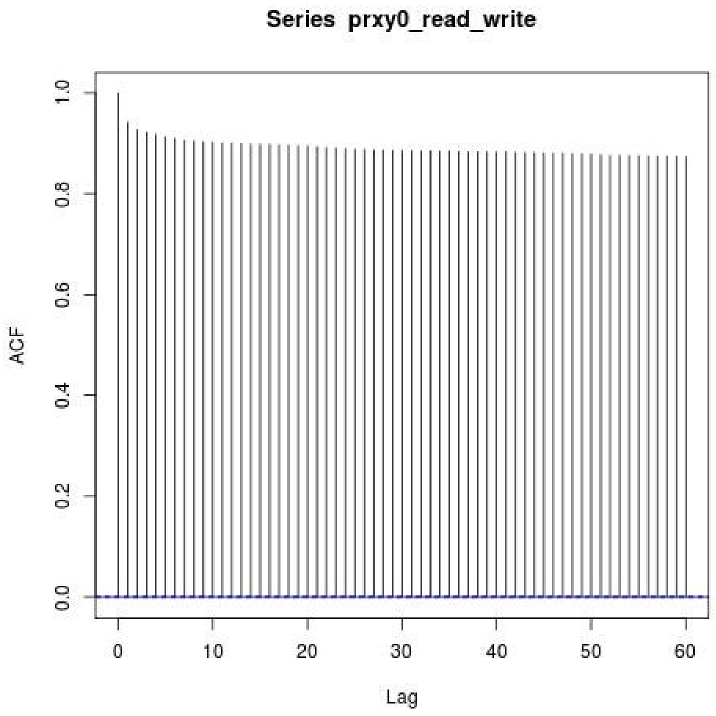
(xv)


Series prxy1_read_write
ACF
Lag
-0.2
0.0
0.2
0.4
0.6
0.8
1.0
0
5
10
15
20

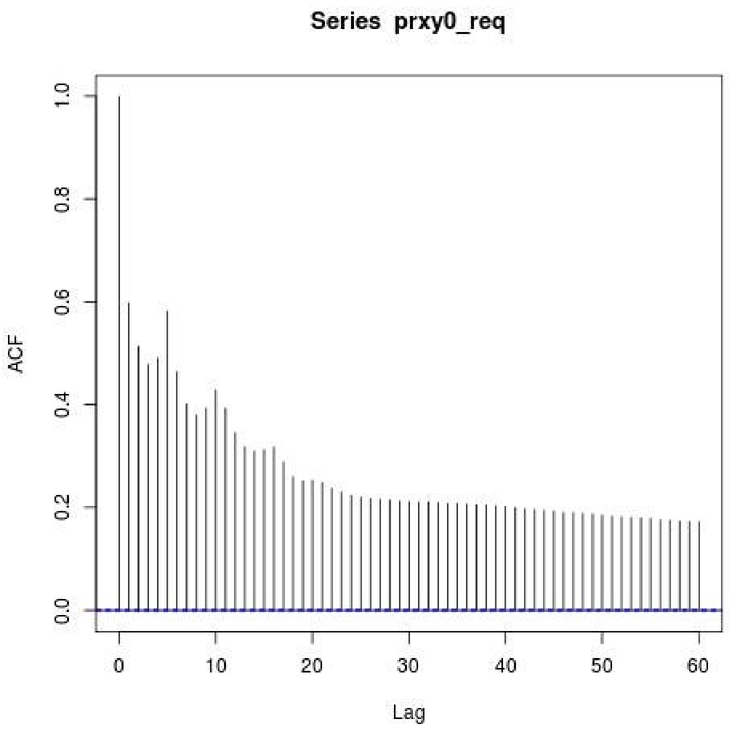
(xvi)


Series prxy1_block
ACF
Lag
1.0
0.8
0.6
0.4
0.2
0.0
0
5
10
15
20

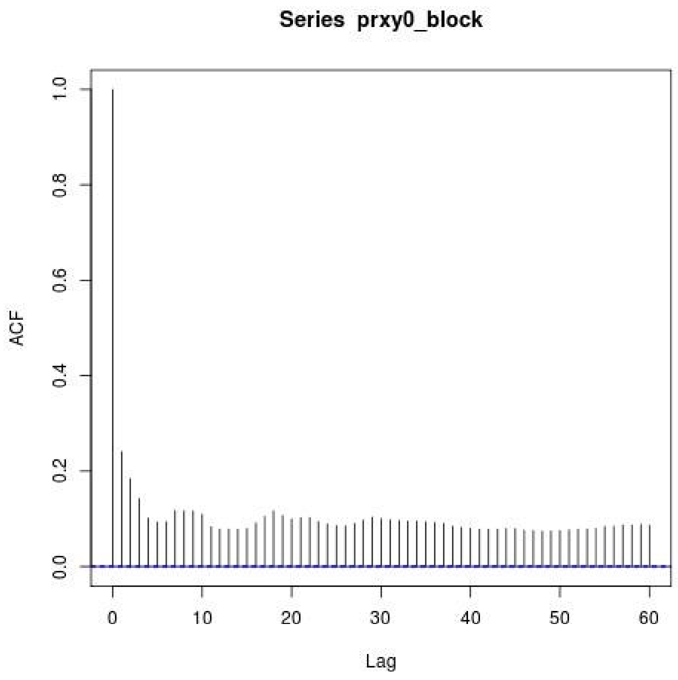
(xvii)


Series prxy0_read_write
ACF
Lag
1.0
0.8
0.6
0.4
0.2
0.0
0
10
20
30
40
50
60

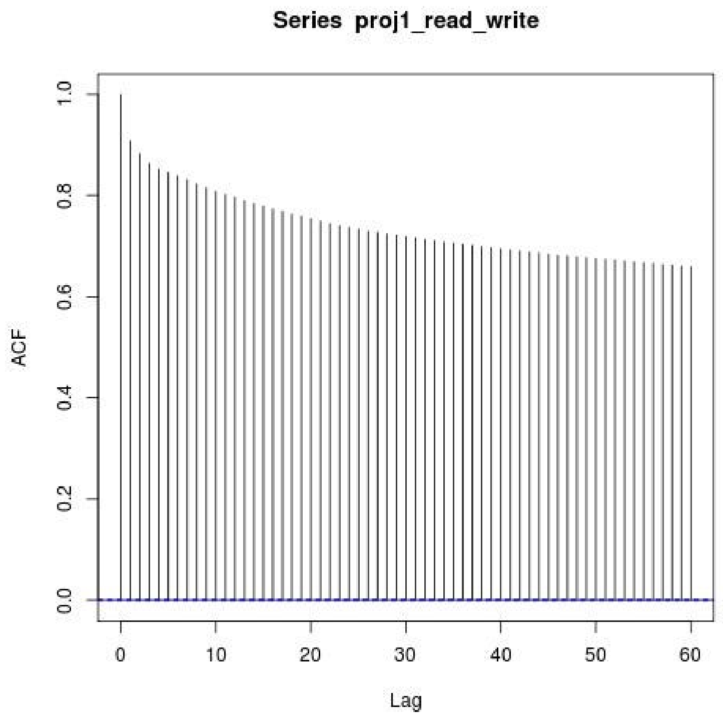
(xviii)


Series prxy0_req
ACF
Lag
0.0
0.2
0.4
0.6
0.8
1.0
0
10
20
30
40
50
60

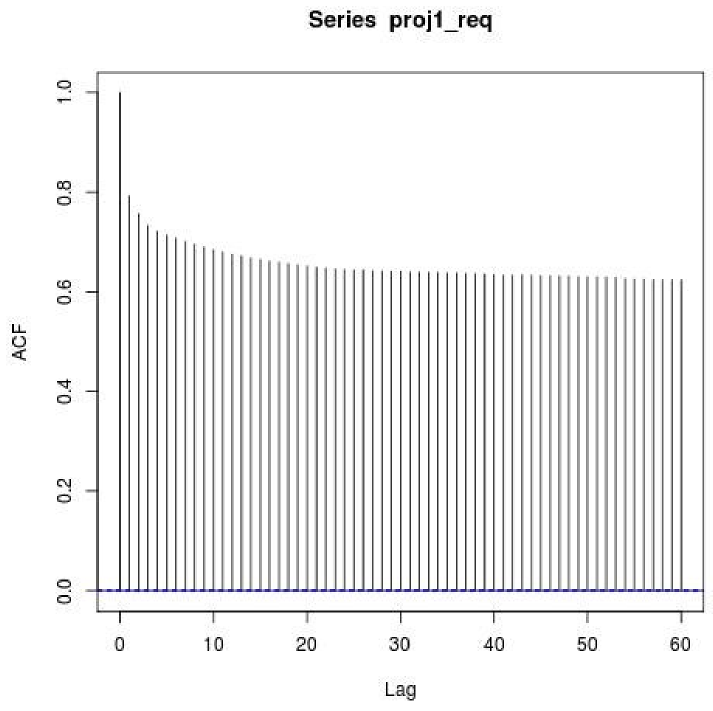
(xix)


Series prxy0_block
ACF
Lag
0.0
0.2
0.4
0.6
0.8
1.0
0
10
20
30
40
50
60

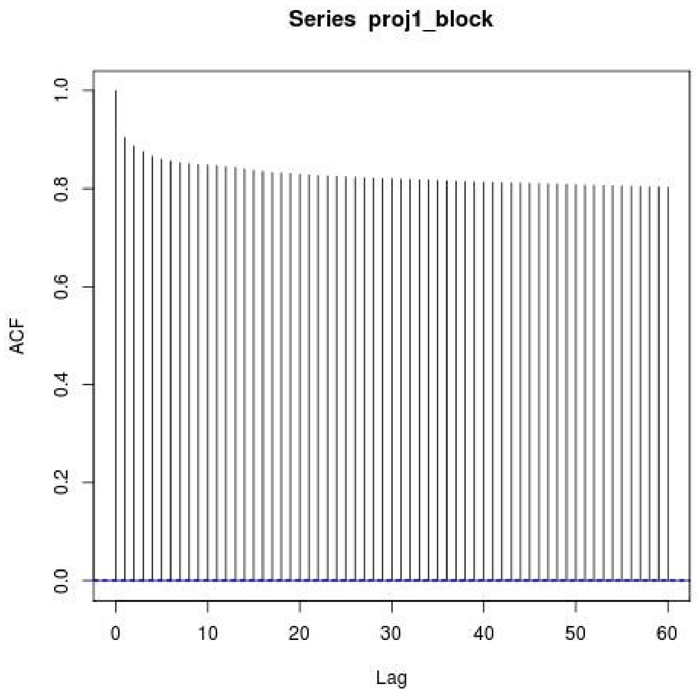
(xx)


Series proj1_read_write
ACF
Lag
0
10
20
30
40
50
60
0.0
0.2
0.4
0.6
0.8
1.0

(xxi)

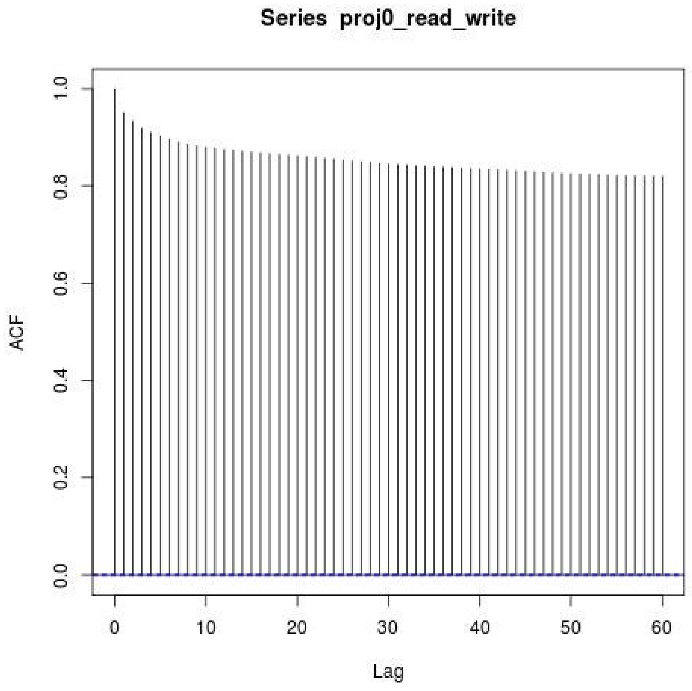

Series proj1_req
ACF
Lag
0
10
20
30
40
50
60
0.0
0.2
0.4
0.6
0.8
1.0

(xxii)



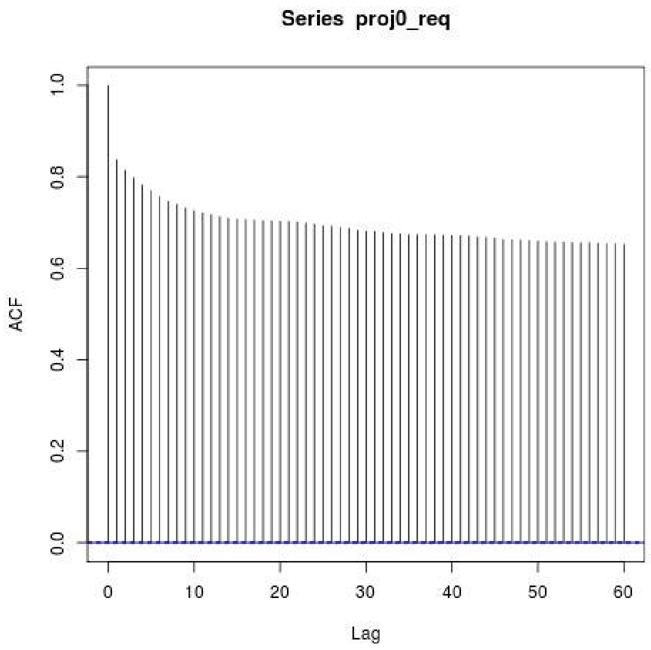
Series proj1_block
ACF
1.0
0.8
0.6
0.4
0.2
0.0
0
10
20
30
40
50
60
Lag

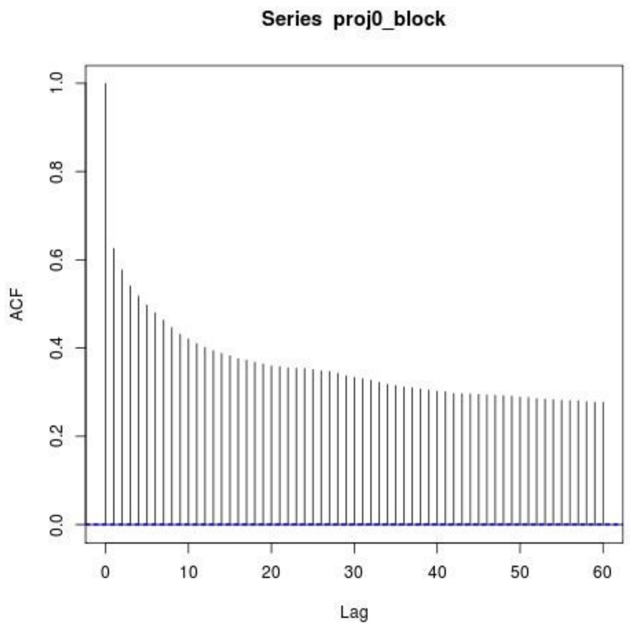
(xxiii)

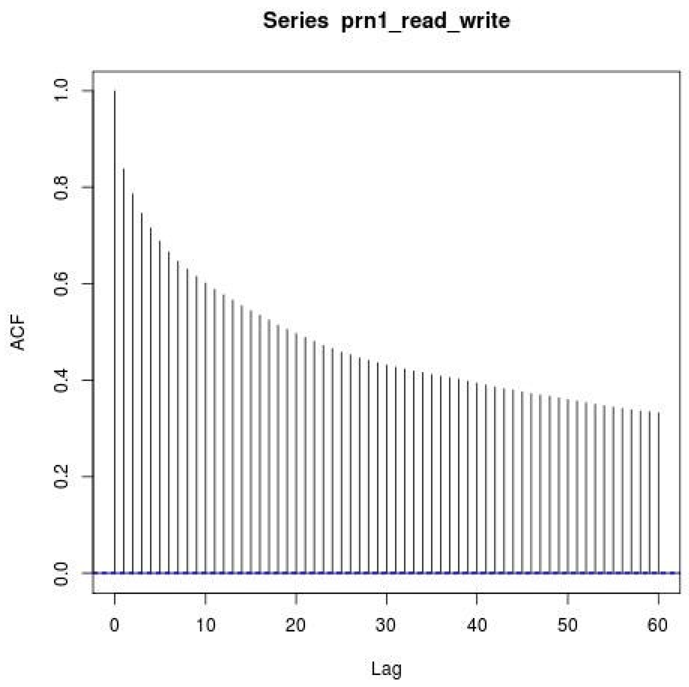


Series proj0_read_write
ACF
1.0
0.8
0.6
0.4
0.2
0.0
0
10
20
30
40
50
60
Lag

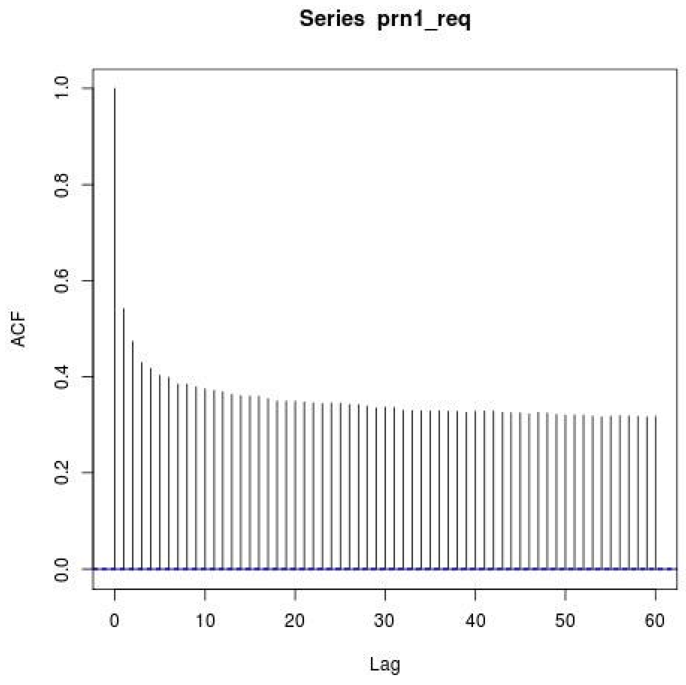
(xxiv)


Series proj0_req
ACF
Lag
0
10
20
30
40
50
60
0.0
0.2
0.4
0.6
0.8
1.0
(xxv)

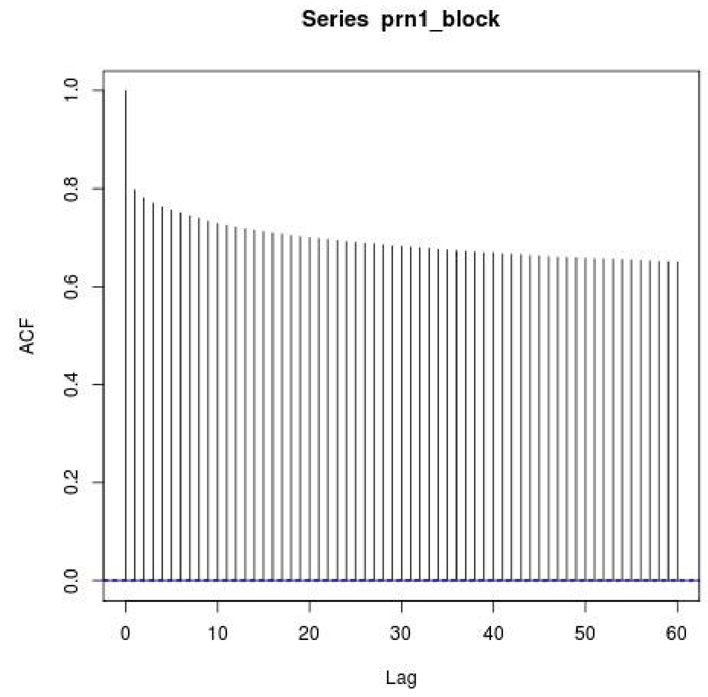

Series proj0_block
ACF
Lag
0
10
20
30
40
50
60
0.0
0.2
0.4
0.6
0.8
1.0
(xxvi)



(xxvii)



(xxviii)



(xxix)

Figure 19: Autocorrelation functions of MSR traces (i) prn0 rw (ii) prn0 req (iii) prn0 blk (iv) mds1 rw (v) mds1 req (vi) mds1 blk (vii) mds0 rw (viii) mds0 req (ix) mds0 blk (x) hm1 rw (xi) hm1 req (xii) hm1 blk (xiii) hm0 rw (xiv) hm0 req (xv) hm0 blk (xvi) prxy1 rw (xvii) prxy1 blk (xviii) prxy0 rw (xix) prxy0 req (xx) prxy0 blk (xxi) proj1 rw (xxii) proj1 req (xxiii) proj1 blk (xxiv) proj0 rw (xxv) proj0 req (xxvi) proj0 blk (xxvii) prn1 rw (xxviii) prn1 req (xxix) prn1 blk. Here the y-axis represents the auto-correlation, and x-axis represents the trace index in log scale.

Autocorrelations in MSR traces also showed gradually decreasing values, and some traces like proj1 showed very good long range dependency. The level of long-range dependence is captured by the hurst parameter. Fractional differencing can be used to understand long-range dependence [30]. This process is derived from ARIMA (0,n,0) process.

We know that the random walk can be denoted by ARIMA (0,1,0) given by:

$$X_t = X_{t-1} + \varepsilon_t$$

And, the relationship between hurst parameter and 'n' here is H = n + 0.5.

5.2 Burstiness in Workloads

One can use Hurst parameter to understand the burstiness of the entire trace, but if one wants to study the patterns within a certain time interval, then Multifractals can be used [25]. This is especially useful when one wants to capture the dependencies while issuing IOs. Burstiness of one of the Seagate customer traces is shown in figure 20 below:

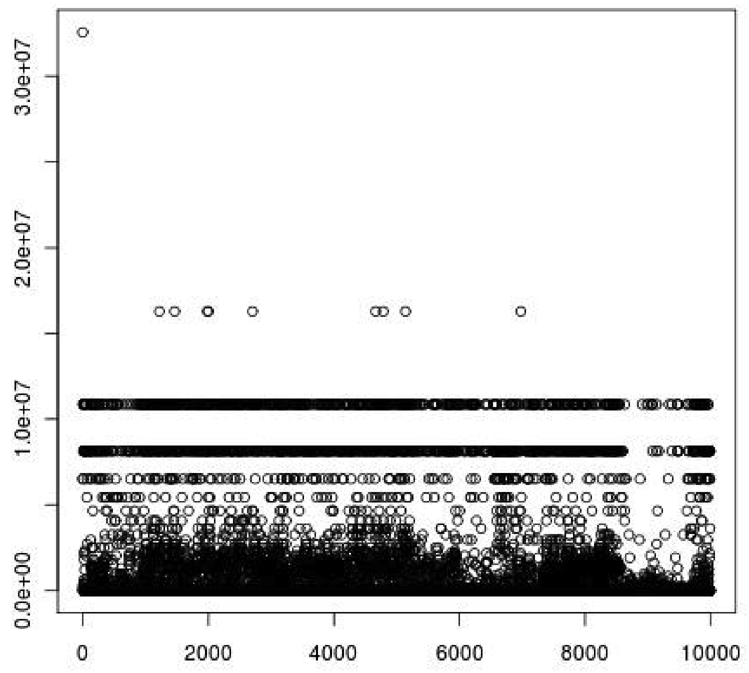


Figure 20: Burstiness of a Seagate customer trace

Here a scaled burstiness metric used is

$$\frac{1}{(IAT + 1)} * Maximum\ IAT$$

Where, IAT is the inter-arrival time.

The result obtained shows that the workload experiences scattered high burst periods, and there is some dependence and symmetry in the Inter-arrival pattern. Replaying this burstiness behavior is important in the workload replay engine to realistically simulate the workloads.

5.3 ARIMA Models

As discovered before, workloads show bursty behavior, and dimensions such as block accessed, read or write request pattern and request size show a slowly degrading auto-correlation function, and show long-range dependence. After identifying this behavior, forecasting or predicting future workload patterns using a model is an intuitive thing to pursue. Using of such models will help to perform resource allocations, failure prediction, bottleneck prediction, adjustment of QoS requirements etc. There are different approaches to model fitting, and some performed better than the other based on input trace, and application behavior.

5.3.1 Sliding Window Approach

Phase-detection using clustering showed that many segments of the trace are self-similar or follow a particular distribution or model. Similarity in trace is a good thing for modeling, as many sub-components of the trace might fit into the same statistical model. In case, the user wants to maintain a historical database to capture all previously seen distributions, then there is some saving in terms of space complexity for such workloads. The sliding window approach works by observing past 'n' IOs and studying the influence of this segment on the future [16, 17]. In order to get accurate modeling, it is a good approach to use a validation set and a training set. The training set is used to train the model, and it generally captures affect of past observations on the present. The model parameters are captured in the parameter coefficients. Validation set on the other hand is used to test the accuracy of the model and make improvements if necessary. An ARIMA(p,q) model is denoted by:

$$Y(t) = \varphi1 \cdot Y(t-1) + ...\varphi p \cdot Y(t-p) + \varepsilon(t) + \theta1 \cdot \varepsilon(t-1) + ... + \theta q \cdot \varepsilon(t-q)$$

where $\varepsilon(t)$ are normally distributed random variables with variance $\sigma2$. Here, as shown in figure 21 the input data is divided into two sets: Test and Validation Set. As a common practice, 70% of the inputs were used for test set, and the rest 30% for validation set. Multiple dimensions of the workloads show different behavior, and hence for test purposes each dimension was modeled. Here, in order to predict the possible IOs in prediction window, one uses the model built from the test set using a sliding window approach. This approach ensures that the model prediction is good for the test set.

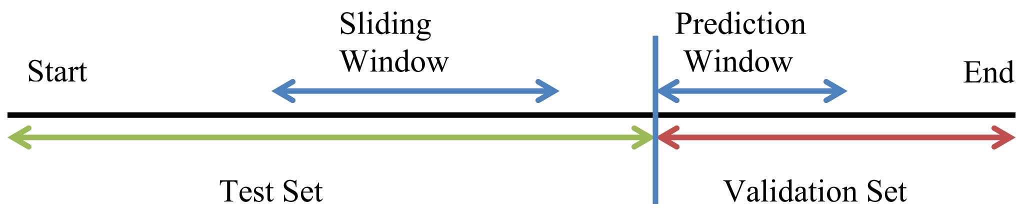


Figure 21: Sliding window approach for fitting ARIMA models

The “forecast” package in R was used to fit the ARIMA model, which can analyze uni-variate time series, and fit appropriate model. In this example, one of the customer traces captured from home was used, and the first phase as identified by phase detection had a total of 785 elements, and out of this 70% i.e. 550 elements were used as test set, and the rest 30% i.e.. 235 elements were used as validation set. From the results obtained we can see that ARIMA(1,0,1) process fits this data best, and other model parameters were

```
Coefficients:
        ar1      ma1     intercept
     0.7050  -0.4449     81.2166
s.e. 0.0759   0.0946      6.0287

sigma^2 estimated as 5680:  log likelihood=-3157.78
AIC=6323.56   AICc=6323.64   BIC=6340.8
```

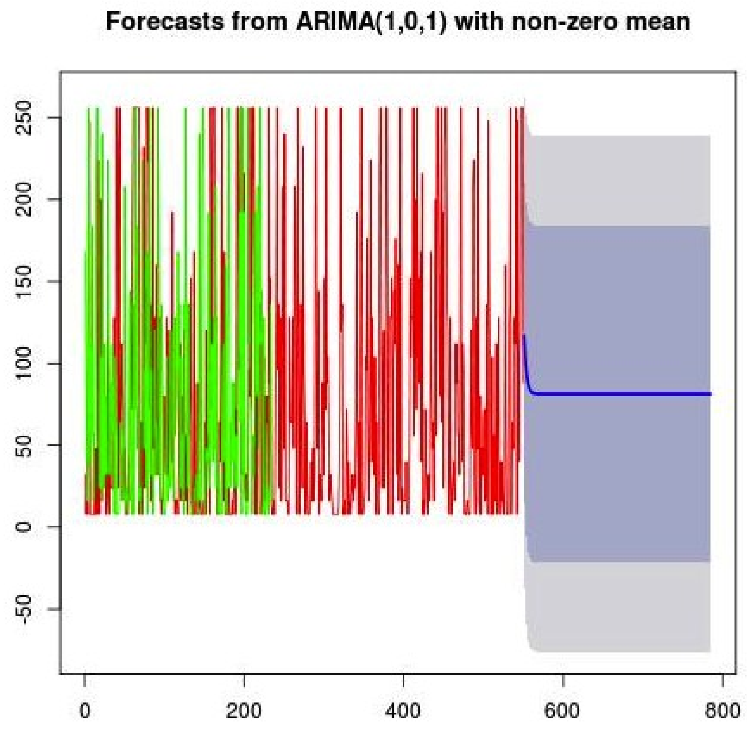


Figure 22: ARIMA model forecast for request size in a production trace in a particular phase

Figure 22 shows that the prediction is somewhat accurate, and the Akaike information criterion (AIC) here helps to measure the goodness of the statistical model. When comparing multiple models, the ones with minimum AIC value are the best. The AIC for a model with k AR terms fitted to a series of length n is defined as

$$AIC = \ln \hat{\sigma}2 + 2 \cdot k/n$$

Another model was fit to find the model for blocks accessed (see figure 23). Here, the best fit was obtained using the ARIMA(2,0,2) model. The model details are shown below:

```
Coefficients:
       ar1     ar2          ma1      ma2      intercept
    0.6941   0.0068     0.1816   0.1351   62197899
sigma^2 estimated as 5.293e+14:  log likelihood=-10104.18
AIC=20220.36   AICc=20220.52   BIC=20246.22
```

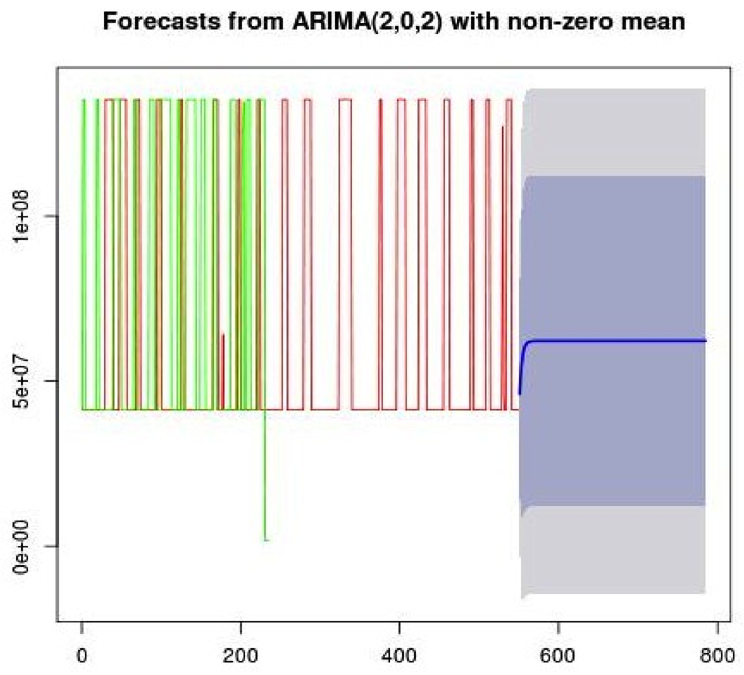


Figure 23: ARIMA model forecast for block accessed in a production trace in a particular phase. The model prediction followed similar statistical distribution, but there were some errors in prediction. The AIC value for this model was high, but the model achieved similar block values, which is important characteristic of the application.

5.3.2 Holistic Approach

Another approach for modeling is fitting a model for the entire data set, and this approach had its limitations that the predictions will not be accurate. ARIMA models try to forecast the future using the parameters extracted, the AIC value can tend to increase with increasing complexity of the data sets. It would be best to model each phase as they have some unique characteristics.

5.3.3 Limitations of using ARIMA models:

1. In a trace with around 100000 elements, there could be hundreds of possible ARIMA models. Different dimensions again will have a lot of models.
2. Correlations between different dimensions have to be captured using additional statistical tools
3. Traces with higher dependencies will have a better model, whereas traces with

lesser autocorrelation might have a weaker model.

4. Traces with higher phases will have more models than traces with lower number of phases. Traces with self-similarity will have lesser total models, hence it is not possible to give a number for amount of data to store for regenerating similar traces
5. Modeling is a compute intensive process, for longer traces, this process requires initial compute time and resources.
6. ARIMA models could get poor at performing long-term forecast and might produce straighter lines than predicting series with jumps or turns.

However, Model based trace generation is easily portable, and it is easy to understand. The efficiency of such approach also depends on the code to choose the best ARIMA model fit.

5.4 Entropy in Workloads

Entropy in traditional sense is the measure of uncertainty in a random variable, and it is typically measured in bits, nats or bans. One form of entropy is Shannon entropy (H), and it just calculates the average randomness or unpredictability of a random variable. Entropy is also sometimes called as information content.

$$H = -\sum_{i=1}^{m} \frac{n_i}{n} \log_2\left(\frac{n_i}{n}\right)$$

and n_i here is the frequency of a_i (where a = {a_1, a_2…. a_m} are the list of symbols). For example, if a = {1,2,3,4,5,4,5,6,7,8,7,8}
then P(1) = P(2) = P(3) = P(3) = P(6) = 1/12 and P(4) = P(5) = P(7) = P(8) = 2/12.
H = -((4/12) $\log_2$(1/12) + (8/12) $\log_2$ (2/12)) = 2.92.
For one example, entropy of various dimensions of the workload is plotted in figure 24. In this plot, the input trace was taken and divided iteratively, and the corresponding averages of Entropy values were plotted.

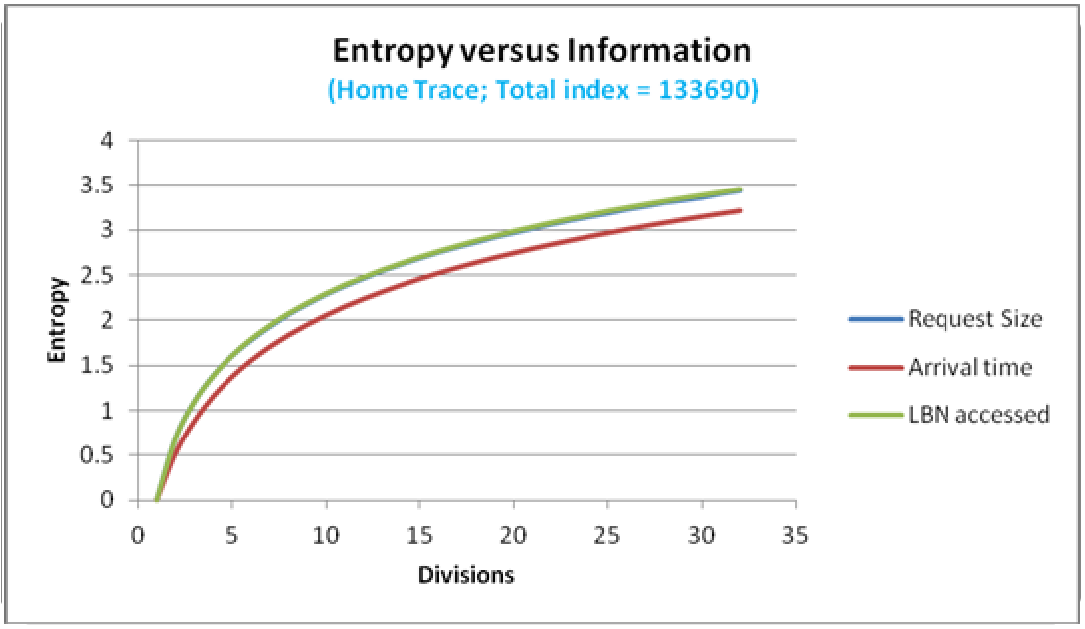


Figure 24: Entropy Plot

5.5 Summary

Disk IO workloads are found to show long-range dependency in auto-correlation, and hurst parameter could be used for understanding this dependence. Workloads are said to be similar if H value is between 0.5 and 1. IO trace parameters like request size, block accessed and read/write request showed gradually decaying autocorrelation function, whereas the inter-arrival time showed very less auto-correlation, indicating that the workload is possibly bursty. ARIMA models showed good predictions for request sizes and block sizes. There might be over 100 ARIMA models required for a trace with around 100000 elements. Phases detected provided to be a good cut off points in the trace for diving the trace, and performing modeling. Entropy study done provides us with information regarding long range dependency, and the variation of information with additional information, or IOs in this scenario.

Chapter 6
ELOS (Entropy Locality Operation Size) – Synthetic Workload Generator

6.1.Overview

As described in previous chapters, the existing workload generators lack the ability to generate workload traces with desired locality. Synthetic trace generation involves identification of important characteristics that dictate the behavior of the application. The common characteristics that are extracted across all different workloads are read/write ratio, random/sequential IO and request size. Although these parameters are necessary, they are not sufficient to describe a workload. Some of the common drawbacks in existing workload generators are listed below:

1. Application locality (temporal and spatial) is not preserved in synthetic workloads.
2. Many workload generators have predefined workload profiles to choose from, and have limited tunable parameters
3. Phase detection capabilities exist in almost none of the synthetic benchmarks and workload generators
4. Replay engine with controllable inter-arrival time, ordering and controller or buffer queue size is lacking
5. In spite of the importance of caching in system performance, a caching module to perform caching studies is not a common feature in synthetic workload tools

Hence, a workload generator is proposed, which is designed to address the previous problems, and that mimics the real workload in terms of temporal and spatial locality, and provides a more representative synthetic trace. ELOS also has a high-fidelity trace replay module (called hfreplay), which is used in conjunction with the trace generation module to perform performance studies.

6.2. Synthetic Workload Generation - Approaches

There were a lot of methods that were tried for synthetic trace generation. Some of the important revisions made and approaches taken are explained below. (Note: the first two methodologies were tried mainly by Nohhyun Park, who initially started this research)

6.2.1. Multi-dimensional Histogram Approach: Histograms are a common way to capture the distribution of a variable. When using such an approach, carefully choosing the size of the histogram, and its bin size are important. Consider X is a normally distributed variable, and its mean is μ then, the histogram bins associated with values around mean (μ) will tend to have higher values. The min and max bins of histograms will play an important role understand the range of the function. Here, we dealt with single dimensional histograms first, and then dealt with multi-dimensional histograms. In this approach each dimension of the workload is treated independently, an assumption here is that there is no correlation between them. However, this method was not good at capturing correlations between dimensions, and reproducing desired localities. Refer to figure 25 for details.

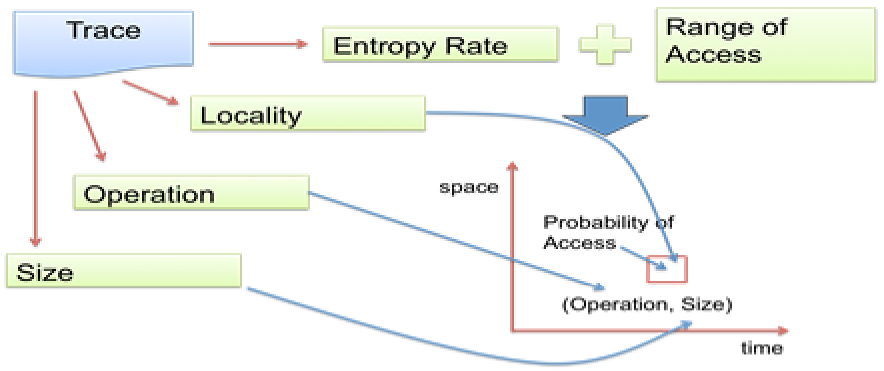


Figure 25: Multidimensional Histogram Based Approach

6.2.2. Forced distributions: Another method to generate synthetic workloads is to look at the distribution of the original trace, and then force similar symmetry in the synthetic trace. However, this method lacks statistical backing, and may lead to erroneous characterization. In one example, the author tried to mimic the block distance of a trace, however, it turned out to be highly error prone. The results are shown in figure 26. This indicated that metrics were correlated to each other, and block distance cannot be generated independently without considering other factors. Apart from that, our finding that there are long range dependencies, lead us to study the patterns in the workloads by doing phase detection, and studying temporal and spatial localities.

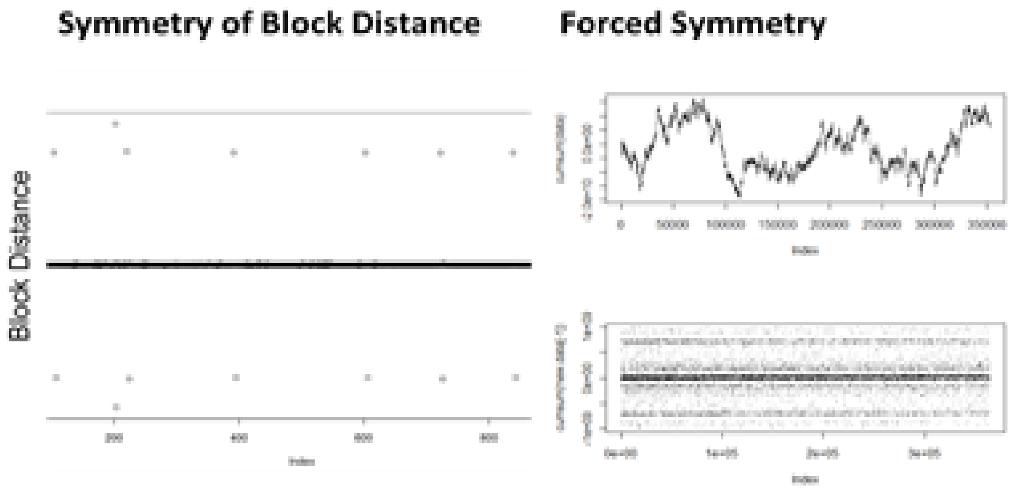


Figure 26: Forced distributions in synthetic workload generator

6.2.3. <u>Block distance convergence</u>: In order to get more realistic values of the block distances, the concept of associated LBNs was used. Where, consecutive LBNs are grouped together, and added to a set. For example, if we have a trace access such as:

LBN accessed: 10, 20, 30, 1000, 5000, 5010, 5020, 99, 1000

Then the LBN associations would be {10, 20, 30} and {5010, 5020}

6.2.4. <u>Stack Distance Convergence using Simulated Annealing:</u> As described in chapter 2, having temporal locality similar to the original trace in synthetic trace is crucial to having representative workloads. When modeling multiple dimensions of the workloads, capturing all possible correlations might get difficult. The method of simulated annealing has been used in lot of biological, chemical and physical parameter extraction and optimization problems. It was probabilistically easy to generate a trace with similar block distance as the original trace. In order to get similar stack distance in the original trace, some form of shuffling was required, and simulated annealing was a good candidate for the same.

6.2.4.1. Simulated Annealing

Simulated annealing is a probabilistic methodology to calculate the global minimum of a function, which might have several local minima points (see figure 27).

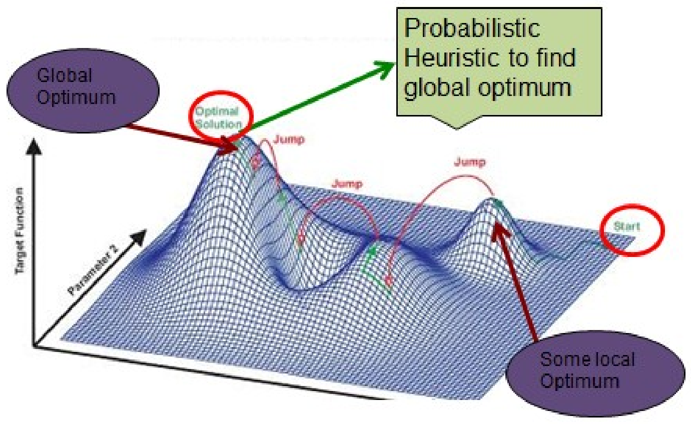


Figure 27: Simulated annealing methodology (source: [33])

In simulated annealing, we first start with an input state, which we could think of a particular ordering the IOs. A cooling schedule is used to control the speed of convergence. Faster the cooling implies lesser confidence in the final result. Hence, the cooling schedule should be chosen carefully. Let the current state be ‘s’. The energy of the current state, E(s), is computed. The energy in our case is the distance (Euclidean or difference in Chi-square metric) for real versus the synthetic trace. If the energy of the current state, E(s), is less than the previous state E (s-1), then the current state is chosen with a probability P. However, if the new state has higher energy, it is not rejected immediately.

Here P = 1 if E(s) < E(s-1); and exp(-((E(s)-E(s-1))/T) otherwise

The process is iterated and finally, the end state will be close to the desired state.

Refer to figure 28 for the algorithm.

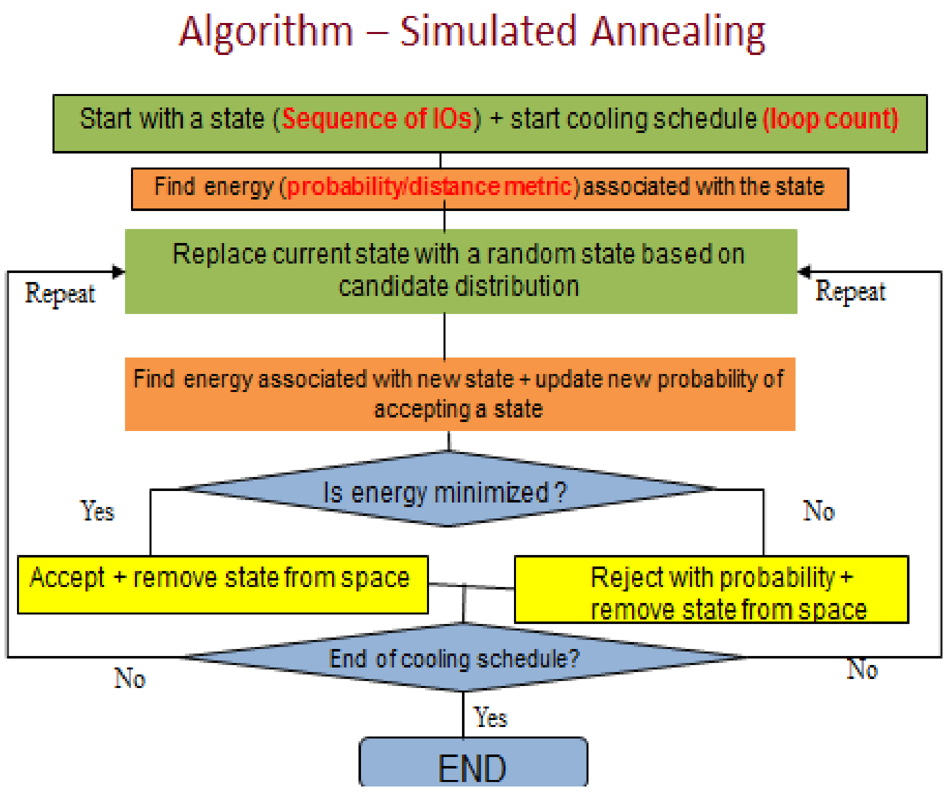

Algorithm – Simulated Annealing
Start with a state (Sequence of IOs) + start cooling schedule (loop count)
Find energy (probability/distance metric) associated with the state
Replace current state with a random state based on candidate distribution
Repeat
Repeat
Find energy associated with new state + update new probability of accepting a state
Is energy minimized ?
Yes
No
Accept + remove state from space
Reject with probability + remove state from space
End of cooling schedule?
No
No
Yes
END

Figure 28: Simulated annealing algorithm

In case of simulated annealing, the convergence time increases with increase in the size of input trace, the reason for this is that - N IOs can have N! possible states. Apart from that, Simulated Annealing is a decision problem and is NP complete. For our case, finding local minima is enough. The time complexity ~ O(n^2) (using neighbor generator function), but the complexity is bound.

6.3.ELOS Workload Generator

ELOS is a probabilistic workload generator framework that is easily usable, and has the support to accept or tune various workload parameter values to mimic real or predicted workloads. Refer to Appendix for details of the code. Additionally, this workload generator has integration with a replay engine to issue trace IOs to a real system, or a storage simulator. To help perform caching studies, this tool allows users to get realistic cache simulation results, and has various inbuilt cache algorithms. Apart from these, one can perform phase detection studies using the added phase detection capability. Identifying phases in input workloads will help to better produce a synthetic trace, which is a good representative of the original trace.

There are a lot of ways in which this work adds to the existing workload generators. Firstly, this tool can accept input traces and study the properties of that application. The tool can be used to create a backend storage repository and compare the synthetic trace with the traces in repository to see if they belong to a particular class of application. Secondly, the tool produces synthetic traces, which match the real application in temporal and spatial locality with high degree of confidence. The existing macro benchmarks like FIO [26] are not fully representative of the input application, in terms of locality. There are certain workload generators which try to mimic locality using zipf like distributions, however, it is important to understand that not all distributions follow zipf law, and hence these trace generators will work best for applications whose stack distance are close to a zipf distribution. Thirdly, the proposed workload generator has a trace replay engine, which can replay traces to the underlying storage device with high fidelity. One of the best features of this trace replay module is that it can control the maximum queue depth seen by the block storage device, and hence, one can perform any kind of realistic load testing or performance studies. Additionally, with a lot of tunable knobs, one can also replay out the trace, exactly as per the requirements. Lastly, in order to speed up the process of characterization, our tool uses various optimizations. One of such optimizations is in the stack distance calculation module, here a O(nlgn) red black tree based stack distance implementation of the stack algorithm is used, as opposed to the traditional $O(n^2)$ algorithm. This tool also uses all standard libraries, which makes it an easy to use and portable tool. The proposed trace generation could be considered as a three step process as shown in figure 29 below.

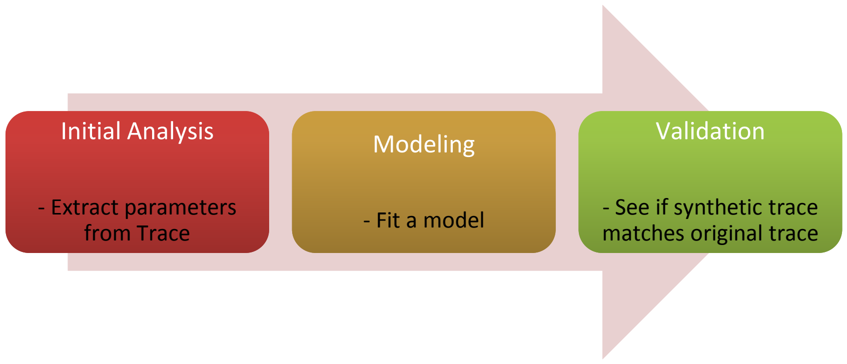

Initial Analysis
- Extract parameters from Trace
Modeling
- Fit a model
Validation
- See if synthetic trace matches original trace

Figure 29: Workload characterization and trace generation steps

6.4. Workload Generator Framework

The Framework used for synthetic workload generation is shown below in figure 29. In this framework, the input is the real trace, in case one has extracted parameters from the trace, then that could help one bypass the parameter extraction module. Once the trace is fed into the system, information such as inter-arrival time, stack distance, read-write ratio, block distance distribution metric etc. are collected and recorded. As described before, these metrics characterize an application, and modifying these could help to recreate the scenario where there are some changes in input application or the associated hardware and software. After capturing the IO workload parameters, these parameters are input to a synthetic trace generation module that takes in the input trace, and generates the synthetic trace based on it's the statistical parameter list. The synthetic trace generated is supposed to mimic the real trace in terms of block distance and stack distance. In order to converge to the correct stack distance values, convergence method based on simulating annealing was used. If the synthetic trace is not good enough then the process is repeated. Additionally, in order to validate the methodology on a real system, integration of the workload generator with a trace replay engine is done. The synthetic and real trace could be replayed on a storage simulator or a real system. The performance comparison of the synthetic versus the real trace is done, and results are statistically quantified. Detailed description of workload generator framework is described below (see figure 30):

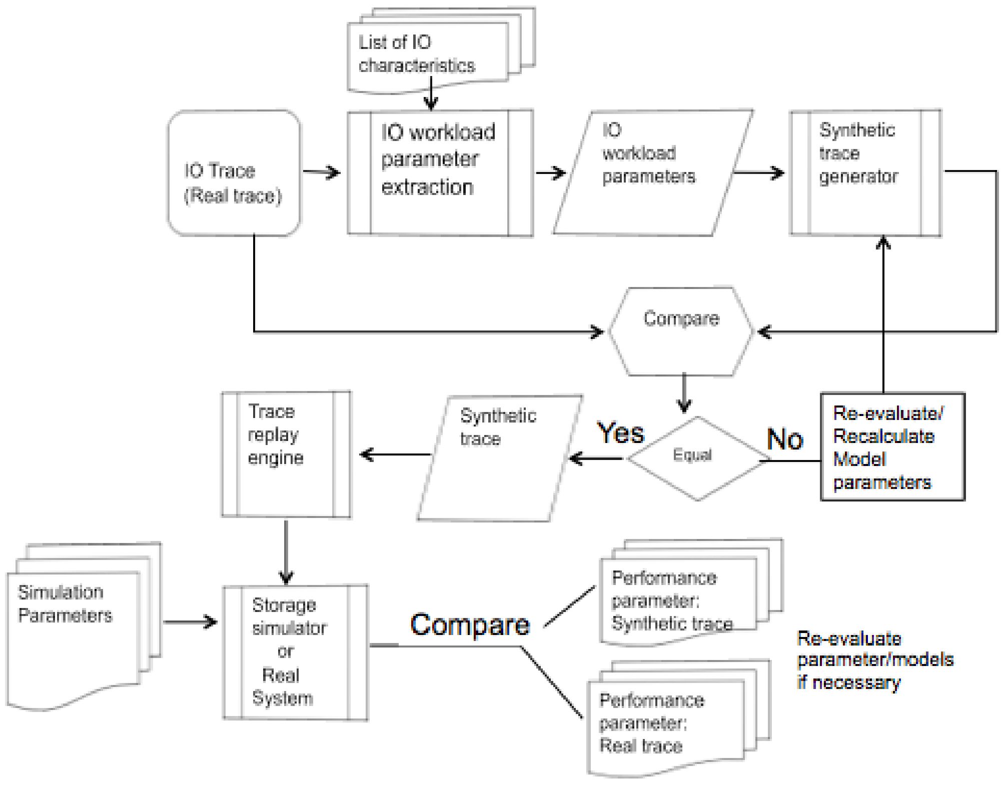
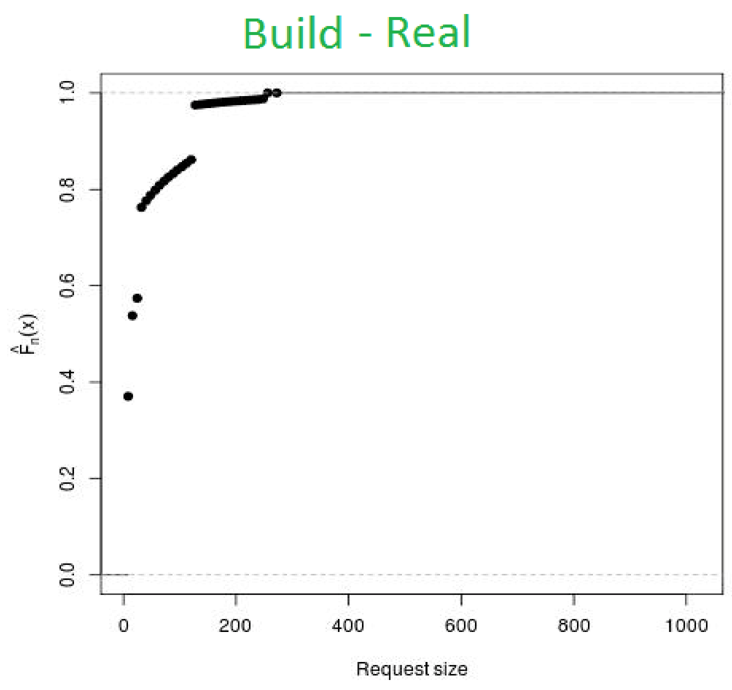

List of IO characteristics
IO Trace (Real trace)
IO workload parameter extraction
IO workload parameters
Synthetic trace generator
Compare
Trace replay engine
Synthetic trace
Yes
Equal
No
Re-evaluate/ Recalculate Model parameters
Simulation Parameters
Storage simulator or Real System
Compare
Performance parameter: Synthetic trace
Re-evaluate parameter/models if necessary
Performance parameter: Real trace

Figure 30: Synthetic Workload Generator Framework

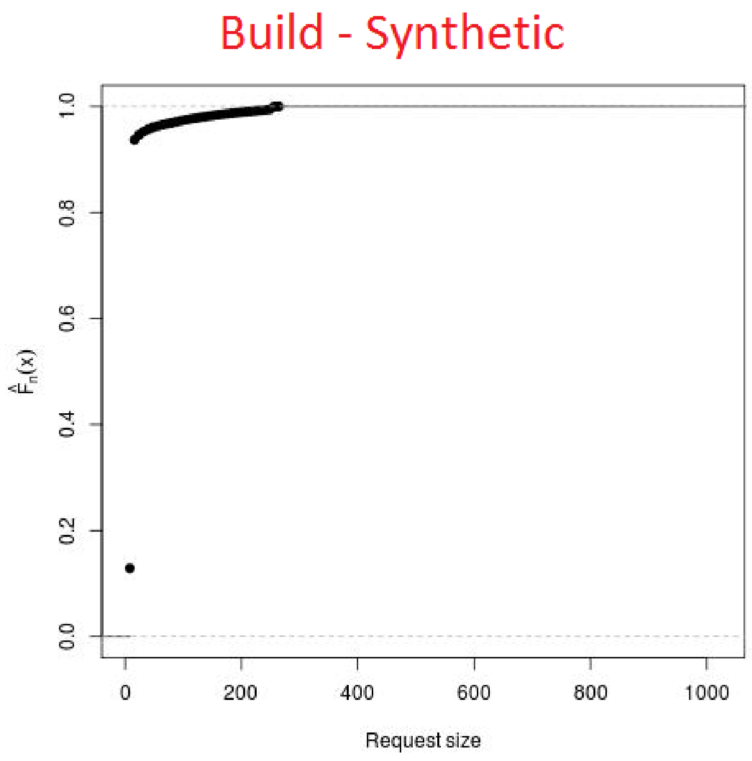
6.5.Results and Validation

6.5.1 Comparing Request Sizes

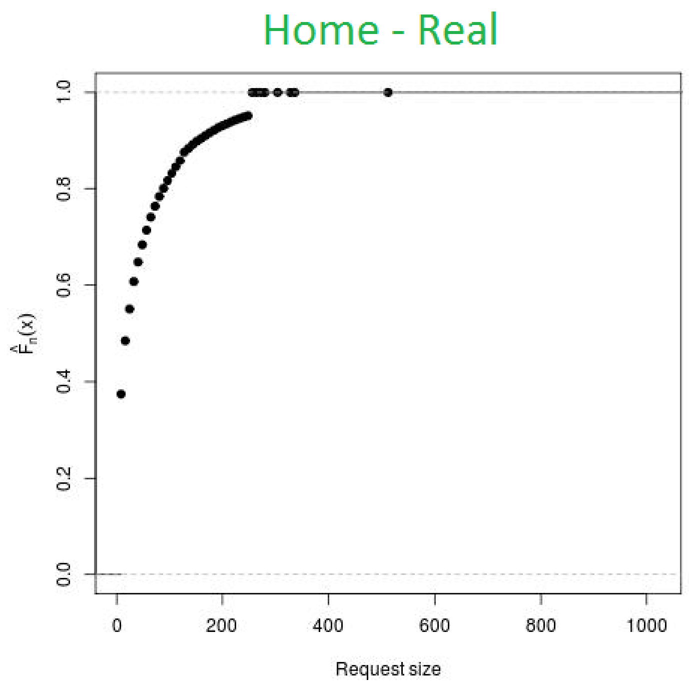
The parameter extraction step uses a script to get the distributions of the input trace in terms of histograms and probability distributions. By doing this it was possible to reproduce similar distributions in the synthetic trace. Figure 31 shows the comparison of the CDFs of real trace versus the synthetic trace, for request size distributions.


Build - Real
$\hat{F}_n(x)$
0.0
0.2
0.4
0.6
0.8
1.0
0
200
400
600
800
1000
Request size

(i)

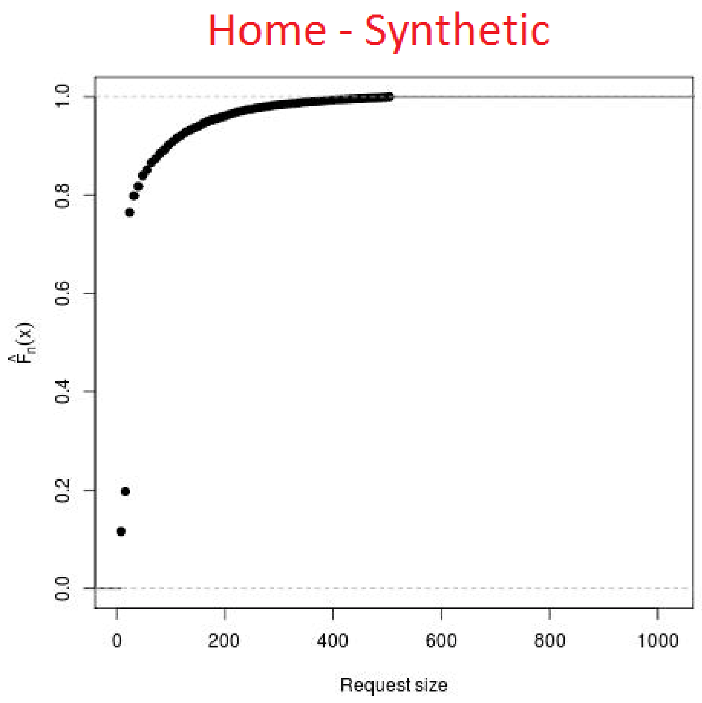

Build - Synthetic
$\hat{F}_n(x)$
0.0
0.2
0.4
0.6
0.8
1.0
0
200
400
600
800
1000
Request size

(ii)



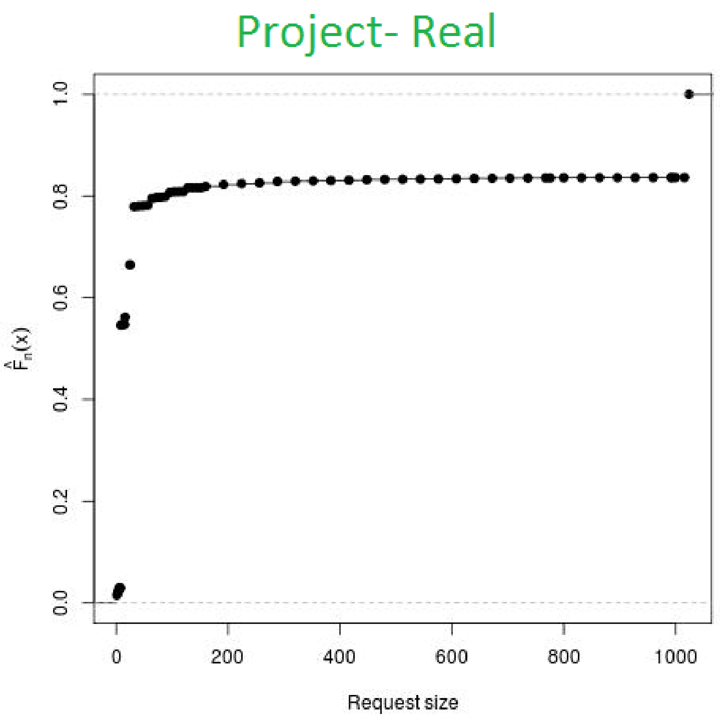
Home - Real
1.0
0.8
0.6
0.4
0.2
0.0
$\hat{F}_n(x)$
0
200
400
600
800
1000
Request size

(iii)

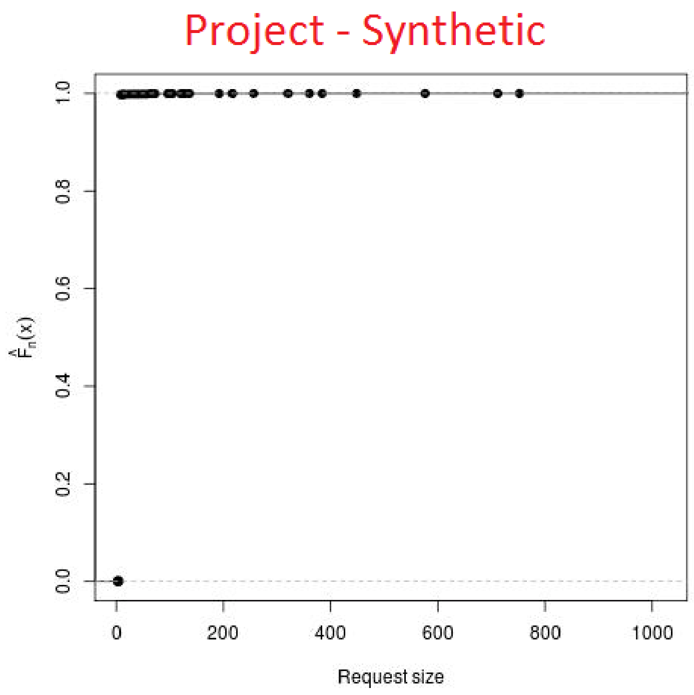


Home - Synthetic
1.0
0.8
0.6
0.4
0.2
0.0
$\hat{F}_n(x)$
0
200
400
600
800
1000
Request size

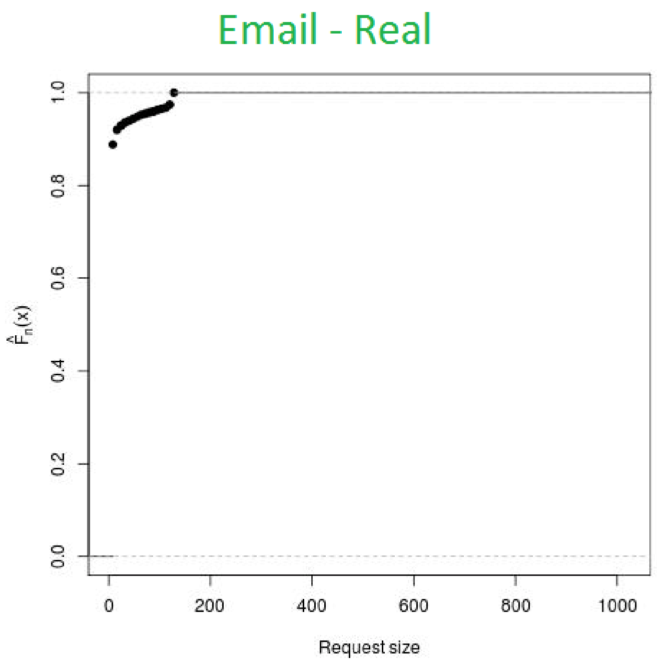
(iv)


Project- Real
$\hat{F}_n(x)$
0.0
0.2
0.4
0.6
0.8
1.0
0
200
400
600
800
1000
Request size

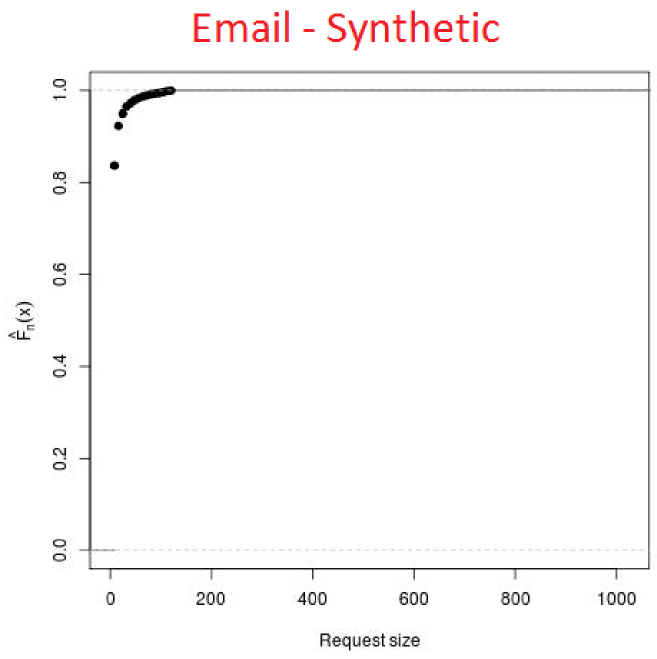
(v)


Project - Synthetic
$\hat{F}_n(x)$
0.0
0.2
0.4
0.6
0.8
1.0
0
200
400
600
800
1000
Request size

(vi)



(vii)



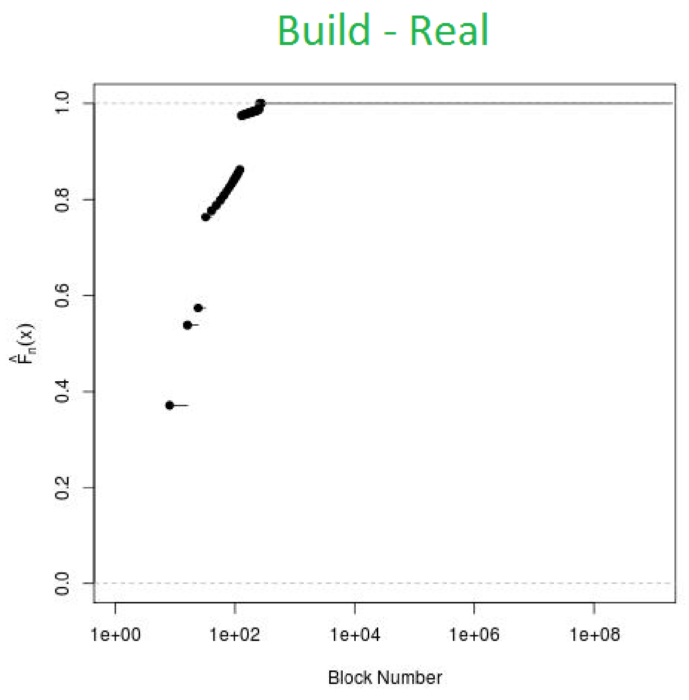
(viii)

Figure 31: CDF of request sizes for real and synthetically generated traces
(i) Build real (ii) Build synthetic (iii) Home real (iv) Home synthetic (v) Project real
(vi) Project synthetic (vii) Email real (viii) Email synthetic

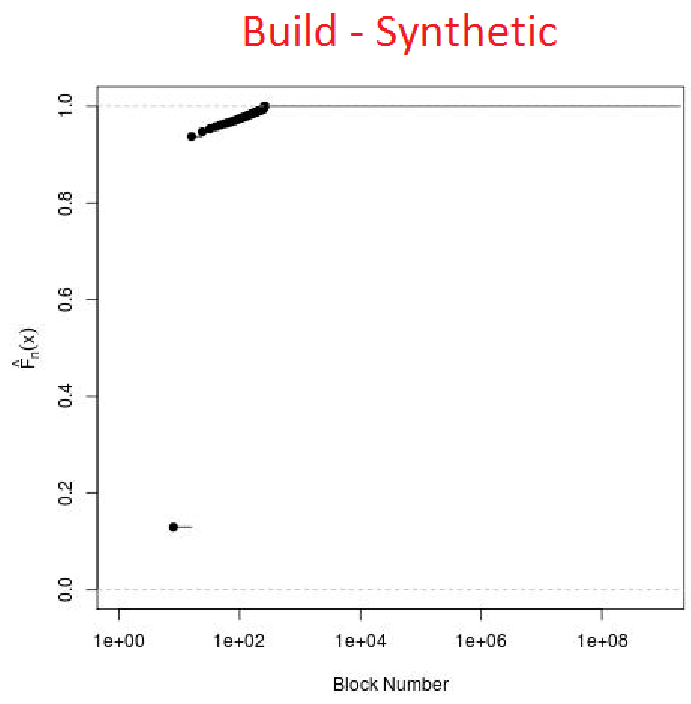
Here, we can see the synthetic trace CDF has request size function, which is more continuous than the real trace. The CDF of real trace has more jumps, indicating more discrete nature. This behavior was prominent for Build and Home trace, but not so much for Project and Email trace.

6.5.2 Comparing Block Distribution

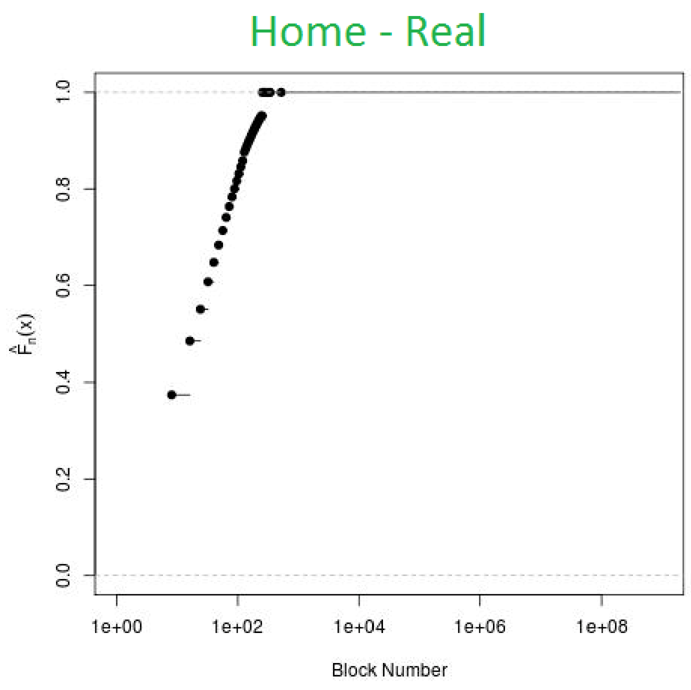
Similar analysis was done by comparing CDFs of block size distributions for real versus the synthetic trace. From figure 32, we can see that Build and Project trace s had a more discrete function for their CDFs, which was not completely captured in the respective synthetic traces. Home and Email trace have very similar CDF for their synthetic traces and the results were within 90% confidence interval.



(i)

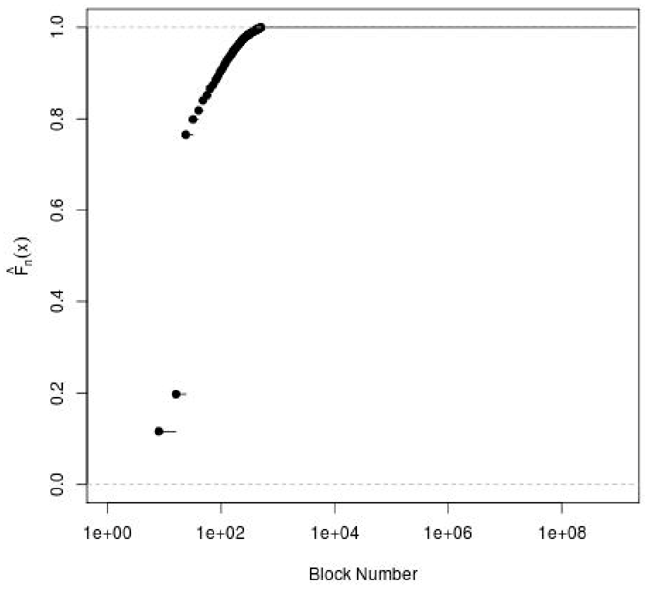

Build - Synthetic
1.0
0.8
0.6
0.4
0.2
0.0
$\hat{F}_n(x)$
1e+00
1e+02
1e+04
1e+06
1e+08
Block Number

(ii)


Home - Real
1.0
0.8
0.6
0.4
0.2
0.0
$\hat{F}_n(x)$
1e+00
1e+02
1e+04
1e+06
1e+08
Block Number

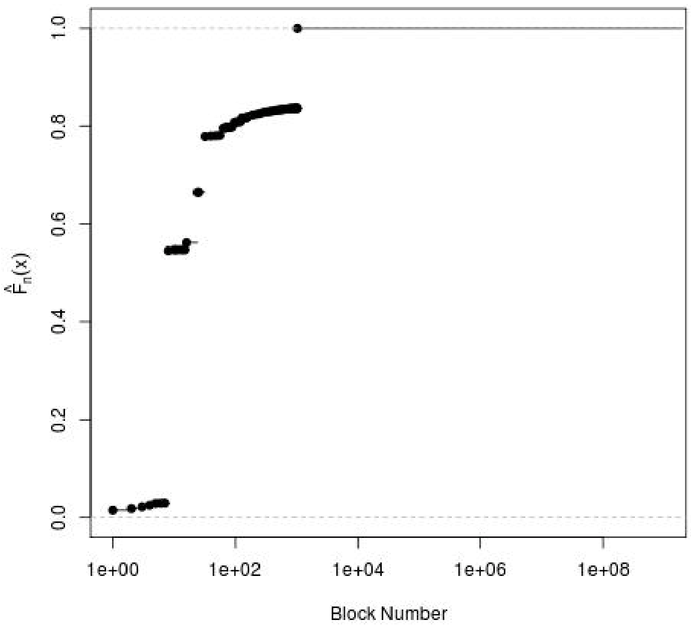
(iii)

Home - Synthetic

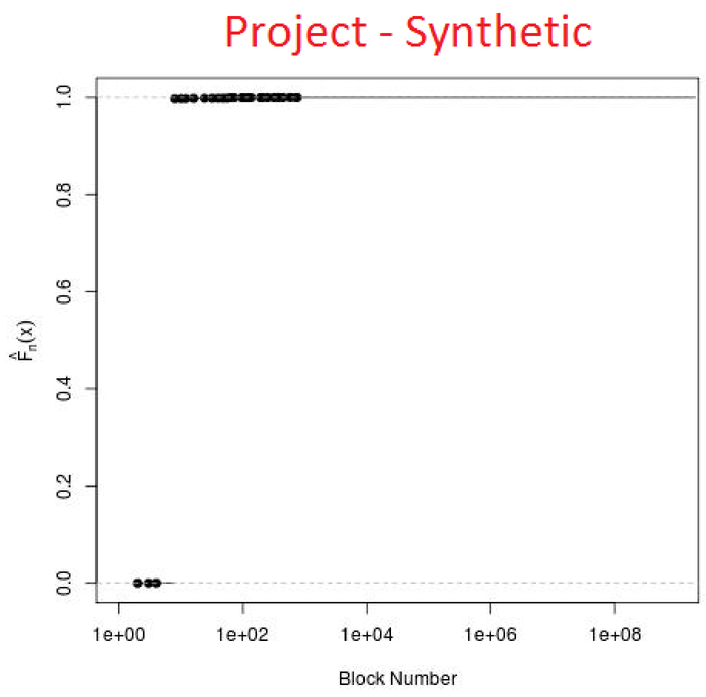


(iv)

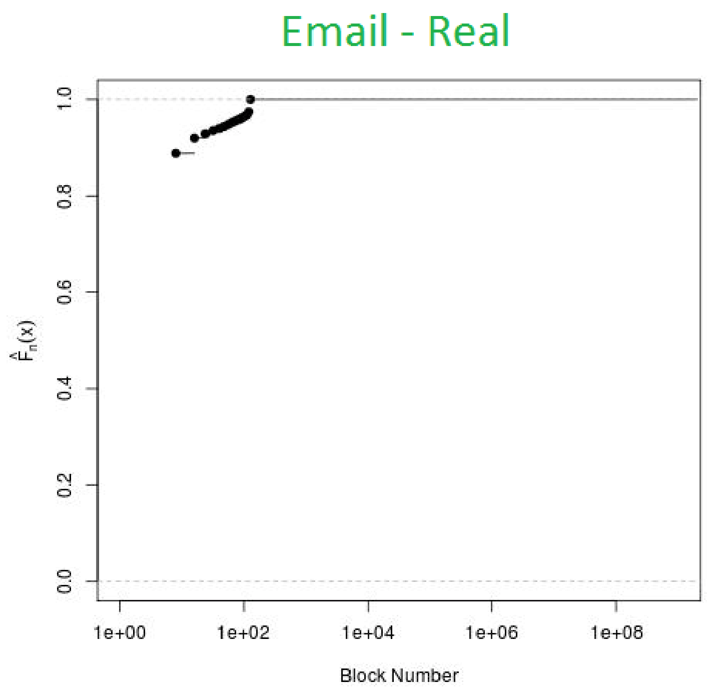
Project - Real



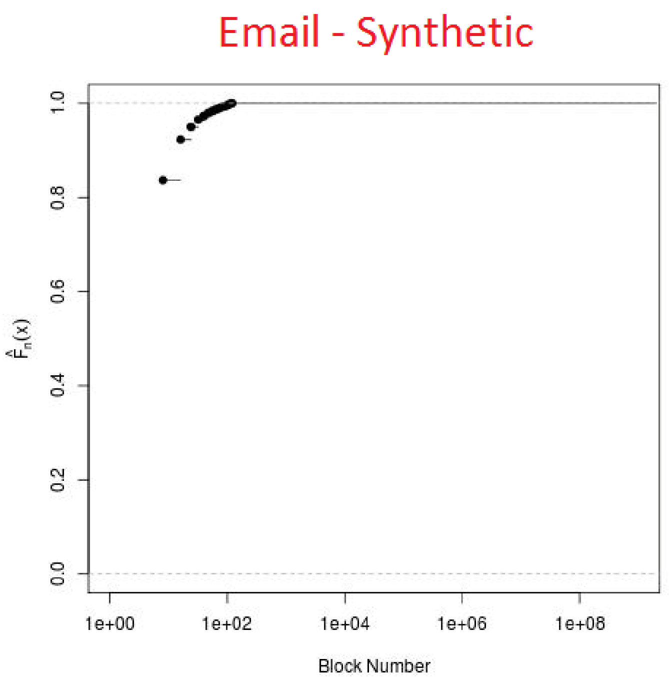
(v)


Project - Synthetic
1.0
0.8
0.6
0.4
0.2
0.0
$\hat{F}_n(x)$
1e+00
1e+02
1e+04
1e+06
1e+08
Block Number

(vi)


Email - Real
1.0
0.8
0.6
0.4
0.2
0.0
$\hat{F}_n(x)$
1e+00
1e+02
1e+04
1e+06
1e+08
Block Number

(vii)



(viii)

Figure 32: CDF of block distributions for real and synthetically generated traces (i) Build real (ii) Build synthetic (iii) Home real (iv) Home synthetic (v) Project real (vi) Project synthetic (vii) Email real (viii) Email synthetic

6.5.3 Comparing Read and Write Ratios

Read write ratios could be captured and reproduced in the synthetic trace, by using simple histograms. We can specify how much percentage reads and how much percentage writes need to be generated, and reproduce them in the workload generator. Another methodology used in our generator is to just generate read or write traces. The generator can extract read or write requests, and generate statistics for just one of those components.

Build (Real and Synthetic): Reads = 97.8%

Home (Real and Synthetic): Reads = 98.4%

Project: (Real and Synthetic): Reads = 87.8%

Email: (Real and Synthetic): Reads = 90.6%

6.5.4 Comparing Stack Distance

Once a trace with similar block access profile is obtained, it was passed through the stack distance convergence module. As described in section 6.2.4, the stack distance convergence takes long time, as the size of trace increases. For a trace with more than 10000 elements, the convergence time increases almost exponentially. This is because the number of possible states in our case would be N! where, N is the number of IOs. One example of a trace convergence for ~10000 IOs is shown below.

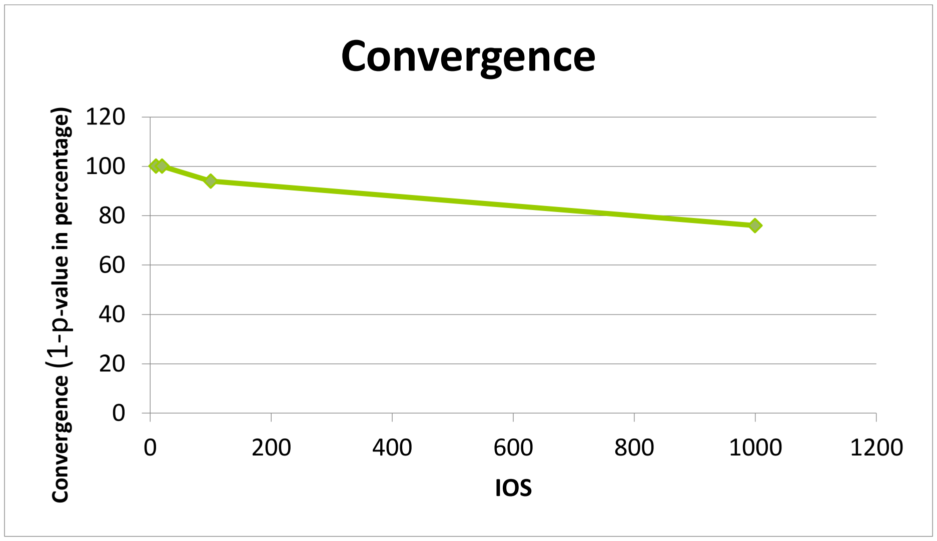


Figure 33: Convergence of simulated annealing methodology

In figure 33, chi-square metric for stack distances was computed for every iteration of simulated annealing. The p-value of the chi-square metric helps to quantify how close the real and synthetic traces are. The similarity metric is defined as p-value*100. The higher the p-value, more similar the real and synthetic traces are. For example if the p-value in iteration is 0.32, then the similarity score would be 32%. In most workloads, convergence of up to 90% was possible by modeling smaller sub-traces.

In order to get more realistic stack distance values, divide and conquer methodology was used. Now, the entire trace is taken, chunked into pieces, and shuffled individually. The visual representation of this process is shown in figure 34.

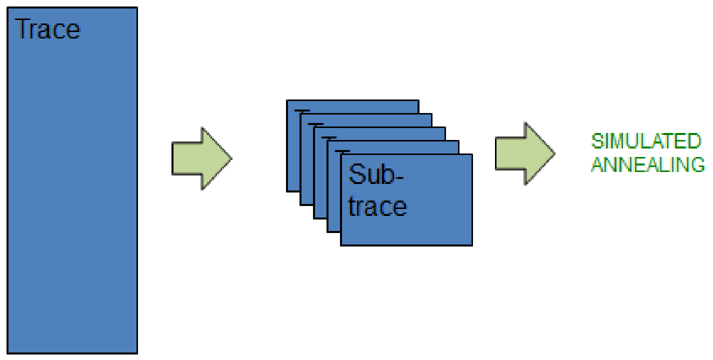


Figure 34: Chunking process for more effective simulated annealing convergence

The hit rates of the real vs. synthetic trace for the entire file and for 1 million MSR trace IOs on a fixed size cache (100KB) were plotted, and the difference in hit rates are shown in figure 35. The original hit ratios for both real and synthetic MSR traces are shown in Figure 36.

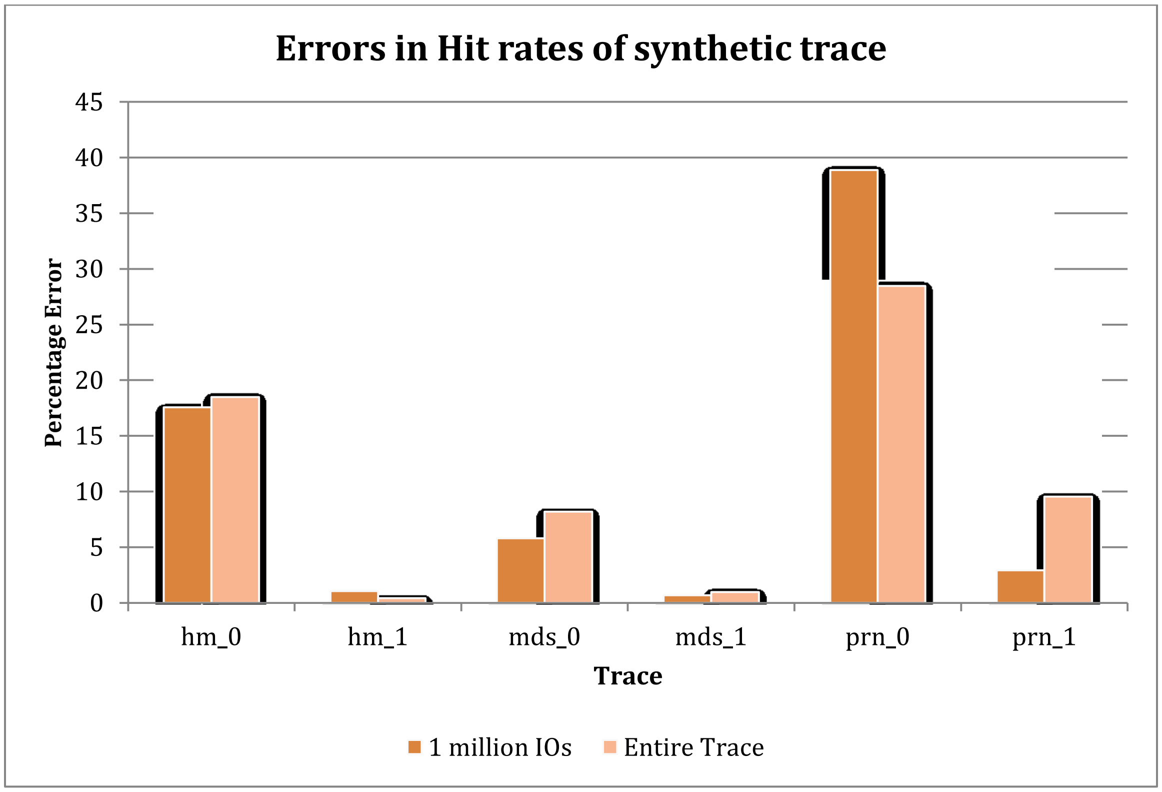


Figure 35: Comparing absolute error in hit rates for synthetic MSR traces for LRU cache

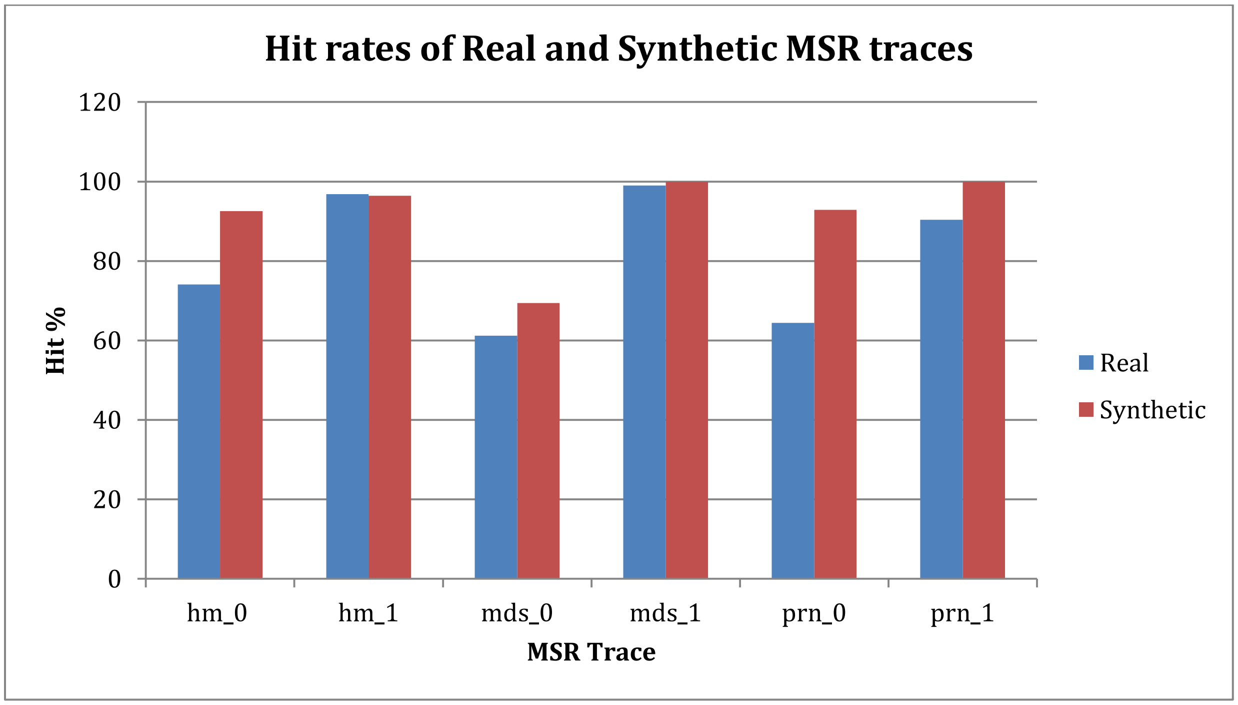


Figure 36: Comparing hit rates for real vs. synthetic MSR traces for LRU cache

We can see that traces like hm_1, mds_0, mds_1, prn_1 had very low error rates, with the error rate of mds_1 as low was 0.67% for 1 million IOs. Whereas hm_0 had an error of 17.6% and prn_0 had a significantly high error rate of 38.8%. And when the entire trace was considered, hm_1 had even much lower error rate of 0.4% (a decrease of 0.6%). Similar decrease in error rate was seen for prn_0 trace. Other MSR traces however showed some increase in error rates. The error rates plotted here have 90% significance level, and measurements repeated 5 times. Hence, we were able to achieve similar ordering and hit rates in synthetic traces.

6.5.5 Tuning Workload Parameters

Many times application behavior might change due to various reasons like change in software, hardware or change in system configuration. Our model helps us capture these changes, by changing the extracted parameters. For example, assume that the system cache is now twice as big, leading to IOs with stack distance less than a particular value to be eliminated, as the data might be in the higher cache level. In this situation, one can easily go through the stack distance profile, and specify to eliminate IOs with stack distances below a threshold. If there are some changes in hardware or software parameters, resulting in different block sizes or read write traffic, then that could also be easily tuned, by modifying the extracted parameters.

6.5.6 Compression

Since, the proposed method allows storing of extracted statistics from the trace, we can achieve compression in the data stored. The trace files that were of the order of few GBs, were compressed by a factor of ~175x. The absolute values of trace sizes are shown in figure 37, and we can clearly see that as the captured trace size increases, the benefit of using synthetic traces also increases. For example, prn_0 trace requires storage space of 0.57GBs, whereas the synthetic trace only requires 56MBs.

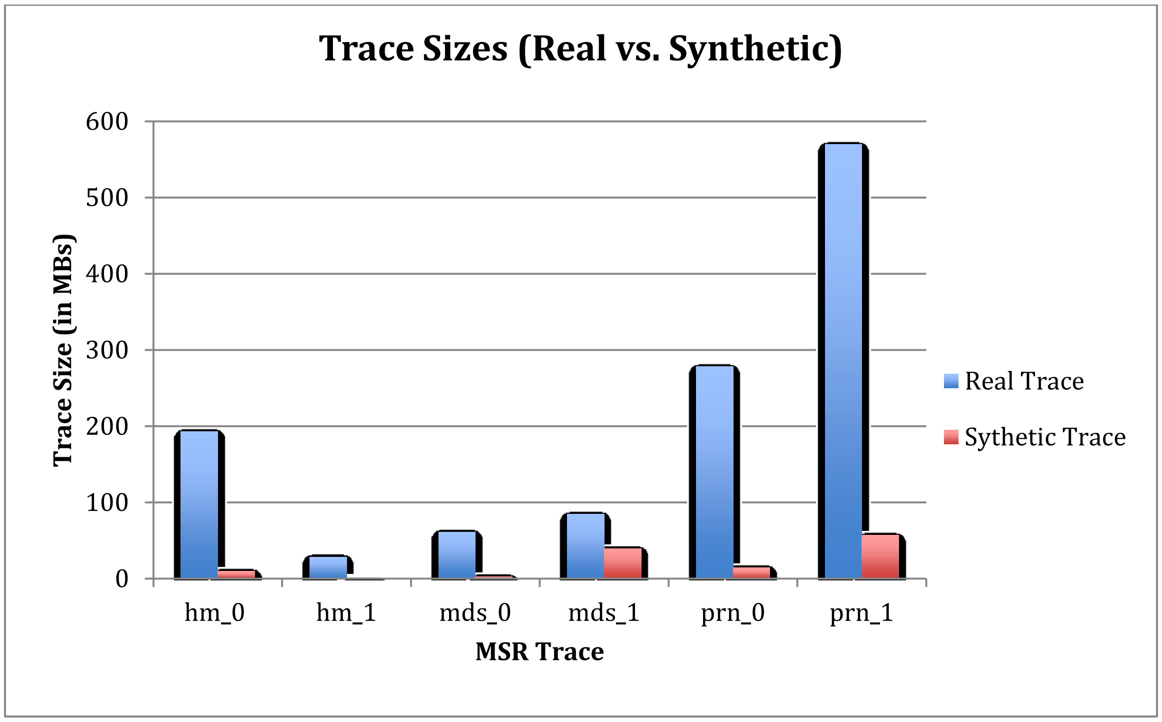


Figure 37: Trace sizes for real and synthetic MSR traces

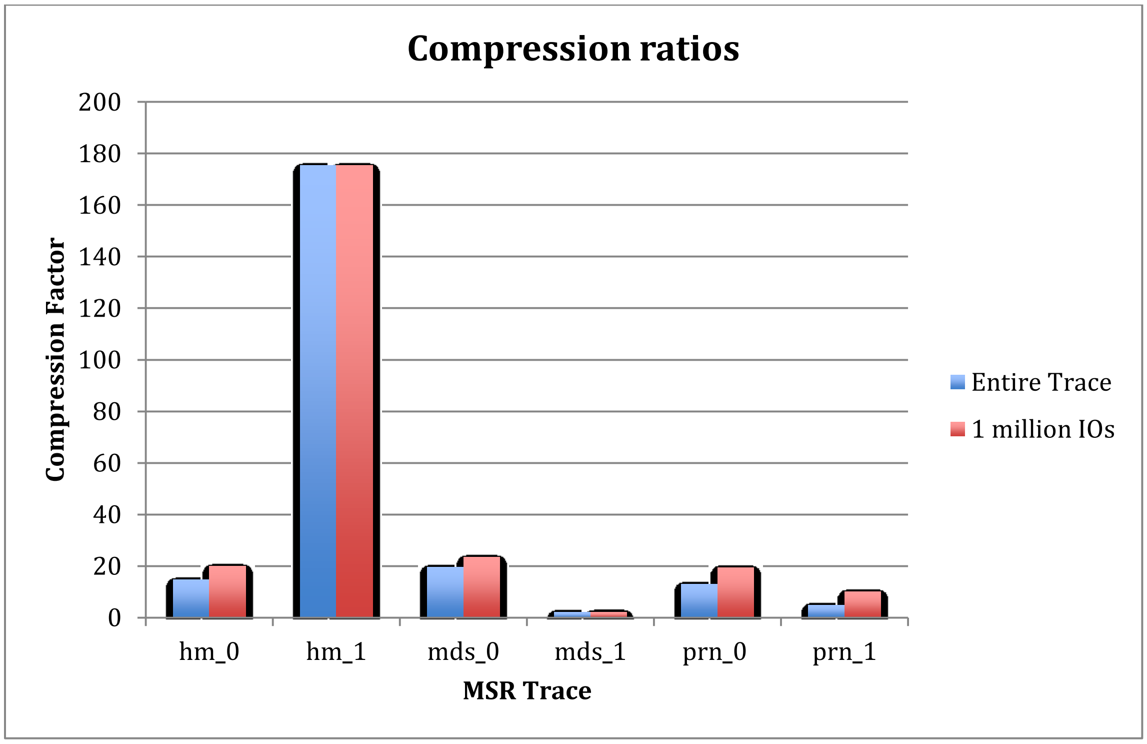


Figure 38: Compression ratios obtained for MSR traces

The compression factors obtained from MSR Cambridge traces are shown in figure 38. Here, we can see that even for a million IOs the compression ratio obtained is significant, and hm_1 trace has a very good compression ratio of 175x. This indicates that hm_1 trace has a very good locality. Our tool can easily reproduce traces with good locality. We can observe this from our synthetic trace as well, and we can observe from figure 35 that the error rate in cache hits for synthetic trace was only 0.4%, which is low. mds_1 trace has poor locality, and hence the compression ratio was also poor. Overall, good space savings were observed across all MSR traces analyzed.

6.6 Trace Replay Module

For replaying the trace hfreplay replay engine (figure 39) was used and this workload generator was developed by Alireza Haghdoost, as student from CRIS, UMN.

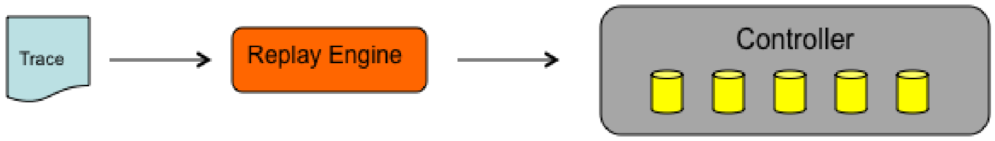


Figure 39: Replaying a trace using Hfreplay tool

This replay tool could be easily used by modifying the input trace format in the ELOS workload generator, and the characteristics of the trace on a real system could be observed.

6.7 Summary

ELOS (Entropy Locality Operation Size) workload generator was developed, that produces similar temporal and spatial locality as the original block IO trace. The workload generator framework consists of a parameter extraction module that captures various probabilities and correlations in workloads. Then using the gathered information a model is fit, and the phases that exist are studied. The next step is to generate a synthetic trace that is similar to the real trace in terms of read-write ratio, block access profile and block distances. After this stage, the synthetic trace is passed through a stack distance convergence module. In the end, one can successfully produce a synthetic trace that is similar to the original trace with more than 90% confidence in results. The errors in hit rates varied between 0.67% for mds_1 to 38.8% for traces like prn_0 for 1 million IOs and between 0.4% for hm_1 and 28.4% for pn_0 when considering the entire trace. Good compression ratios were obtained for the traces, and even for traces with 1 million IOs, the compression ratios reached ~175x, which leads to considerable amount of space savings.

Chapter 7
Conclusions and Future Work

7.1 Conclusion

In this thesis, the author has shown that locality in workloads can be captured using statistical analysis, and it is possible to produce workloads with desired locality. It is shown that just describing basic workload parameters like read-write ratio, random or sequential IOs and block size ratios in synthetic workload generators are not enough to achieve the required long-range dependencies and localities. Having different ordering of the traces implies different locality, and the trace will not be representative.

To completely describe a workload (w), we need a tuple 'W' such that W = {r, p, s, t, τ, s, b, S}; where r = request type, p= access pattern, l =request size, t=arrival time, τ = inter-arrival time, s = burstiness, b=block distance, S= stack distance. These characteristics uniquely describe the input trace. Stack distance is a measure of temporal locality of the workloads, and block distance is the measure of spatial locality. Having similar stack and block distances would mean having similar cache statistics.

A phase is defined to be the subset of a workload that has similar behavior. Clustering, multi-clustering and model fitting were used to identify phases in input trace. Multi-clustering tends to produce good results due to majority based voting. By maintaining a historical repository one can build upon the model, and improve the phase detection during runtime as well.

Disk IO workloads are shown to have long-range dependency in auto-correlation, and hurst parameter could be used for understanding this dependence. Workloads are said to be similar if H value is between 0.5 and 1. IO trace parameters like request size, block accessed and read/write request showed gradually decaying autocorrelation function, whereas the inter-arrival time showed very less auto-correlation, indicating that the workload is possibly bursty. ARIMA models showed good predictions for request sizes and block sizes. Phases detected provided to be a good cut off points in the trace for diving the trace, and performing modeling.

Simulated Annealing could be used for stack distance convergence for synthetic trace. The similarity metric used here, could be the Euclidian distance or chi-square value of real vs. synthetic trace. For a trace with more than 10000 elements, the convergence time increases almost exponentially. This is because the number of possible states in our case would be N!, where, N is the number of IOs.

ELOS (Entropy Locality Operation Size) workload generator was developed that produces similar temporal and spatial locality as the original block IO trace. The workload generator framework consists of a parameter extraction module that captures various probabilities and correlations in workloads. Then using the gathered information a model is fit, and the phases that exist are studied. The next step is to generate a synthetic trace that is similar to the real trace in terms of read-write ratio, block access profile and block distances. After this stage, the synthetic trace is passed through a stack distance convergence module. The errors in hit rates varied between 0.67% for mds_0 to 38.8% for traces like prn_0 in case of 1 million IOs and between 0.4% for hm_1 and 28.4% for pn_0 when considering the entire trace. Good compression ratios were obtained for the traces, and even for traces with 1 million IOs, the compression ratios reached ~175x, which leads to considerable amount of space savings. In the end, one can successfully produce a synthetic trace that is similar to the original trace, and replay it using the provided Hfreplay tool.

7.2 Future Work

An initial framework for producing more representative synthetic workloads is provided. Stack distance is found to be good for studying LRU cache replacement policy, and the author hopes to add other replacement policies as well to get more cache statistics. Another version of the adaptive replacement cache is developed, and would be added to the next version of the workload generator. Additionally, there are a lot of tunable parameters that could be changed in the current design, but there are no guidelines that exist for how to tune them. Some suggestions will be added to guide the user to better tune their workloads if necessary. Design of experiments concepts will be incorporated to find the best workload configurations for a given hardware, and see how the behavior of input application needs to change to extract maximum benefits. Additionally, the author hopes to perform characterization of cloud and VM workloads, and understand their characteristics.

References

[1] Traeger, Avishay, Erez Zadok, Nikolai Joukov, and Charles P. Wright. "A nine year study of file system and storage benchmarking." *ACM Transactions on Storage (TOS)* 4, no. 2 (2008): 5.

[2] Kurmas, Zachary, Kimberly Keeton, and Kenneth Mackenzie. "Synthesizing representative I/O workloads using iterative distillation." In *Modeling, Analysis and Simulation of Computer Telecommunications Systems, 2003. MASCOTS 2003. 11th IEEE/ACM International Symposium on*, pp. 6-15. IEEE, 2003.

[3] Iosup, Alexandru, Dick HJ Epema, Carsten Franke, Alexander Papaspyrou, Lars Schley, Baiyi Song, and Ramin Yahyapour. "On grid performance evaluation using synthetic workloads." In *Job Scheduling Strategies for Parallel Processing*, pp. 232-255. Springer Berlin Heidelberg, 2007.

[4] Bahga, Arshdeep, and Vijay Krishna Madisetti. "Synthetic workload generation for cloud computing applications." *Journal of Software Engineering and Applications* 4, no. 07 (2011): 396.

[5] Ganger, Gregory R. "Generating representative synthetic workloads: An unsolved problem." In *in Proceedings of the Computer Measurement Group (CMG) Conference*. 1995.

[6] Pfneiszl, Hannes, and Gabriele Kotsis. *Synthetic workload generation for parallel processing systems*. Springer Berlin Heidelberg, 1996.

[7] Mehra, Pankaj, and Benjamin Wah. "Synthetic workload generation for load-balancing experiments." *IEEE Concurrency* 3, no. 3 (1995): 4-19.

[8] Eeckhout, Lieven, Koen De Bosschere, and Henk Neefs. "Performance analysis through synthetic trace generation." In *Performance Analysis of Systems and Software, 2000. ISPASS. 2000 IEEE International Symposium on*, pp. 1-6. IEEE, 2000.

[9] Ahmad, Irfan. "Easy and efficient disk I/O workload characterization in VMware ESX server." In *Workload Characterization, 2007. IISWC 2007. IEEE 10th International Symposium on*, pp. 149-158. IEEE, 2007.

[10] Kavalanekar, Swaroop, Bruce Worthington, Qi Zhang, and Vishal Sharda. "Characterization of storage workload traces from production windows servers." In *Workload Characterization, 2008. IISWC 2008. IEEE International Symposium on*, pp. 119-128. IEEE, 2008.

[11] Gulati, Ajay, Chethan Kumar, and Irfan Ahmad. "Storage workload characterization

and consolidation in virtualized environments." In *Workshop on Virtualization Performance: Analysis, Characterization, and Tools (VPACT)*. 2009.

[12] Riska, Alma, and Erik Riedel. "Disk Drive Level Workload Characterization." In*USENIX Annual Technical Conference, General Track*, pp. 97-102. 2006.

[13] Carns, Philip, Robert Latham, Robert Ross, Kamil Iskra, Samuel Lang, and Katherine Riley. "24/7 characterization of petascale I/O workloads." In *Cluster Computing and Workshops, 2009. CLUSTER'09. IEEE International Conference on*, pp. 1-10. IEEE, 2009.

[14] Trace analysis report LCG (http://gwa.ewi.tudelft.nl/datasets/gwa-t-11-lcg/report/)

[15] Gregg, B.D. (2014, May 29). perf Examples [Web log post]. Retrieved from http://www.brendangregg.com/perf.html.

[16] IOMETER, Team. "IOMETER: I/O subsystem measurement and characterization tool." (1997).

[17] Islam, Sadeka, Jacky Keung, Kevin Lee, and Anna Liu. "Empirical prediction models for adaptive resource provisioning in the cloud." *Future Generation Computer Systems* 28, no. 1 (2012): 155-162.

[18] Ding, Chen, and Yutao Zhong. "Reuse distance analysis." *University of Rochester, Rochester, NY* (2001).

[19] Delimitrou, Christina, Sriram Sankar, Kushagra Vaid, and Christos Kozyrakis. "Accurate modeling and generation of storage I/O for datacenter workloads." In*Proceedings of the 2nd Workshop on Exascale Evaluation and Research Techniques, EXERT, Newport Beach, CA (March 2011)*. 2011.

[20] Al-Kiswany, Samer, Abdullah Gharaibeh, and Matei Ripeanu. "The case for a versatile storage system." *ACM SIGOPS Operating Systems Review* 44, no. 1 (2010): 10-14.

[21] Wong, Theodore M., and John Wilkes. "My Cache Or Yours?: Making Storage More Exclusive." In *USENIX Annual Technical Conference, General Track*, pp. 161-175. 2002.

[22] Palanivel, Keerthi, Chow, Kingsum., Ban, Khun, & Lilja, David. J. "A Stepwise Approach to Software-Hardware Performance Co-optimization Using Design of Experiments. " In *in Proceedings of the Computer Measurement Group (CMG) Conference* (2013)

[23] Draper, N. R. (1985). Plackett and Burman designs. *Encyclopedia of Statistical Sciences*.

[24] Chen, Shimin, Phillip B. Gibbons, Todd C. Mowry, and Gary Valentin. "Fractal prefetching B+-trees: Optimizing both cache and disk performance." In*Proceedings of the 2002 ACM SIGMOD international conference on Management of data*, pp. 157-168. ACM, 2002.

[25] Hong, Bo. "Techniques for Synthetic Input/output Workload Generation." PhD diss., UNIVERSITY OF CALIFORNIA SANTA CRUZ, 2002.

[26] Axboe, Jens. "Fio-flexible io tester." *uRL: http://freecode. com/projects/fio*(2008).

[27] Rusling, David A. "The linux kernel." (1999).

[28] Kasavajhala, Vamsee. "Solid state drive vs. hard disk drive price and performance study." *Proc. Dell Tech. White Paper* (2011): 8-9.

[29] Fraley, Chris, and Adrian E. Raftery. "How many clusters? Which clustering method? Answers via model-based cluster analysis." *The computer journal* 41, no. 8 (1998): 578-588.

[30] Samorodnitsky, Gennady. "Long range dependence." *Foundations and Trends® in Stochastic Systems* 1, no. 3 (2007): 163-257.

[31] Berryman, Alex, Prasad Calyam, Matthew Honigford, and Albert M. Lai. "Vdbench: A benchmarking toolkit for thin-client based virtual desktop environments." In *Cloud Computing Technology and Science (CloudCom), 2010 IEEE Second International Conference on*, pp. 480-487. IEEE, 2010.

[32] Norcott, W. "Iozone benchmark." *See www. iozone. org* (2001).

[33] “Mathematical Process Optimization,” http://www.frankfurt-consulting.de/English/optimierung_us.htm

Appendix

ELOSv0.4 (Entropy Locality Operation Size) – Workload Generator

Modules

1. Characterize
2. Cache
3. Elos Trace
4. Hfreplay
5. Parse_sg
6. synth_elos
7. synth_new
8. testSet
9. SA (simulated annealing)

Characterize Module

File name: characterize.R

Author: Keerthi Palanivel

Usage: Rscript characterize.R -b(for block) -r(for request size) -iat(for inter-arrival time) input_file_name output trace_type("Seagate" or "MSR")

Details: This file is used for plotting workload characteristics (of Seagate and MSR Traces)
Options: In the current version, the following characteristics could be plotted using the given options

Block Distance (-b)

Input: Rscript characterize.R –b input_trace_file_to_characterize Output_file Seagate
Output: You will have a plot file created named Output_file_block.jpg

Request Size (-r)

Input: Rscript characterize.R –r input_trace_file_to_characterize Output_file Seagate
Output: You will have a plot file created named Output_file_req.jpg

Inter-arrival time (-iat)

Input: Rscript characterize.R -iat input_trace_file_to_characterize Output_file Seagate
Output: You will have a plot file created named Output_file_iat.jpg

Read write ratio (-rw)

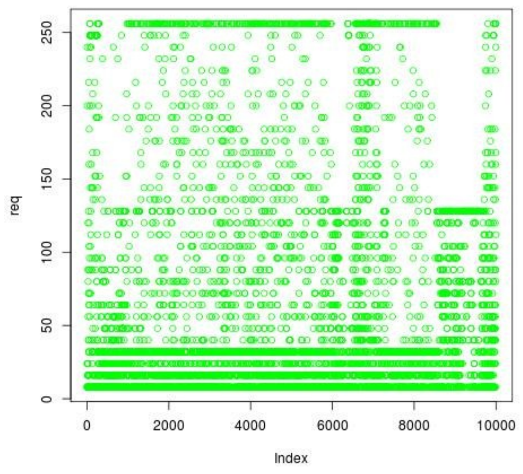
Input: Rscript characterize.R –rw input_trace_file_to_characterize Output_file Seagate
Output: You will have a plot file created named Output_file_rw.jpg

In case one wants to plot all the options then use:

Plot all (-a)

Input: Rscript characterize.R –rw input_trace_file_to_characterize Output_file Seagate
Output: You will have a plot files created named
Output_file_block.jpg
Output_file_req.jpg
Output_file_iat.jpg
Output_file_rw.jpg

Sample Trace Plot



Cache Module

File name: cache.C
Author: Nohhyun Park
Description: LRU cache; has constructor and destructor for cache, and has functionality for cache lookups, find and return stack size.

File name: cache.H
Author: Nohhyun Park
Description: Header file for cache descriptions

File name: cache_sim.cfg
Author: Keerthi Palanivel and Nohhyun Park
Description: Configuration file for cache description; can specify trace file, trace_type; duration, output folder, number of threads to use, roa (read on arrival), rla (real look-ahead).

```
#------- Sample File --------------
trace = [input trace name]
type = MSR
duration = 10
out = [output folder name]
thread = [number of threads]
roa = [number of elements to read]
rla = [number of elements to look
ahead]
```

File name: cacheTest.C
Author: Nohhyun Park
Description: Perform cache test for cache definition in cache.H

File name: csim.C
Author: Keerthi Palanivel and Nohhyun Park
Description: Main function parses the trace file, extracts parameters and puts it in a

record, and in histograms. Here if iosToProcess option is -1, then all IOs are processed. Configuration could be added to describe number of ios to process, trace name, duration, time to process, number of threads to use, output folder, roa and rla values. Histograms for logical_block_number and size are described. After processing, a synthetic trace file is written to the output folder.

Usage:

Generic options:

```
  -v [ --version ]              print version string
  --help                        print help message
  -c [ --config ] arg             configuration file path
```

Configuration:

```
  -i [ --ios ] arg (=-1)                  Number of IOs to process
  -t [ --trace ] arg                       Input trace file path
  -d [ --duration ] arg (=18446744073709551615)
                                                   Time to process in minutes
  --verbose                                 verbose priting of progress
  --thread arg (=1)                        enable thread
  -o [ --out ] arg (=out)                          Directory to store output traces
  -t [ --type ] arg (=Seagate)             Trace - MSR or Seagate
  --roa arg                               ROA values
  --rla arg                                RLA values
```

File name: extract.C
Description: Extract information from trace file.

File name: fread.cpp
Author: Keerthi Palanivel
Description: Utility file to read input file and parse; create histograms from read files and generate traces using stored histograms

File name: fread.h
Author: Keerthi Palanivel
Description: Utility file header to read and parse input files

File name: fread_test.cpp
Author: Keerthi Palanivel
Description: Test file to test fread functionality

File name: hist.cpp
Author: Keerthi Palanivel
Description: Utility file for histograms

File name: hist.h
Author: Keerthi Palanivel
Description: Utility headers file for histograms

File name: lru.cpp
Author: Keerthi Palanivel
Description: LRU cache description; has functionality for initialize, insert, search, remove, evict from cache and print statistics.

File name: lru.h
Author: Keerthi Palanivel
Description: Header files for LRU cache

File name: lru_test.cpp
Author: Keerthi Palanivel
Description: Test file to test lru cache functionality

File name: record.C
Author: Nohhyun Park
Description: Stores records of extracted parameters

File name: record.H
Author: Nohhyun Park
Description: Header file for record file

File name: sa.C
Author: Keerthi Palanivel
Description: [old version of simulated annealing]

File name: stackAlgo.C
Author: Nohhyun Park and Keerthi Palanivel
Description: Stack distance calculation algorithm (initial), and utility to print summary statistics.

File name: stackAlgo.H
Author: Nohhyun Park and Keerthi Palanivel
Description: Header files for stack algorithm

File name: stackTest.C
Author: Nohhyun Park
Description: Test file to test stack algorithm

File name: summary.C
Author: Nohhyun Park
Description: Hash lookups and summary function

File name: summary.H
Author: Nohhyun Park
Description: Header files for summary function

File name: summaryTest.C
Author: Nohhyun Park
Description: Test files for summary function

File name: trace.C
Author: Nohhyun Park
Description: Functions for trace records

File name: trace.H
Author: Nohhyun Park
Description: Header file for trace file to store trace records

File name: Makefile
Description: Makefile for cache module

File name: synthetic.cpp
Author: Keerthi Palanivel
Description: Calls functions to extract histograms from the traces, and store them in intermediate files; using the stored information generate synthetic trace (with help of generate.sh)

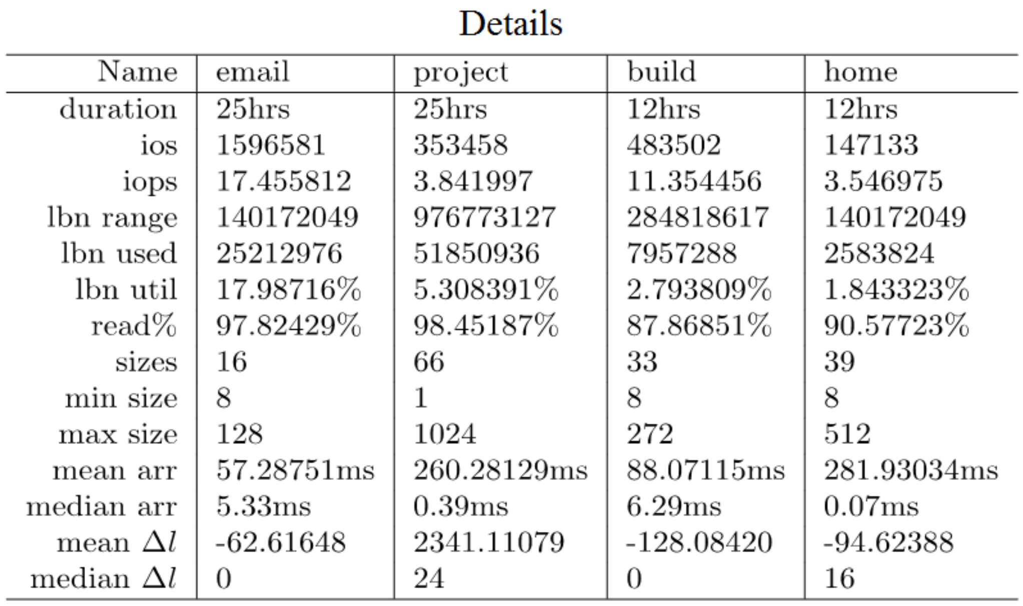
File name: generate.cfg
Author: Keerthi Palanivel
Description: Configuration files to generate synthetic MSR traces. Specify input file, output folder and maximum number of IOs to process. Here '-1' for maximum IOs means that process the entire file.

File name: generate.sh
Author: Keerthi Palanivel
Description: Wrapper script to generate synthetic MSR traces. This script also does cache analysis for a set size cache, and prints the cache statistics (hits and misses) at the end. The cache used here is a simple lru-cache, and the model can be easily changed to add complex caches.

Elos Trace

Traces:

1. Build – 17.52 Megabytes
2. Email – 7.88 Megabytes
3. Project -3.32 Megabytes
4. Home – 6.08 Megabytes



Details

Name	email	project	build	home
duration	25hrs	25hrs	12hrs	12hrs
ios	1596581	353458	483502	147133
iops	17.455812	3.841997	11.354456	3.546975
lbn range	140172049	976773127	284818617	140172049
lbn used	25212976	51850936	7957288	2583824
lbn util	17.98716%	5.308391%	2.793809%	1.843323%
read%	97.82429%	98.45187%	87.86851%	90.57723%
sizes	16	66	33	39
min size	8	1	8	8
max size	128	1024	272	512
mean arr	57.28751ms	260.28129ms	88.07115ms	281.93034ms
median arr	5.33ms	0.39ms	6.29ms	0.07ms
mean Δl	-62.61648	2341.11079	-128.08420	-94.62388
median Δl	0	24	0	16

Hfreplay

---

hfreplay 2.2
High Fidelity Workload Replay Engine
Authors: Jerry Fredin, Ibra Fall, Alireza Haghdoost, Sai Susarla, Weiping He
Center for Research in Intelligent Storage (CRIS)

University of Minnesota
http://cris.cs.umn.edu

----

parse_sg

Author: Nohhyun Park
Details: This folder contains functions to convert binary Seagate trace to text file.

File name: gen_txt.sh
Details: Automated script to parse Seagate trace (calls sg_tp internally).

File name: gen_RData.sh
Details: Automated script (calls traceT2R.R internally)

----

synth_elos; synth_new

Description: Folder to store synthetic traces

----

testSet

Traces for test purpose; these traces are a subset of the original trace.

SA (Simulated Annealing)

Description: This module is used for stack distance convergence

File name: hash_table.c
Author: Nohhyun Park
Details: Hash table utility file; it has procedures for hash table entry, initialization, look up, add and delete elements.

File name: hash_table.h
Author: Nohhyun Park
Details: Header file for hash table utility

File name: lru_stack.c
Author: Nohhyun Park and Keerthi Palanivel
Details: LRU stack utility file

File name: lru_stack.h
Author: Nohhyun Park
Details: Header file for LRU stack utility.

File name: main.c
Author: Keerthi Palanivel
Details: Main file

File name: red_black_tree.c
Author: Nohhyun Park (modified by Keerthi Palanivel)
Details: Red black tree structure utility; used for stack distance algorithm.

File name: red_black_tree.h
Author: Nohhyun Park
Details: Header file for red black tree

File name: sa_script.sh
Author: Keerthi Palanivel
Details: Script to perform simulated annealing

Usage: ./csim -t original_trace synthetic_trace threshold_value

Threshold value here is currently the Eucledian distance between stack distance distributions of real and synthetic trace. This parameter could be easily changed in the file.
[NOTE: csim in cache module, and SA module are different – since I initially started modifying csim code, so I left the name as it is; to avoid confusion, the name could be changed]

File name: sglib.h
Author: Marian Vittek, Bratislava, http://www.xref-tech.com/sglib
Details: Utility script

File name: random.pl
Author: Keerthi Palanivel
Details: Perl script to perform shuffling of trace file; this file will be called by the simulated annealing script

File name: global.h
Author: Nohhyun Park
Details: Header file for trace format and other environment declarations

----