Paragone: What's next in block I/O trace modeling

Rukma Talwadker
NetApp Inc.
Email: rukma.talwadker@netapp.com

Kaladhar Voruganti
NetApp Inc.
Email: kaladhar.voruganti@netapp.com

***Abstract*—Designers of storage and file systems use I/O traces to emulate application workloads while designing new algorithms and for testing bug fixes. However, since traces are large, they are hard to store and moreover inflexible to manipulate. Thus, researchers have proposed techniques to create trace models in order to alleviate these concerns. However, the prior trace modeling approaches are limited with respect to 1) number of trace parameters they can model, and hence, the accuracy of the model and 2) with respect to manipulating the trace model in both temporal and spatial domains (that is, changing the burstiness of a workload, or scaling the size of the data supporting the workload). In this paper we present a new algorithm/tool called Paragone that addresses the above mentioned problems by fundamentally re-thinking how traces should be modeled and replayed.**

I. Introduction

Leveraging traces to emulate application workloads for testing out new algorithms or bug fixes has become a common practice amongst system architects. However, traces are hard to use because 1) they consume a lot of resources when representing workloads over an extended period of time 2) they are hard to change if one wants to emulate new workload behavior and 3) it is difficult for someone to get a macroscopic view of a workload with respect to its read/write ratio, random/sequential ratio etc.

A. Related Work

In order to overcome the above mentioned deficiencies associated with traces, researchers have proposed creating models from them [2], [4], [5], [8], [10], [11]. These models are described using workload parameters such as read/write ratios, random/sequential ratios etc. Quite often, these parameters are passed as input to workload generators like Iozone, FileIO etc. A sizeable amount of previous work also builds simple models with one or more of the commonly known trace parameters [2], [8], [11]. After examining and working with many traces we have obtained key insights that some of the primary assumptions made by prior art in modeling traces need to be re-visited:

1) Use of Average Statistics: Prior trace modeling efforts [2], [8], [11] indicated that a trace model that preserves the average statistical values of various workload parameters, is sufficient for studying system behavior for varying concurrency levels or system loads. However, in order to drive system studies in data prefetching, cache admission/eviction and workload migration [6], [7] there is a need to preserve workload properties like burstiness, sequentiality etc. This, in turn, makes it necessary to preserve the specific ordering in I/O sequences than mere statistics.

978-1-4799-0218-7/13/$31.00 © 2013 IEEE

2) Self-Similarity of Traces: Prior work in trace modeling assumes that workloads exhibit similar behavior in different regions of a storage LUN/Volume [4], [5]. Our experience has shown that this assumption varies across applications and especially does not apply when many virtual disks share an underlying storage LUN. One cannot simply state that the workload characteristics observed on a 100GB LUN can be reproduced on a 10GB LUN, by chopping-off the I/Os to region beyond 10GB.

3) Manual Detection of Trace Regions: A workload's behavior varies over a period of time (regions). Prior art [2], [8], [11] expects users to manually partition a trace into regions in order to evaluate and create region specific trace models. We found that manually coming up with a region size besides being time consuming is many a times error prone. In order to address the challenges listed above, we looked at the existing trace modeling work, and then asked fundamental questions about how to a) parse a trace b) represent a trace as a model and c) re-generate the workload using the created trace model. We felt that innovation was required in all of these three areas.

II. Techniques

Figure 3 shows the component diagram of Paragone. In this section we provide an intuitive overview for these steps with a special focus on the components where we are proposing something novel.

Auto-Regression Segmenter: Workload phase changes often manifest themselves as significant changes in the distributions of one or more spatial attributes. Paragone leverages the technique of Auto Regression (AR) for trace chunking. In the auto regression model for a time series of an I/O attribute X , a subset of the time series (X_t) is represented as polynomial function of the same time series minus the most recent data point (x_{t-1}). The time series window is progressed each time by a fixed number of I/O's (sliding window). The trace is segmented with respect to the corresponding I/O attribute if the difference (Euclidean distance) between the polynomial functions of the consecutive partially overlapping time series vary more than 20%. In other words, for an attribute X each of the coefficients of the corresponding polynomial function would represent the contribution of a particular value of X to the overall function. Any change in the distributions of X would reflect in either the degree or the coefficients of the polynomial.

Distributions lack information on the temporal order of the values taken by an attribute X. Ordering is important when it comes to synthetic regeneration of workloads for validation of prefetch algorithms and in simulating the exact disk bottlenecks. Markov chain has been known for a long

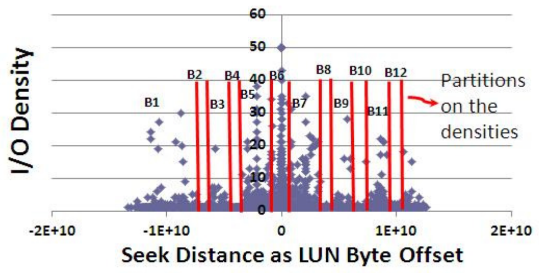


Fig. 1. Seek distance densities and dimension mapper detected boundaries

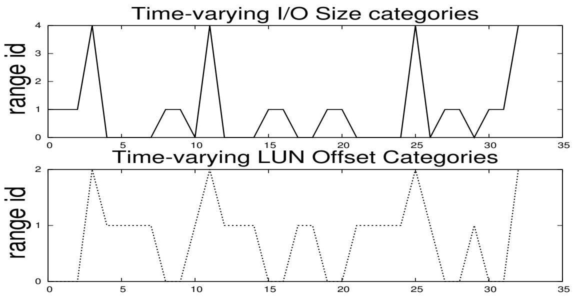


Fig. 2. Mutually dependent parameters in UMASS

time as a technique for order preservation. The size of the Markov chain however, would be quadratic in the number of attributes and the range of values per attribute. Hence the use of Markov chain would limit the number of workload attributes one can model. This limitation leads to two new challenges: 1) Summarize continuous values of attributes into discrete categories 2) Reduce the number of spatial attribute by modeling the "key" trace attributes. Former would reduce the complexity by a factor whereas the later would reduce it by a order of magnitude. These challenges create a need for the following logical blocks in Paragone.

Dimension Mapper: For attribute categorization Paragone adopts a simple method of densities for all the modeled attributes. In this method contiguous area of identical attribute value densities is considered as a bucket and assigned an id. Figure 1 shows the various buckets for seek distances formed by Paragone over its density graph. Discretization of attributes based on their density maps relaxes the scaling overheads associated with the size of the Markov chain.

Paragone Atom: We leveraged a variant of independent component analysis (ICA) that is widely used in the information/coding theory area called the *Mutual Information* [1] (MI) for the redundant dimension elimination. Figure 2 illustrates categorical values (derived from the Dimension Mapper) taken by I/O Size and LUN offsets over a small time interval in the UMASS (Table II) trace. For every single category taken by an I/O size there are one or more fixed categories taken by the LUN offsets. MI treats every attribute as being a time varying signal. Paragone overlays two signals at a time to find the degree of coherence in their patterns. This degree of coherence between the two signals is termed as mutual information. Paragone leverages the Hierarchical Agglomerative Clustering (HAC) to preserve inter-attribute coupling within the non-redundant dimensions. Post clustering, each of the I/O attribute values are mapped to one of the distributions like *Gaussian, Normal, Gamma, Uniform, Linear* and *Constant*. If the underlying distributions do not fit any of these distributions, Paragone models them using piece-wise similar empirical distributions (EPdf). In Section IV we highlight the importance of these methods within the Paragone Atom experimentally.

Trace Regenerator: It reads what-if configurations from a Paragone config file, manipulates the Paragone trace model in-memory and regenerates a new workload trace. At present Paragone supports the following what-if capabilities:

1) Multi-threading: Since the model is read-only, several non-blocking calls can be made independently to the same Markov state simultaneously by spawning multiple threads

2) Time scaling: This implies running the entire trace x times faster/slower by issuing the requests in the trace according to the scaled timestamps

3) Space scaling: This will be discussed in Section IV. Replay is done by a separate tool, which is not a part of Paragone.

In Paragone, an average trace I/O regeneration time is in the order of microseconds. Once generated, Paragone can send the I/O to the underlying block device using its trace replay interface, which is yet another user space application in the same VM.

III. Paragone Atom Implementation Details

In this section we discuss the specific implementation details of Paragone Atom which is the prime model building component of Paragone.

Paragone Atom: Atom leverages the Mutual Information (MI), Markov models, Hierarchical Agglomerative Clustering (HAC) and Empirical PDFs (EPdf).

Mutual Information: Atom constructs a multi-dimensional matrix with each of the spatial attributes as dimensions and their categories as the respective dimension ranges. A cell in the matrix corresponds to an I/O with spatial attribute values represented by its position in the matrix. If an I/O A precedes I/O B in the workload trace, we create a directed link from cell A to cell B. Probability of a directed edge from A to B is computed as the ratio of its edge weight and the sum total of weights of edges outgoing from A. When an I/O reaches cell B the cell count of B is incremented by 1.

Mutual Information (MI) between two spatial attributes X and Y is defined as:

$$MI(X;Y) = \sum_{y \in Y} \sum_{x \in X} p(x,y) log \frac{p(x,y)}{p(x)p(y)} \quad (1)$$

where $p(X,Y)$ is the joint probability distribution function of X and Y, and $p(X)$ and $p(Y)$ are the marginal probability distribution functions of X and Y respectively. p(X=x,Y=y) is computed as the number of times attribute X takes bucket id x, attribute Y takes bucket id y, divided by total number of I/O's in the trace. Paragone Atom computes MI between every pair of attributes. Transitive pairs (two or more attributes) with equal MI values are grouped together to form an *MI group*. Hence an MI group can have two or more attributes. In all of

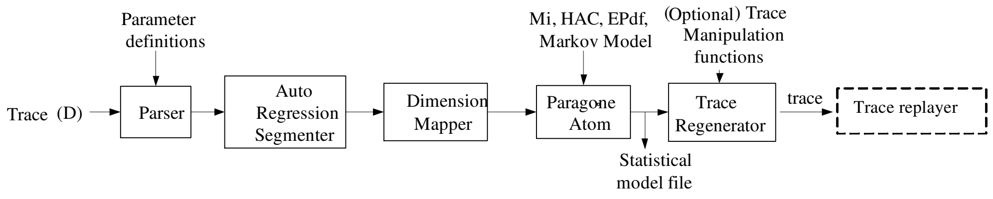


Fig. 3. Paragone component Diagram

the traces analyzed by us (Table II) we found that either MI groups show very high or extremely low (below 1) MI values. For those groups with high MI, Paragone Atom chooses a representative attribute and drops the rest. In the case of groups with negligible MI values, Atom includes every attribute that has not appeared in groups with high MI value.

Markov Models: The multi-dimensional matrix constructed by the Dimension Mapper is the representation of the Markov chain in Paragone. Post MI, Paragone collapses the cell counts for the deleted cells into the corresponding non-deleted cells. Every transition either between the two deleted cells or between a deleted and a non-deleted cell is re-mapped to a self-transition to the corresponding non-deleted cell.

HAC and EPDF We use the HAC's single link algorithm. The algorithm terminates when optimum number of clusters have been obtained using the Davies Bouldin index. Post clustering, Paragone fits data corresponding to each attribute within a cluster to the best known distribution function. After mapping to the best known function, Paragone reproduces some I/Os within the same cluster and compares them with actual data points offline. An error within 10% in terms of the spread of the data items is considered fine. In case the error is beyond 10%, Paragone uses a built in library of *Matlab* called the "empirical Pdfs" to segment attribute values within a cluster and fit piecewise similar functions.

IV. Evaluation

In this section we present four sets of experiment results that 1) evaluate the benefits of various Paragone optimizations 2) validate how Paragone preserves trace sequentiality 3) validate how Paragone preserves I/O burstiness and 4) assess Paragone's accuracy in space scaling. We ran both the original traces and the Paragone model regenerated traces via a Linux based VM client with 2GB memory, 100MBps network and storage server, with SAN support, 6GB memory, 1 TB volume with 7500 rpm disks and 1GBps network. We took care to ensure that the storage server state was reset in order to prevent biasing of the results due to cache warm up and on-disk data layout differences. We replayed original as well as regenerated traces using a trace replayer written by our group.

We define a *penalty figure* for each metric in (Table I) as the root mean square of the horizontal distance between the CDFs for original and regenerated workloads. We avoided the use of metric averages as a similarity measure because they tend to summarize interesting time varying trends into values which can give a fake illusion of accuracy.

Experiment 1: Baseline

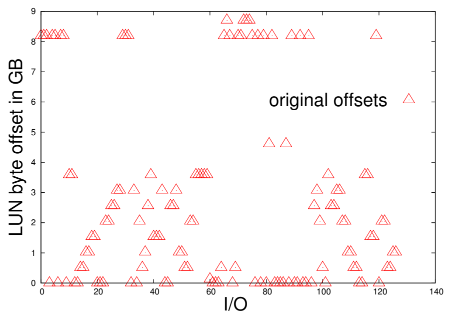
We ran three different variants (called V1, V2 and V3) of Paragone on five different traces from Table II. V1 with MI+HAC+EPdf, V2 with only MI+Gaussian distribution, and V3 with MI+HAC+Gaussian distribution. Our goal is to prove that the various algorithmic steps (Section III) taken in the Paragone atom are indeed very essential. Table I lists the penalty figures for the various metrics for the three variants. V1 is the algorithm presently implemented in the Paragone Atom and reports penalty figures only as high as 5% in the average case. This implies over 95% accuracy. As in table II, V1 also compressed the trace by minimum of 85% and in most cases even higher. V2 and V3 are the non-optimized

TABLE I. Metrics and Penalty figures (%) for Baseline

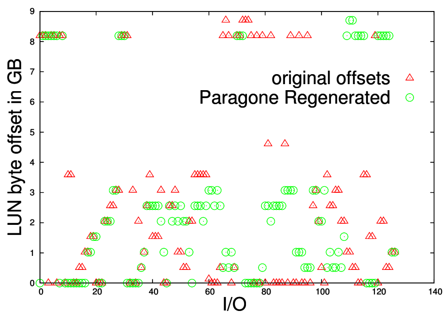
Metric	Algo.	**Penalty figures as %**				
		MSR-P	MSR-W	UMASS	OG2	AN2
Per disk write IO size **(A)**	V1	3	6	3	5	4
	V2	14	12	14	18	10
	V3	11	10	8	8	5
Per disk read IO size **(B)**	V1	3	5	4	4	3
	V2	12	16	13	20	11
	V3	9	12	10	11	6
Read Size **(C)**	V1	4	5	3	3	3
	V2	14	10	12	15	10
	V3	8	10	12	12	8
Write Size**(D)**	V1	4	8	4	3	4
	V2	14	12	9	14	9
	V3	8	12	11	10	6
Read Latency **(E)**	V1	5	8	2	4	8
	V2	23	21	24	12	17
	V3	14	17	10	8	11
Write Latency **(F)**	V1	6	7	3	4	6
	V2	14	26	16	11	10
	V3	11	24	13	9	10
Read ahead total **(G)**	V1	7	6	3	6	11
	V2	23	21	24	12	17
	V3	14	17	10	8	11
Read ahead Sequential **(H)**	V1	5	9	3	6	11
	V2	21	12	10	13	12
	V3	18	10	9.7	10	11
Read ahead Random **(I)**	V1	7	5	9	5	9
	V2	17	22	11	12	11
	V3	17	22	10.3	11	9
Cache working set size **(J)**	V1	2	6	5	4	8
	V2	14	15	14	22	15
	V3	12	14	10.9	16	12
Cache hit ratio **(K)**	V1	5	8	5	6	7
	V2	13	14	10	16	12
	V3	14	13	9	10	8
read Ops **(L)**	V1	4	5	3	2	5
	V2	10	10	11	15	9
	V3	10	9	9	10	9
Write Ops **(M)**	V1	5	4	4	3	5.5
	V2	11	8	13	13	10
	V3	9	8	9	8	8
CPU Utilization **(N)**	V1	7	8	5	4	4
	V2	15	11	12	10	8
	V3	11	10	10	7	6

TABLE II. Description of Traces

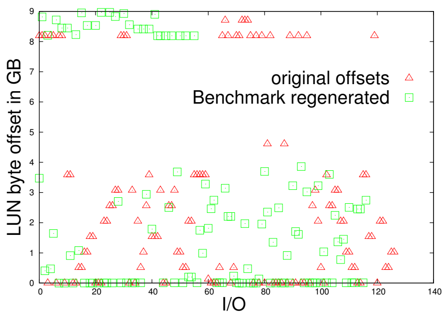
Label	Workload Type	Trace dur. (weeks)	Trace size MB	Trace source	Paragone % space saving
MSR-P	proj dir	2	1600	[3]	94
MSR-W	Webserver	2	276	[3]	96
UMASS	Financial1	0.5	155	[9]	90
OG1	Gas & Oil	20Hrs	137	customer	85
OG2	Gas & Oil	1	256	customer	95
AN1	Animation	0.2	201	customer	90
AN2	Animation	1	197	customer	85
HT1	Terasort	0.1	387	customer	90
AS1	log analytics	1.3	256	customer	96



(a) Original Terasort (HT1) trace



(b) Original vs Paragone Regenerated



(c) Original vs modified micro benchmark

Fig. 4. Sequentiality preservation for Hadoop Terasort (HT1) trace

versions of Paragone Atom which lack one or more techniques described in Section III. V2 shows penalty figures as high as 25%. V3 performs better than V2 but as we see V1 gives the best results.

Comparative approach implementation We have attempted to compare Paragone re-generated workloads in the subsequent experiments with those generated by benchmarks using a technique very similar to that published in this field [10]. The algorithm described in [10] chunks the trace every 10 seconds into segments. Post chunking, the paper adopts an incremental method of merging similar segments based on a threshold to reduce workload emulation overheads. Since we are not comparing the emulation overheads here, we operated at 10 seconds segments. Post segmentation the model extracts parameters like: I/O sizes, Read/Write%, Random/Sequential%. In the paper [10] there is no mention of whether the values of model parameters are fed as *Averages* or emulated using empirical distributions within the load generators like File Bench. In the absence of an ability in these benchmarks (Iozone/File Bench) to the best of our knowledge to emulate distributions we resort to using averages. We used our own I/Ozone like internal benchmark and patched it with functionalities at par to the latest I/Ozone version offerings so that the inputs and the behavior are comparable.

Experiment 2: Sequentiality Modeling Sequentiality refers to accessing consecutive block addresses in a LUN. Typically sequential patterns are interleaved by other sequential or random accesses. This is quite true in the case of multiple VM's mapped to the same storage LUN (multi-tenancy) or several simultaneous client streams accessing the LUN (multi-threading). Figure 4(a) shows LUN access patterns in a Terasort application running on a Hadoop cluster (HT1) accessing our storage. Access patterns within different LUN regions are self-similar but quite different across locations and are interleaved over time. For regeneration, one has to satisfy the following key *properties* 1) group regions of self-similar access patterns 2) model the length of sequential access as a distribution 3) model request arrival rates within and across self-similar regions.

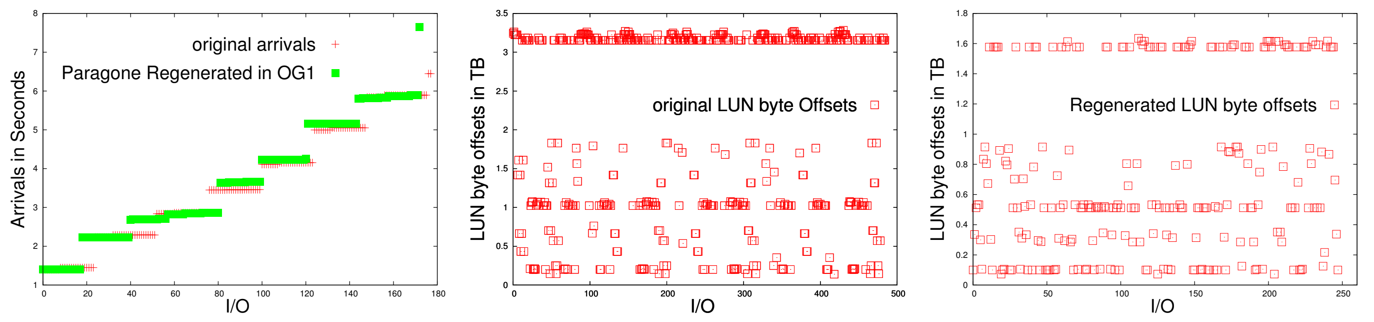
Firstly, we regenerated the HT1 workload using our internal benchmark with technique described above as shown in the Figure 4(c). Property 1 was simulated by manually detecting and creating multiple files each corresponding to a region of self-similar access patterns. Each file was mapped to a new benchmark instance. Neither Iozone/File Bench nor our internal benchmark accepts seek distance as an input. As a result the LUN offset location for each random I/O in the LUN region was randomly chosen by the benchmark. We extracted the penalty figures for the original, Paragone regenerated and regenerated using the comparative approach and observed that penalty figures for the benchmark were as high as 25% when compared to the original.

In Paragone, seek distances as well as the request inter-arrival times are parameterized. Interleaving of accesses at different LUN offsets are handled by means of transitions between Markov states. In Figure 4(b) (Paragone model re-generated trace) we see that the spatial as well the temporal aspects of the workload are well preserved with the penalty figures for the specific metrics being only as high as 8%.

Experiment 3: I/O Burst Modeling In multi-tenant deployments, taking care of the workload bursts is a major challenge. Workload OG1 (Table II) is from a customer dealing with oil and gas.

Figures 6 and 5(a) show a staircase pattern of request arrivals in OG1. Each step represents a burst. Vertical distance between two consecutive steps represents a no-activity or idle interval. In order to faithfully reproduce the burst pattern we need to satisfy the following *properties*: 1) model the step lengths as a distribution 2) model the length of the idle or low activity periods as a distribution.

Benchmarks do not model request inter-arrival times and mostly follow a closed loop arrival model. We modified our internal benchmark to expose Gaussian as well as Poisson pattern of arrivals as a function argument. Figure 6, shows requests generated by the benchmark, using the comparative approach discussed above. There are no distinct burst and idle periods in Figure 6, as two periods interleave too often in a 10 seconds time window in the original workload. We also found it impractical to switch benchmark instances with different request arrival distributions in less than 10 seconds. This fetched the benchmark a minimum regeneration penalty figure of 15%. Paragone models the entire trace interval as one model because, though auto-regression detects that inter-arrival times (during the bursty and idle periods) follow different distributions, the distributions themselves do not change. A transition to and from idle period (OG1) to a bursty period is modeled as a separate distribution by the Paragone Atom which takes care of the above mentioned properties of burst modeling. Figure 5(a) overlay the Paragone reproduced request



(a) Original vs Paragone (OG1)
(b) Original access pattern in MSR-W
(c) Down Scaling a LUN to half its size

Fig. 5. Paragone based workload modeling for OG1 and MSR-W workloads

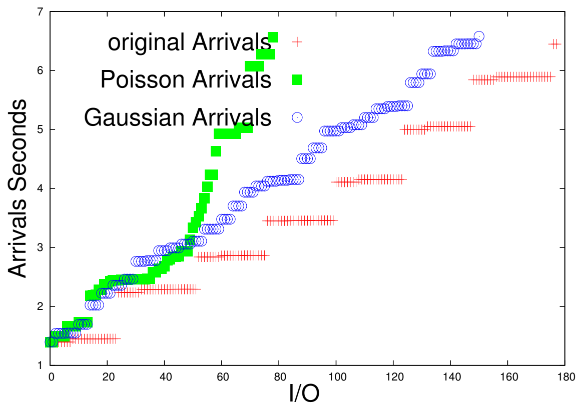


Fig. 6. Original vs micro benchmark (OG1)

arrival patterns over the original trace. The maximum penalty figures for the Paragone regenerated trace was as low as 5%.

Experiment 4: Space Scaling Given a trace, Paragone can scale it up or down spatially or temporally and addresses some of the existing limitations of scaling in the state of the art [12]. During trace scale up, we expect the same region of the LUN to expand so that the disk/network bandwidth or file access related bottlenecks start showing up. Due to the space limitation we only show results of workload scaled down for a trace segment of MSR-W workload in Figure 5(b).

Scaling down a trace would mean reducing the total number of files stored on the LUN by x% with corresponding reduction in the I/O load. In this experiment we use a scaling factor of 0.5. The LUN offset categories of the workload derived by the Dimension Mapper for the original trace in Figure 5(b) are re-mapped to newer scaled down offsets by the regenerator. During this regeneration, Paragone reduces the intensity of I/O's by the scaling factor analogous to the reduction in the number of workload clients. Since the seek distance and request I/O size metrics are preserved, if any I/O request generated by the regenerator falls beyond the resized boundary for any LUN location, it is ignored (foul I/O). Figure 5(c) shows the corresponding scaled trace. It was observed that Paragone generated almost half the number of I/O in the same interval in this case.

In reality, we can only assume that if the LUN size decreases/increases by a factor of x, the number of files and hence the request intensity will approximately scale by factor of x. Paragone is unique and first of its kind in regenerating a scaled up/down trace to benefit storage engineering activities.

V. Conclusion

In this paper we presented an algorithm/tool called Paragone that advances the state of the art in block I/O trace modeling by solving the following key challenges: 1) How to preserve the sequential patterns present in a trace in the model representing the trace 2) How to preserve I/O bursts and 3) How to manipulate the trace model to scale between the size of datasets (LUNs) in the actual trace versus the different sizes the users want to emulate via the model. In this paper we have combined a number of existing statistics based data analysis techniques in a novel manner to create Paragone. We tested our algorithm using many traces and our results show that our techniques in general lead to less than 5% error in the accuracy of the trace model. In future, we plan to extend our work to file I/O traces.

References

[1] Thomas M. Cover and Joy A. Thomas. Elements of information theory. http://http://paulbourke.net/miscellaneous/ar/, 1991.

[2] C. Delimitrou, S. Sankar, B. Khessib, K. Vaid, and C. Kozyrakis. Time and cost-efficient modeling and generation of large-scale tpcc/tpce/tpch workloads. In *Proceedings of the Third TPC Technology conference on Topics in Performance Evaluation, Measurement and Characterization*.

[3] N. Dushyanth, D. Austin, and R. Antony. Write off-loading: Practical power management for enterprise storage. *Trans. Storage*.

[4] G. Ganger. Generating representative synthetic workloads: An unsolved problem. In *Proceedings of the CMG Conference, 1995*.

[5] M. E. Gomez and V. Santonja. Self-similarity in i/o workload: Analysis and modeling. In *Proceedings of the Workload Characterization: Methodology and Case Studies*.

[6] B. Hong and T. Madhyastha. The relevance of long-range dependence in disk traffic and implications for trace synthesis. In *MSST 05*.

[7] T. Kimbrel, A. Tomkins, R. Hugo Patterson, B. Bershad, and Pei Cao. A trace-driven comparison of algorithms for parallel prefetching and caching. In *(OSDI 1996)*, 1996.

[8] Z. Kurmas, K. Keeton, and K. Mackenzie. Synthesizing representative i/o workloads using iterative distillation. In *(MASCOTS 2003)*.

[9] SNIA. Iotta trace repository. http://iotta.snia.org/tracetypes/3/.

[10] V. Tarasov, K. S. Kumar, J. Ma, D. Hildebrand, A. Povzner, G. Kuenning, and E. Zadok. Extracting flexible, replayable models from large block traces. In *(FAST '12)*.

[11] J. Zhang, A. Sivasubramaniam, H. Franke, G. Natarajan, and S. Nagar. Synthesizing representative i/o workloads for tpc-h. In *(HPCA '04)*.

[12] N. Zhu, J. Chen, and T. Chiueh. Tbbt: Scalable and accurate trace replay for file server evaluation. In *(FAST 2005)*.