Realistic Request Arrival Generation In Storage Benchmarks

Rekha Pitchumani
UC Santa Cruz
Email: rekhap@soe.ucsc.edu

Shayna Frank
UC Santa Cruz
Email: smfrank@soe.ucsc.edu

Ethan L Miller
UC Santa Cruz
Email: elm@soe.ucsc.edu

***Abstract*—Benchmarks are widely used to perform apples-to-apples comparison in a controlled and reliable fashion. Benchmarks must model real world workload behavior. In recent years, to meet web scale demands, Key-Value (KV) stores have emerged as a vital component of cloud serving systems. The Yahoo! Cloud Serving Benchmark (YCSB) has emerged as the standard benchmark for evaluating key-value systems, and has been preferred by both the industry and academia. Though YCSB provides a variety of options to generate realistic workloads, like most benchmarks it has ignored the temporal characteristics of generated workloads. YCSB's constant-rate request arrival process is unrealistic and fails to capture the real world arrival patterns.**

Existing workload studies on disk, filesystem, key-value system, network, and web traffic all show that they all exhibit some common temporal properties such as burstiness, self similarity, long range dependence, and diurnal activity. In this work, we show that the commonly observed traffic patterns can be modeled using the three categories of arrival processes: a)*Poisson*, b)*Self similar*, and c)*Envelope-guided* process. The three categories presented are a necessary and sufficient set of request arrival models that all storage benchmarks should provide. To demonstrate the ease of incorporating the models in benchmarks, we have modified YCSB to generate workloads based on all three models, and show the effect of realistic request arrivals through an example database evaluation.

I. Introduction

Many systems designed to solve real-world problems prove the worth of their solution through an evaluation framework that replays real-world workload traces. Though a good approach, such traces are not abundant and the ones that are available may not always be applicable. In such situations, benchmarks are used in the evaluation of different designs with the same goals, not just in academic research, but also in real world product promotions. Hence, standard benchmarks have to be representative of real world needs, modeled based on observed real-world workloads. Even though the workload's temporal characteristics are a big influencer on the system behavior, it has been mostly ignored by benchmarks.

Key-value stores have become a vital component of cloud computing applications and high performance web scale databases. The key-value interface, being a simple and versatile interface, is applicable to both the large distributed stores and the individual storage nodes that make up the large distributed stores. Being a device agnostic interface, it is suitable for all the different kinds of devices in the storage hierarchy, and as such serves as a unified model for all the layers in the hierarchy. Building distributed key-value stores that meet web scale demands and scale as demand rises is an active research area.

Though many works exist on building better key-value systems, a trace based evaluation is out of reach, as, to the best of our knowledge, no key-value workload trace is publicly available. The only published key-value workload study is that of an in-memory key-value caching layer, Facebook's Memcached deployment [1], and even if obtained will be unsuitable to evaluate individual key-value storage nodes, such as the Kinetic [2] disks, the ethernet key-value disks from Seagate.

In recent years, the Yahoo! Cloud Serving Benchmark (YCSB) [3] has emerged as the standard benchmark of choice for evaluating key-value systems. YCSB has been used both in the evaluation of large distributed key value stores [4], [5], and individual key-value storage nodes [6]. YCSB has also been used to generate representative data serving scale-out workload in the evaluation of modern processor limitations [7]. YCSB comes with a workload generator that is flexible in the selection of different mixes of operations and data sizes, and key selections based on different distributions. But its framework does not come with the flexibility to generate realistic request arrivals.

YCSB's constant-rate request arrival process is unrealistic and fails to capture real world arrival patterns. Even then, YCSB has been used in the evaluation of an elasticity controller for cloud-based elastic key-value stores designed to automatically respond to changes in workload [8], by simply adding and removing YCSB clients that generate request at a constant rate. Other systems that use YCSB for evaluation in a similar manner include a system for achieving datacenter-wide per-tenant performance isolation and fairness [9], and systems performing live database migration to tolerate load variations in multi-tenant databases [10], [11].

Storage devices have background tasks, such as scrubbing, cache de-staging, data migration across tiers and automatic backups, that need to co-exist with foreground request processing. Modern storage media, such as NAND Flash and Shingled Magnetic Recording disks, require log-structured data management approaches to overcome the media's inability to update data in-place. Such devices also come with background compaction (garbage collection and reorganization) processes that oftentimes interfere with incoming media access requests. Different designs handle the additional overhead in different ways, and are in need of an evaluation framework that does not bombard the system with requests continuously, but rather generates realistic arrivals with periods of both low and high activity.

Workload studies that analyze traces collected from real systems aid in realistic synthetic workload generation. The only published key-value workload study, the one on Facebook's in-memory key-value caching layer [1], notes that the observed workloads are bursty, and diurnal with traffic spikes. The observed temporal patterns do not come as a surprise, as existing studies show that disk, filesystem, network, and web traffic all exhibit some common temporal properties such as burstiness, self similarity, long range dependence, and diurnal activity.

We classify typically observed temporal patterns into three kinds of arrival processes: a)*Poisson*, b)*Self similar*, and c)*Envelope-guided* processes. We have implemented inter-arrival time generation based on all three temporal models, and extended YCSB's framework to enable a workload executor that sends requests based on them. We evaluate the generated requests and show that the statistical properties of the generated requests conform to that of the arrival process selected. Further, to demonstrate the usefulness of realistic arrivals, we use the modified code to evaluate a modern embeddable key-value store, LevelDB [12], and show the variations in observed latencies, both as a histogram and on a timeline. The high latencies could be attributed to periodic background activity in LevelDB that strives to garbage collect and reorder the key-value pairs stored in the system, and could be brought down via scheduling the activities during downtime. Our work could be used to generate realistic downtime to evaluate such schemes.

II. Related Work

Some simulation systems extract request inter-arrival time distributions from real-world systems or traces, and mimic the arrival pattern in the simulated traffic by sampling the inter arrivals from the configurable distribution parameter [13], [14], [15]. In TPC-W, a widely used traditional client-server benchmark, user arrivals are defined by a Poisson process. To rectify its lack of ability to produce burstiness, Mi *et al.* [16] injected burstiness into it, using a Markov-modulated process, based on the popular ON/OFF traffic models used in networking to create correlated inter-arrival times.

One of the methods we use to generate realistic arrivals, the b-model, a simple model to generate self similar, bursty traffic for a wide range of time scales, was first used by Wang *et al.* to generate self similar disk IO traces [17]. Hong and Madhyastha argued that there was no need to model self similarity at large time scales in disk traffic, as it is irrelevant for measuring disk response times and queuing behavior, and used the b-model to generate synthetic arrivals at short time scales [18]. But as we discussed earlier, realistic arrivals can be useful in evaluating many system functionalities that span across all storage media at different time scales, and it is important that benchmarks recognize the importance of the temporal characteristics of generated workloads.

YCSB++ [19] extended YCSB with a set of additional features that can be used in database advanced functionality performance testing and debugging, but does not address the lack of arrival variability. Features provided by YCSB++ and our work could be complementary to each other.

III. Request Inter-Arrival Times Process Models

Request arrival process is a stochastic process, and most of the time it is strongly correlated to itself. Autocorrelation is the cross-correlation of a time series with itself, and is a measure of whether a workload is correlated to itself or not. The autocorrelation function (ACF) can be used to measure the similarity between the original arrival time series and the same time series shifted by some time delay, as a function of the time lag between them [20]. In other words, the ACF shows whether the request interarrivals at any point of time is dependent on it's previous values, or is independent. The ACF of a stochastic process $X = (X_1, X_2, X_3, ..)$ with mean μ and variance σ^2 at lag k is given by

$$r(k) = \frac{\frac{1}{n-k} \sum_{i=1}^{n-k} (X_i - \mu)(X_{i+k} - \mu)}{\sigma^2}$$

The above function is normalized and would result in values between $[-1, 1]$. When lag is 0, the series is compared to itself unmodified and the ACF will have the highest value 1. Positive ACF values mean that the random variable has a high probability to be followed by another variable of the same order of magnitude, while negative ACF implies the inverse.

Real requests do not arrive at a constant rate with a fixed time interval between them. It is important for a benchmark's workload generator to offer a variety of choices, that are both realistic and configurable, to match the workload a user has in mind. In this section, we categorize the request arrival process into three categories: a)*Poisson*, b)*Self similar*, and c)*Envelope-guided*, and argue based on evidence from existing web, disk, file and network IO studies that the categories presented are both necessary and sufficient to represent the real world needs.

1) Poisson Process: A poisson process is a simple and widely used stochastic process for modeling arrival times. Requests can be modeled as a poisson process if the request inter-arrival times are truly independent and exponentially distributed. The ACF of a poisson process is usually low and close to zero even at lag 1. Unless the inter arrivals are truly uncorrelated, the poisson process is an unsuitable choice. Research shows that most arrivals are correlated, and cannot be modeled accurately by a poisson process [21]. Nevertheless, when many different kinds of independent workloads are run on a system, the resulting traffic could look like a poisson process.

Cao *et.al.* [22] studied the internet traffic and found that as the rate of new TCP connections increases, arrival processes (packet and connection) tend locally toward Poisson, and that time series variables (packet sizes, transferred file sizes, and connection round-trip times) tend locally toward independent. They concluded that the cause of the nonstationarity is superposition: the intermingling of sequences of connections between different source-destination pairs, and the intermingling of sequences of packets from different connections. Similar behavior can be expected in web scale cloud scale key-value workloads too. Hence, we have chosen the poisson process as out first category.

2) Self Similar Process: Self similarity means the series looks similar to itself at different time scales. Self similar workloads typically include bursts of increased activity, and similar looking bursts appear at many different time scales. A poisson process too looks bursty at smaller time scales, as other processes following a long-tailed distribution do. But when aggregated and viewed at higher time scales, gets smoothened, whereas aggregating streams of self similar traffic typically intensifies the self similarity instead of smoothing it. Long range dependence means the series is correlated to not just its immediate past, but also its distant past. So, the ACF of a long range dependent process decays slowly. Self similarity and long range dependence, though separate phenomenons, are typically observed together.

Many real life observed workloads are both self similar and long range dependent. Self similarity has been observed in WWW traffic [23], Ethernet local area network (LAN) traffic [24], file-system traffic [25] and also in disk-level I/O traffic [26], [27]. Further, researchers investigated a number of wide-area TCP arrival processes [21], and concluded that even if the finite arrival process derived from a particular packet trace does not appear self similar, if it exhibits large-scale correlations suggestive of long-range dependence then that process is almost certainly better approximated using a self similar process than using a Poisson process. Hence, we believe benchmarks should also provide the facility to model request arrivals based on a self-similar process.

Hurst parameter, H, is the exponent that describes the cumulative expected deviation from the mean after n steps in a random walk [20]. Higher values of H are the result of stronger long-range dependence. The Hurst parameter is often used to quantitatively measure the self similarity of a time series. H is equal to 0.5 for a Poisson process, and is in the range 0.5-1 for a self similar process. A variety of methods exists to estimate the value of H of a time series, and for a thorough description of the most popular methods we recommend referring to Feitelson's book on workload modeling [20]. We use Selfis [28] to compute the Hurst parameter of the generated inter-arrivals using five different estimation methods.

Feitelson also describes in detail a variety of methods to model self-similarity [20]. In our work, we generate self-similar traffic using the *b-model*, a simple model to generate self similar, bursty traffic for a wide range of time scales [17]. The model requires a single characteristic parameter, bias b. The idea is to split the entire amount of work recursively into two portions in a proportion determined by the bias b, similar to Figure 1. Thus, the total number of operations N is divided into bN and $(1-b)N$, and whether the first half of the divided time-period receives bN operations or $(1-b)N$ operations is determined randomly. Such recursive work division generates self similar traffic with high local irregularity, where b closer to 1 generates traffic with high irregularity and $b = 0.5$ results in uniform traffic.

3) Envelope-Guided Process: Gribble *et al.* showed that high-level file system events exhibit self similar behavior, but only for short-term time scales of approximately under a day [25]. By examining long-term file system trace data, they showed that high variability and self similar behavior does not persist across time scales of days, weeks and months, and concluded that the file system traffic is well represented by a self similar process for short time scales, but is unsuitable for long time scales.

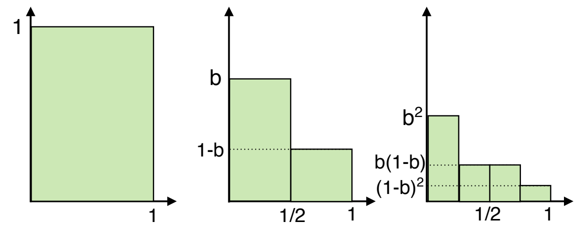


Fig. 1. Multiplicative cascading generation of b-model.

At longer time scales, many workloads exhibit a clear diurnal pattern [25], [1]. Karagiannis *et al.* show that periodicity can obscure the analysis of a signal giving partial evidence of long-range dependence [29]. Also, Akgul *et al.* showed that periodicity-based anomalies affect Hurst parameter estimation, causing unreliable H estimates, and if periodic anomalies exist they should be removed before estimation [30]. The presence of periodicity could have led to the conclusion by Gribble *et al.* that a self similar process is unsuitable for long time scales.

If the traffic is periodic and exhibits a pattern such as a daily/weekly activity cycle, then the ACF plot of the arrival process does not decay slowly as it does for a self similar process. Instead, the ACF oscillates between positive and negative values, corresponding to the periodicity of the original time series. The autocorrelation function can clearly extract and demonstrate periodicity even from much noisier data. As diurnal cycle is common in storage workloads, it is vital that benchmarks come with the option to generate such traffic.

The observed self similarity at smaller time scales can also be attributed to traffic conforming to heavy tailed distributions such as the Pareto distribution. Heavy tailed distributions can also result in larger H values similar to the long range dependent process. As summing heavy-tailed random variables does not average out, but rather leads to a heavy-tailed sum, when a process composed of heavy-tailed samples is aggregated, we will get a process with similar statistics. Paxson *et al.* showed that 'pseudo self similar' processes, arrival processes that appear to some extent self similar, could be produced by constructing arrivals using Pareto interarrivals, and that the generated traffic has large-scale correlations and the visual self similarity property, though the traffic generated is not actually long-range dependent (and thus not self similar) [21].

Roughan *et al.* noted that Internet backbone traffic has both daily and weekly periodic components, as well as a longer-term trend, and superimposed on top of these components are shorter time scale stochastic variations [31]. They modeled such traffic by segmenting the traffic into a regular, predictable component, and a stochastic component. Our final category, the envelope-guided process is similar to their approach. A predictable component such as a sine wave function determines the arrival rate per time interval, and the actual inter arrivals are generated from a secondary distribution such as the Pareto distribution. While the overall arrival rate is determined by an arrival rate function, the burstiness of the arrivals is determined by the secondary distribution selected.

IV. Design and Implementation

The Yahoo! Cloud Serving Benchmark is a client program, written in Java, that is designed to generate requests conforming to user-specified workload. YCSB client architecture, as shown in Figure 2, has a thread-safe workload generator that generates requests according to user specifications in a workload configuration file, a workload executor to execute the generated requests, a database interface layer to connect with and pass requests to the database, and a separate thread to periodically collect and report the status. By default, the workload executor runs a single thread, but is configurable and can be increased by the user. Each executor thread gets the request from the thread-safe workload generator and sends it to the database through its own instance of the database layer.

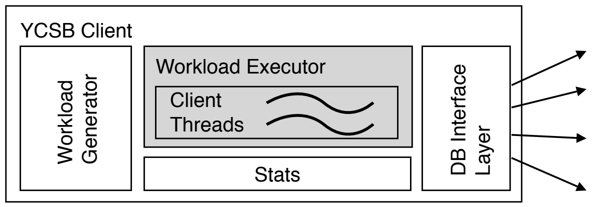
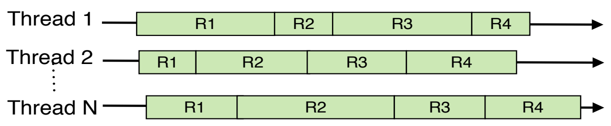
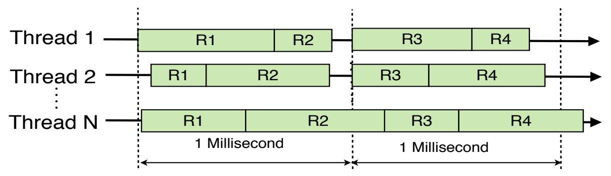


Fig. 2. YCSB client architecture. The workload executor drives multiple threads to send requests generated by the workload generator to the database.

The threads perform back-to-back synchronous IO as shown in Figure 3a. Each thread sends a request to the database, waits for the response and immediately sends the next request once the previous response is received. When configured to run with multiple threads, the total number of operations to be performed is equally divided among the threads and each thread runs its share of operations in the same synchronous fashion, in parallel. As shown in Figure 3a, the threads start execution at slightly different times to avoid hitting the database at the same time, and after that the time requests are sent depends on previous completions.



(a) By default, each thread sends requests, waits for the completion of the sent request, and immediately sends the next request.



(b) If user specifies a target load resulting in say 2 requests per millisecond per thread, requests may be delayed depending on whether 2 request per millisecond has been completed.

Fig. 3. YCSB's request execution process.

To control the load offered to the database, the threads come with the ability to throttle the rate at which requests are generated. When a target arrival rate is specified, after every request the thread monitors the number of requests generated until then and the elapsed time, and sleeps if necessary to maintain the target arrival rate. The resulting request execution process looks as in Figure 3b, in which the user specified target throughput results in 2 requests per millisecond per thread. If the target specified is low, the threads could all hit the database at the same time, but some may not due to timing inaccuracies resulting from sleep.

TABLE I. Additional Configuration Parameters

Parameters	Description
arrivalgenerator	Specifies the generator to use. Accepts *constant*, *poisson*, *self similar*, and *diurnal*.
ss.bias	Specifies bias b in the b-model, used to generate self similar traffic.
diurnal.modulation	Specifies the diurnal cycle modulation.
diurnal.cyclelength	Specifies length of the diurnal cycle in minutes.
diurnal.distribution	Specifies the distribution to use for the stochastic variation.
diurnal.distribution.shape	Specifies the distribution's shape parameter.

We have implemented three inter-arrival time generators as per the three models described in the previous section. Each YCSB client is designed to have its own inter-arrival time generator that determines the arrivals for the generated traffic. For a Poisson process, an exponential distribution is used to generate the inter-arrival times. For a self similar process, we implemented the b model and the bias b is user configurable. For the envelope-guided process, we have implemented a sine wave function, the shape of which is user configurable, which is used to determine the request arrivals per second. The actual inter-arrivals for the envelope-guided process are obtained from a secondary configurable distribution, such as the Pareto distribution. The envelope function could take a number of forms as per the need, and our work could be extended to include more patterns, as well as more secondary distribution choices.

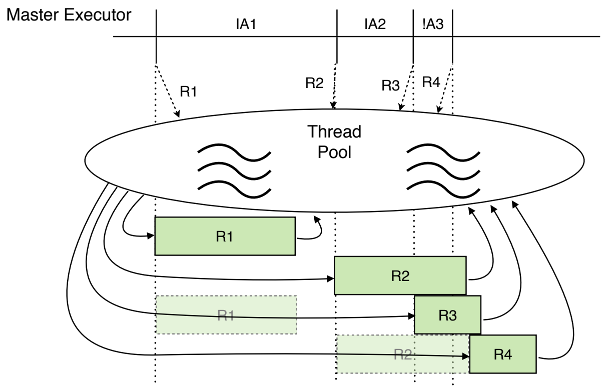


Fig. 4. Our modifications to the workload executor to facilitate realistic arrivals. If threads in the pool are unavailable either due to configuration or slow response, there could be delay as in R4.

We assume that the request sizes are independent of arrival times, as the only published key-value workload study found that the request arrivals are not correlated to the request sizes. Thus, no modifications to YCSB's workload generator was necessary. Our changes include new additions to the configuration parameters, to specify the choice of inter-arrival time generators, and shape parameters for specific generators, and modifications to the workload executor, to utilize the inter-arrival times generated by specified inter-arrival generators. The additional parameters we introduced and their descriptions are listed in Table I

We noticed that, even with nanosleep and big-resolution timers, sleep does not always wake up as instructed and gives rise to lot of timing inaccuracies. Busy wait of all available threads is also out of the question, due to the high CPU overhead. We redesigned the workload executor, as shown in Figure 4, to facilitate generating bursts of IO activity at specified time intervals. We utilize Java's thread pool functionality to have a number of threads active at any given time. Though the thread pool provides its own task queue and can pick threads once they are available to serve other tasks, we found the automatic detection of thread availability to be slower. So, we maintain our own thread queue, to which we add a thread once it is done servicing a request, and instruct the thread pool to execute the first available thread in our queue when needed. The master workload executor obtains the inter-arrival times from the inter-arrival generator, busy waits until the next request is due to be issued, and then issues the request using the first available thread.

V. Evaluation

In this section, we evaluate the accuracy and the effectiveness of our generators. After a brief description of our experimental setup, we generate traffic using a variety of configurations, and show that the generated traffic conforms to the specified arrival processes, both visually and empirically, via illustrations and statistical analysis of the generated traffic. Finally, we demonstrate the usefulness of realistic arrivals by evaluating a state-of-the-art key-value embeddable database library under all three models of request arrivals.

A. Experimental Setup

The evaluation was done on a Fedora 21 linux machine that has a quad-core 3.30 GHz Intel(R) Core(TM) i5-3550 processor with a 128 KB L1 cache, 1 MB L2 cache, and a 6 MB L3 cache, and 16 GB of RAM. For the application demonstration, the database evaluated was an embeddable database and to be able to connect and communicate with YCSB, LevelDB [12] was used with the MapKeeper [32] server. Both the YCSB client and Mapkeeper server was run on the same machine. The database was stored on a separate dedicated 160 GB single platter Seagate SATA disk drive running ext4 filesystem.

B. Realistic Arrival Visualization

For evaluating the arrival characteristics of the generated traffic, we ran the YCSB client against the YCSB's placeholder database, the *basic* database. The basic database receives all requests, does nothing, optionally injects delays, and reports a successful completion. We modified it slightly, to log the requests received with a timestamp. Throughout this subsection, all our experiments specified a target request rate of 10,000 operations per second, for better visual comparison of the different traffic generated.

Figure 5 shows YCSB's original behavior when executed with 1, 4, and 8 threads. We can see from the bottom graph that

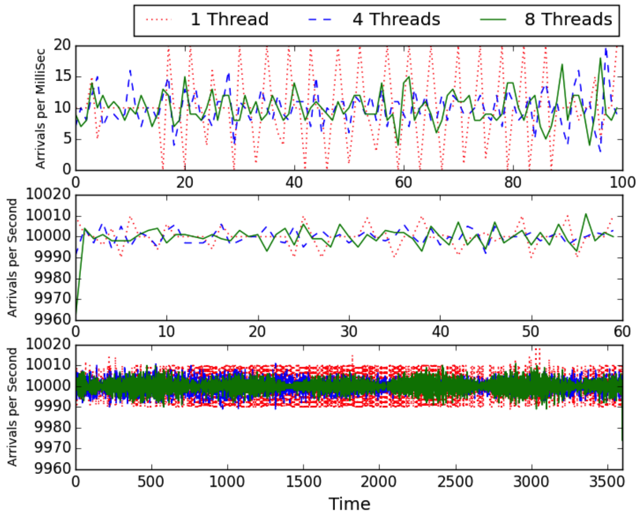


Fig. 5. Original YCSB request arrival pattern for a target rate of 10,000 operations per second.

the arrival rate mostly remains between 9990-10010 operations per second, and the minor variations are typically a result of sleep inaccuracies. The top two graphs zoom in on a small interval of time, 60 seconds and 100 milliseconds. The arrivals at the millisecond interval, the topmost graph, shows the number of arrivals oscillating, and is particularly evident in the single threaded case. This is because the basic database does nothing and returns immediately and the client thread performs all IO at a time and then sleeps. But having multiple threads smoothes them out as different threads are executing and sleeping at different times.

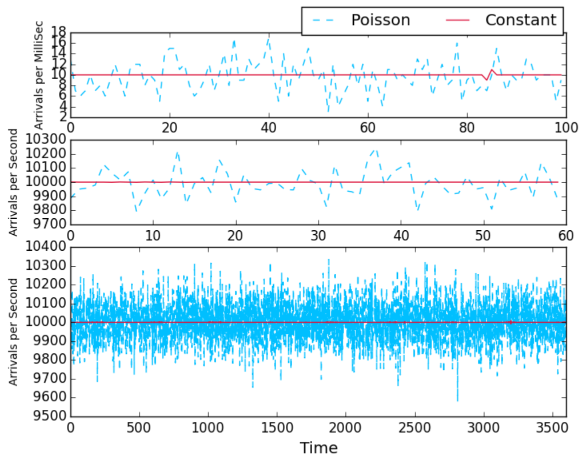


Fig. 6. Traffic generated by our modified workload executor for a target rate of 10,000 operations per second, configured with a constant and Poisson request arrival process.

Figure 6 illustrates the traffic generated when configured with a constant arrival process and a Poisson arrival process by our modified workload executor. The constant arrival process illustrates how our framework is not subject to the sleep related inaccuracies seen in the original YCSB, and is able to send requests at generated intervals precisely. As discussed earlier, the Poisson process may look bursty at smaller time scales,

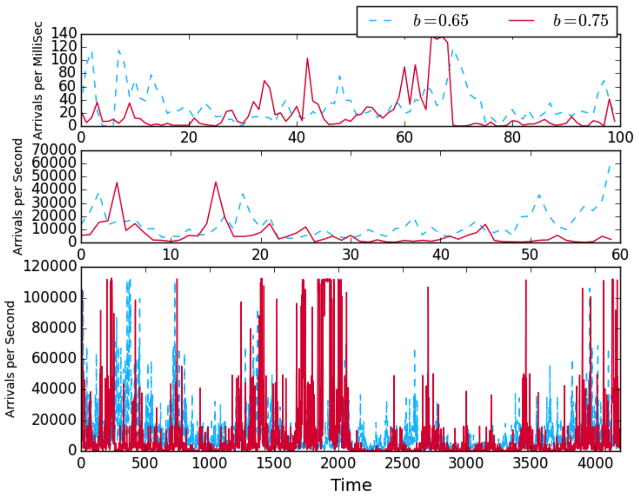


Fig. 7. Self similar traffic generated by our modified workload executor, for a target rate of 10,000 operations per second, using the b-model configured with values 0.65 and 0.75.

but when aggregated gets smoothened.

Figure 7 shows self similar arrivals generated using the b-model. Traffic bursts can be clearly seen at all time scales, minutes, seconds, and milliseconds, and aggregation does not smooth the traffic as in the Poisson process, as described earlier. As noted before, the bias b, which is configurable, determines the burstiness of the generated traffic. A bias equal to 0.75 creates more bursts than does a 0.65 bias.

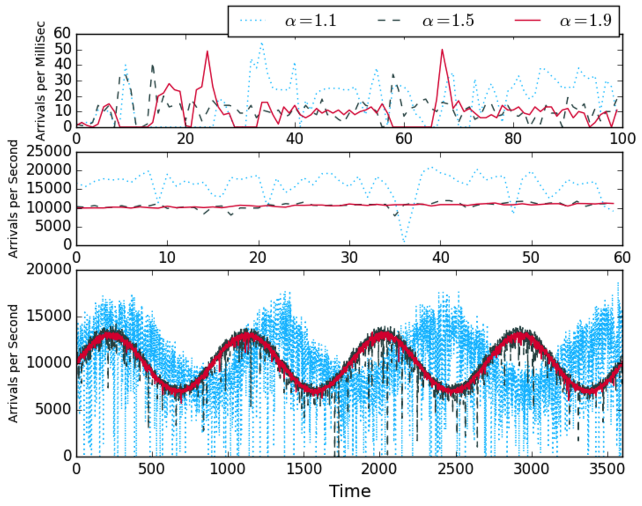


Fig. 8. Envelope-guided arrivals, configured with a diurnal envelope combined with a Pareto stochastic variations, for a target rate of 10,000 operations per second.

The generated envelope-guided arrivals can be seen Figure 8. The envelope function here is a diurnal cycle with the stochastic variations provided via Pareto inter-arrivals. α is Pareto's shape parameter and determines the variations introduced in the traffic. As seen in the figure, $\alpha = 1.9$ generates bursts at smaller timescales and smoothes out when aggregated, similar to a Poisson process. But $\alpha = 1.1$ introduces lots of variations and generates bursts at different timescales. This traffic with periodicity is not really self similar, but behaves a like self similar process.

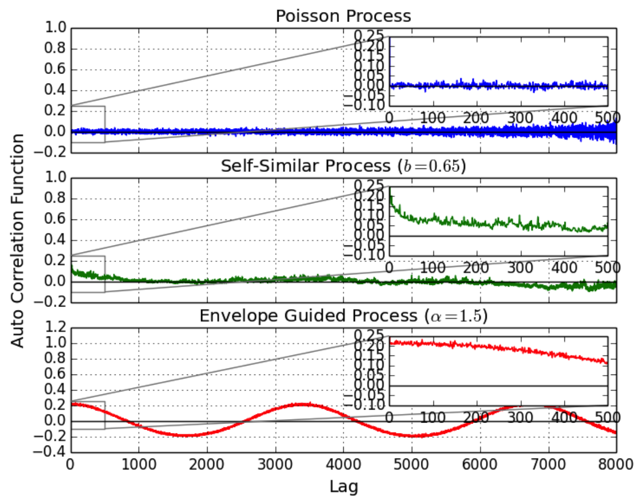


Fig. 9. Auto Correlation Function plot for a sample short workload.

C. Empirical Evaluation of Arrivals

We empirically evaluate the generated request arrival in this subsection to show that their statistic characteristics hold. We use both the Auto Correlation Function plot and Hurst parameter estimation for the evaluation. For better visualization, we present the ACF plot of the inter-arrivals generated for a short duration run, in Figure 9. It can be seen from the figure that, as described earlier, ACF of the inter-arrivals of the Poisson traffic quickly reaches near zero and remains close to zero throughout. But the ACF of the inter-arrivals of the self similar traffic with a bias 0.65 decays slowly to zero. The periodicity present in the envelope-guided diurnal traffic is also clearly visible, even with the presence of stochastic variations. The ACF plots of the arrivals generated per second for the runs shown in the previous subsection was also similar.

Table II shows the results of the Hurst parameter estimation for the arrivals visualized in the previous subsection. We present the estimations for both a short duration inter-arrival time series picked from the beginning of the entire run, and the entire run's arrivals per second time series. As mentioned earlier, we use the tool Selfis [28] to estimate H using five different methods, namely the Aggregate Variance method, R/S plot, Periodogram, the Abry-Veitch Estimator, and the Whittle Estimator, and detailed descriptions of the various methods can be found in Feitelson's book on workload modeling [20]. As seen in the table, there are variations among the values estimated by the different methods, hence the approach of using different methods for the estimation.

The general practice is to declare a process as self similar if most methods result in a Hurst estimation of above 0.5, and if most estimate it close to 0.5, a Poisson process. The results show that the arrivals generated for both the Poisson and self similar process are indeed Poisson and self similar, when seen at both scales. When seen as a whole, the diurnal process also results in higher Hurst estimates owing to high variability in the process.

TABLE II. HURST PARAMETER ESTIMATION

	Aggregate Variance	R/S	Periodo-gram	Abry-Veitch Estimator	Whittle Estimator
Small Part Of The Entire Inter-Arrival Time Series					
	Poisson				
	0.455	0.505	0.481	0.453	0.5
Bias	Self Similar				
0.728	0.698	0.626	0.695	0.509	0.5
0.75	0.761	0.636	0.757	0.656	0.559
alpha	Envelope-Guided				
1.1	0.465	0.501	0.492	0.559	0.5
1.5	0.447	0.491	0.49	0.547	0.5
1.9	0.499	0.482	0.504	0.526	0.5
Entire Arrivals Per Second Time Series					
	Poisson				
	0.447	0.002	0.469	0.529	0.5
Bias	Self Similar				
0.65	0.859	0.777	1.003	0.986	0.954
0.75	0.862	0.663	1.041	1.122	0.996
alpha	Envelope-Guided				
1.1	0.835	0.666	0.72	0.683	0.766
1.5	0.817	0.631	0.673	0.747	0.775
1.9	0.784	0.306	1.494	0.585	0.999

D. Demonstration

In this subsection, we demonstrate the need for realistic arrivals by evaluating a state-of-the-art key-value database library using requests generated by all three arrival models we implemented in YCSB.

LevelDB: LevelDB is an open-source embeddable database library, that was written at Google and follows the same design as BigTable's [33] tablet. Like many modern key-value databases that strive to offer both good random insert and good sequential read performance, it follows a Log-Structured Merge (LSM) [34] tree based data management. An LSM-tree contains multiple ordered log-structured indexes, one in the memory and the others on disk, and when any index exceeds a per-determined size threshold, parts of it are compacted and merged with the index in the next level.

The compaction process, that both cleans and reorganizes data, is typically implemented as a background process that co-exists with foreground requests. Compaction is either done periodically, as in LevelDB or during administrator specified time window, as in HBase [35]. In essence, many systems similar to LevelDB exists, with background tasks that compete with foreground requests and affects performance. Some other systems such as Wang *et al.* 's work [36], extends LevelDB by implementing optimized scheduling and dispatching polices for concurrent I/O requests, to exploit the parallelism in open-channel SSDs. Proper evaluation and comparison of such systems is possible only with realistic arrivals that generates both periods of activity and downtime realistically.

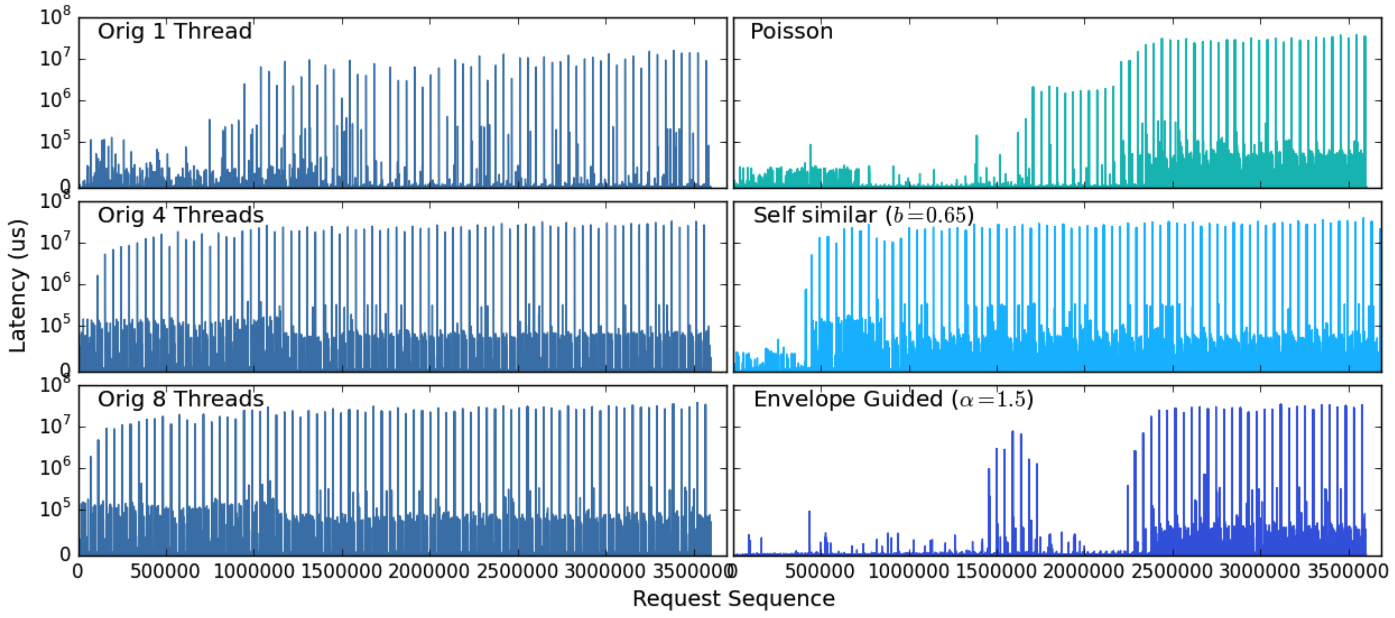
Hence, we have chosen LevelDB as the sample database to demonstrate the usefulness of our work. LevelDB is a user-level library that stores its data as files on the filesystem. Periodically, some of the files are to be compacted, and this happens along with serving incoming requests. As mentioned earlier, we run LevelDB on a dedicated 160GB hard disk drive, and storage management on the drive is done by an ext4 filesystem. LevelDB is an embeddable database and does not have a server communication component. Hence, as recommended by YCSB, we use MapKeeper server configured to use LevelDB as its datastore.

All experiments in this subsection performed 100% random inserts with 16 byte keys and 1 KB values. We ran original YCSB against LevelDB without specifying any arrival rate throttling using a single thread, 4 threads and 8 threads. The overall throughput achieved by the continuous bombardment of requests was 1712.35, 1736.6 and 1733.75 operations per second respectively. We then ran the same workload using a Poisson arrival process, a self similar process with bias 0.65 and a envelope-guided diurnal process with $\alpha = 1.5$ Pareto inter-arrivals, with 1700 average target arrivals per second. The observed overall throughput of the Poisson, self similar and envelope-guided processes are 1663.32, 1690.94, 1655.59 operations per second. For very similar throughputs (in the range 1655–1690), we observed the per-request latencies to vary tremendously.

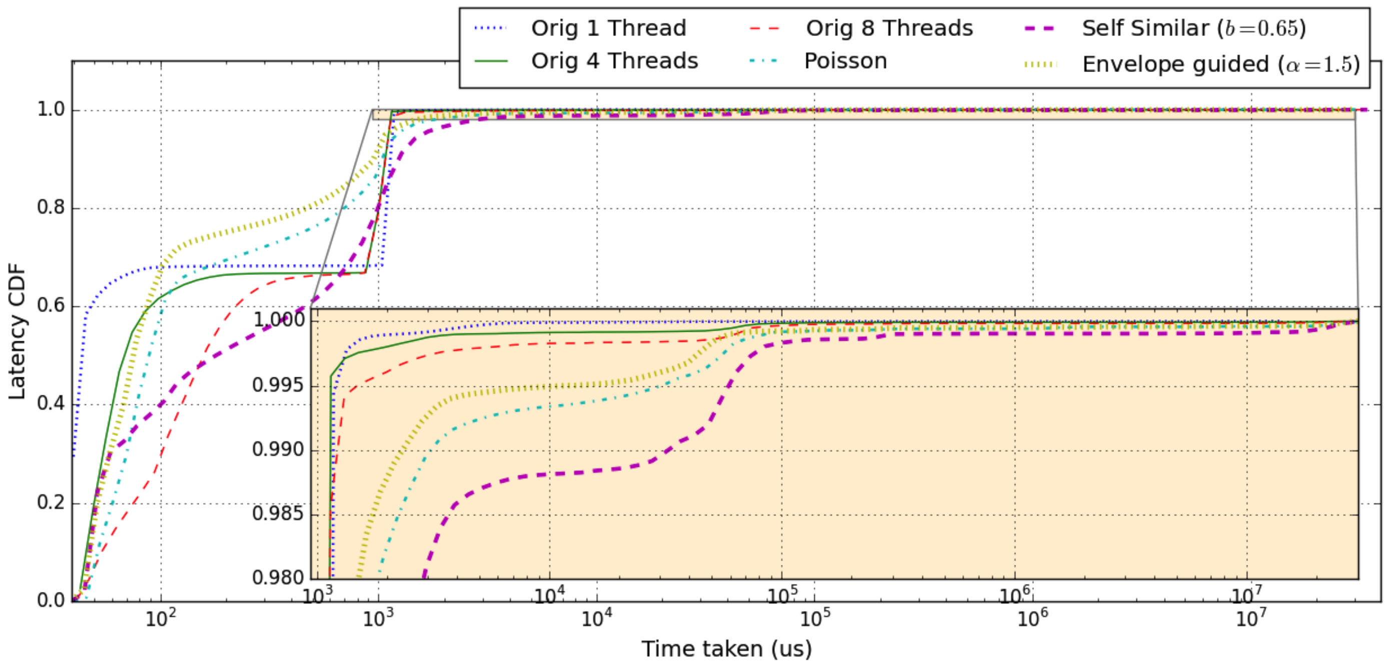
The results of the above experiment is shown in detail in Figure 10. Figure 10a is a semi-log plot of the observed per-request latencies over time, and Figure 10b is the normalized cumulative histogram on the latencies using logarithmic bins. Figure 10a is presented as a semi-log plot to highlight the requests that took really long time to complete. Without varying anything other than the arrival process, we could observe a high degree of variance in the observed latencies.

We can see that the peak delays appear consistently for three of the six runs, while it is absent for most of the time in both the Poisson traffic and the envelope-guided traffic. Determining the cause of these peak delays require an in-depth study of how LevelDB works. Suspecting background compactions, we measured the time spent in compaction during all these runs and found that the self similar traffic spent the least amount of time in compactions, and the single threaded original arrivals spent 141 seconds more than the self similar traffic on compaction. Even though self similar traffic produces the most bursty traffic, it also provides more down-time for the compaction to proceed uninterrupted, thus taking less time. This observation leads us to believe background compaction scheduling could bring the delays down, and our work could not only help identify such cases, but also aid in realistic evaluation of intelligent schemes such as compaction scheduling.

The latencies at the bottom in same graph appear much denser in cases where more requests bombard the database at any given time, that is in case of the multi-threaded original traffic and the self similar traffic. When the traffic is more smoother, more requests are completed sooner, as is evident from the results. This also indicates lack of better multi-request handling, while further investigation is required to confirm the same. The normalized cumulative histogram in Figure 10b also depicts how a workload's temporal characteristics affects the system behavior. The heavy tail highlighted in the figure shows the difference between the original back-to-back traffic and the realistic traffic we generate, and that systems that aim to bring down the heavy tail must most definitely be evaluated with realistic traffic.



(a) Semilog plot of observed latencies over the duration of the experiment.



(b) Normalized cumulative histogram of the observed latencies under various arrival patterns.

Fig. 10. Latencies measured while running a 1 KB random insert workload against LevelDB, tested under various arrival models, demonstrates the effects of realistic request arrivals.

VI. Future Work

We believe storage benchmarks for today's high performance storage systems are in need of features in addition to what we described in this work. In this section, we describe briefly the future directions we would like to take.

1) Scaled-up Realistic Request Arrivals: We have implemented realistic arrivals in a single YCSB client. To be able to generate requests at an arrival rate high enough for modern high performance storage systems, a single client is not sufficient even with a high number of threads. The approach recommended by YCSB is to use multiple clients at a same time for higher loads.

To generate higher loads in a realistic fashion, our work could be used in a user generative model, where each client chooses a model representative of a distinctive user who shares the underlying storage system with other users. For example, to generate realistic requests in a multi-tenant cloud storage system, each client would model a distinct tenant's access pattern. In the future, we would like to extend our work to perform multi-client co-ordinated request arrivals, where multiple clients together generate requests conforming to a single model.

2) Content Generation: Many modern high performance storage systems also perform additional functionalities such as compression and de-duplication to increase the overall throughput of the system. To compare and contrast these functionalities, it would be immensely helpful if standard benchmarks also come with realistic content generation with configurable levels of duplicity and compressibility. For example, YCSB generates values with random bytes and is unsuitable to evaluate such functionalities.

Realistic content generation is not just applicable to stored values, but also the keys in the recently popular variable-length key-value storage systems. Typically, these systems allow the keys to be of any length and not restricted to just numbers. The key-value stores, such as LevelDB, strive to order the keys and values as per the lexicographical order of these keys. The throughput of these systems are largely dependent on key content, as it can trigger different amounts of background compaction activity, and thus, it is important that benchmarks offer both realistic and configurable key content generation.

3) Correlation In Request Sizes: As mentioned earlier, in this work, we assumed that the request size distribution is independent of the request arrival rate, in line with the observation by Facebook's key-value workload study. But the study may not be representative of all use-cases of the key-value model. More real world workload studies on different kinds of key-value workloads are required to validate our assumption.

VII. Conclusion

Request arrival pattern can make a significant difference in the results of storage systems being evaluated, because performance observed under high yet steady client demand may actually be very different from that observed under bursty conditions. In this work, we categorized realistic request arrivals into three kinds, and implemented all three of them in the popular key-value storage benchmark, YCSB. We evaluated the arrivals we generated by showing that their statistical properties are both realistic and in line with commonly observed traffic patterns. We also demonstrated the effects of realistic arrivals on system behavior by evaluating a state-of-the-art key-value database, LevelDB, and conclude that to be representative of real world workloads, all storage benchmarks should provide the flexibility to generate realistic request arrivals.

Acknowledgment

The authors would like to thank members of the Storage Systems Research Center for their support and valuable feedback. This research was supported in part by the National Science Foundation under award IIP-1266400 and industrial members of the Center for Research in Storage Systems.

References

[1] B. Atikoglu, Y. Xu, E. Frachtenberg, S. Jiang, and M. Paleczny, "Workload Analysis of a Large-scale Key-value Store," in *Proceedings of the 12th ACM SIGMETRICS/PERFORMANCE Joint International Conference on Measurement and Modeling of Computer Systems (SIGMETRICS '12)*, 2012, pp. 53–64.

[2] "The Seagate Kinetic Open Storage Vision," http://www.seagate.com/tech-insights/kinetic-vision-how-seagate-new-developer-tools-meets-the-needs-of-cloud-storage-platforms-master-ti/.

[3] B. F. Cooper, A. Silberstein, E. Tam, R. Ramakrishnan, and R. Sears, "Benchmarking Cloud Serving Systems with YCSB," in *Proceedings of the 1st ACM Symposium on Cloud Computing (SoCC '10)*, 2010, pp. 143–154.

[4] R. Escriva, B. Wong, and E. G. Sirer, "HyperDex: A Distributed, Searchable Key-value Store," in *Proceedings of the ACM SIGCOMM 2012 Conference on Applications, Technologies, Architectures, and Protocols for Computer Communication (SIGCOMM '12)*, 2012, pp. 25–36.

[5] B. Fan, D. G. Andersen, and M. Kaminsky, "MemC3: Compact and Concurrent MemCache with Dumber Caching and Smarter Hashing," in *Proceedings of the 10th USENIX Conference on Networked Systems Design and Implementation (NSDI '13)*, 2013, pp. 371–384.

[6] H. Lim, B. Fan, D. G. Andersen, and M. Kaminsky, "SILT: A Memory-efficient, High-performance Key-value Store," in *Proceedings of the Twenty-Third ACM Symposium on Operating Systems Principles (SOSP '11)*, 2011, pp. 1–13.

[7] M. Ferdman, A. Adileh, O. Kocberber, S. Volos, M. Alisafaee, D. Jevdjic, C. Kaynak, A. D. Popescu, A. Ailamaki, and B. Falsafi, "Clearing the Clouds: A Study of Emerging Scale-out Workloads on Modern Hardware," in *Proceedings of the Seventeenth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS XVII)*, 2012, pp. 37–48.

[8] A. Al-Shishtawy and V. Vlassov, "ElastMan: Elasticity Manager for Elastic Key-value Stores in the Cloud," in *Proceedings of the 2013 ACM Cloud and Autonomic Computing Conference (CAC '13)*, 2013, pp. 7:1–7:10.

[9] D. Shue, M. J. Freedman, and A. Shaikh, "Performance Isolation and Fairness for Multi-tenant Cloud Storage," in *Proceedings of the 10th USENIX Conference on Operating Systems Design and Implementation (OSDI '12)*, 2012, pp. 349–362.

[10] S. Das, S. Nishimura, D. Agrawal, and A. El Abbadi, "Albatross: Lightweight Elasticity in Shared Storage Databases for the Cloud Using Live Data Migration," *Proceedings of the VLDB Endowment*, vol. 4, no. 8, pp. 494–505, May 2011.

[11] A. J. Elmore, S. Das, D. Agrawal, and A. El Abbadi, "Zephyr: Live Migration in Shared Nothing Databases for Elastic Cloud Platforms," in *Proceedings of the 2011 ACM SIGMOD International Conference on Management of Data (SIGMOD '11)*, 2011, pp. 301–312.

[12] "LevelDB." https://github.com/google/leveldb.

[13] C. Delimitrou, S. Sankar, K. Vaid, and C. Kozyrakis, "Decoupling Datacenter Studies from Access to Large-scale Applications: A Modeling Approach for Storage Workloads," in *Proceedings of the 2011 IEEE International Symposium on Workload Characterization (IISWC '11)*, 2011, pp. 51–60.

[14] D. Meisner, J. Wu, and T. F. Wenisch, "BigHouse: A Simulation Infrastructure for Data Center Systems," in *IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS '12)*, 2012, pp. 35–45.

[15] C. L. Abad, H. Luu, N. Roberts, K. Lee, Y. Lu, and R. H. Campbell, "Metadata Traces and Workload Models for Evaluating Big Storage Systems," in *Proceedings of the 2012 IEEE/ACM Fifth International Conference on Utility and Cloud Computing (UCC '12)*, 2012, pp. 125–132.

[16] N. Mi, G. Casale, L. Cherkasova, and E. Smirni, "Injecting Realistic Burstiness to a Traditional Client-server Benchmark," in *Proceedings of the 6th International Conference on Autonomic Computing (ICAC '09)*, 2009, pp. 149–158.

[17] M. Wang, N. H. Chan, S. Papadimitriou, C. Faloutsos, and T. Madhyastha, "Data Mining Meets Performance Evaluation: Fast Algorithms

for Modeling Bursty Traffic," in *Proceedings of the 18th International Conference on Data Engineering (ICDE '02)*, 2002.

[18] B. Hong and T. M. Madhyastha, "The Relevance of Long-Range Dependence in Disk Traffic and Implications for Trace Synthesis," in *Proceedings of the 22Nd IEEE / 13th NASA Goddard Conference on Mass Storage Systems and Technologies (MSST '05)*, 2005, pp. 316–326.

[19] S. Patil, M. Polte, K. Ren, W. Tantisiriroj, L. Xiao, J. López, G. Gibson, A. Fuchs, and B. Rinaldi, "YCSB++: Benchmarking and Performance Debugging Advanced Features in Scalable Table Stores," in *Proceedings of the 2nd ACM Symposium on Cloud Computing (SOCC '11)*, 2011, pp. 9:1–9:14.

[20] D. G. Feitelson, "Workload Modeling for Computer Systems Performance Evaluation," This book is being published by Cambridge University Press. http://www.cs.huji.ac.il/ feit/wlmod/wlmod.pdf, 2014.

[21] V. Paxson and S. Floyd, "Wide Area Traffic: The Failure of Poisson Modeling," *IEEE/ACM Transactions on Networking (TON)*, vol. 3, no. 3, pp. 226–244, Jun. 1995.

[22] J. Cao, W. S. Cleveland, D. Lin, and D. X. Sun, "On the Nonstationarity of Internet Traffic," in *Proceedings of the 2001 ACM SIGMETRICS International Conference on Measurement and Modeling of Computer Systems (SIGMETRICS '01)*, 2001, pp. 102–112.

[23] M. E. Crovella and A. Bestavros, "Self-similarity in World Wide Web Traffic: Evidence and Possible Causes," *IEEE/ACM Transactions on Networking (TON)*, vol. 5, no. 6, pp. 835–846, Dec. 1997.

[24] W. E. Leland, M. S. Taqqu, W. Willinger, and D. V. Wilson, "On the Self-similar Nature of Ethernet Traffic (Extended Version)," *IEEE/ACM Transactions on Networking (TON)*, vol. 2, no. 1, pp. 1–15, Feb. 1994.

[25] S. D. Gribble, G. S. Manku, D. Roselli, E. A. Brewer, T. J. Gibson, and E. L. Miller, "Self-similarity in File Systems," in *Proceedings of the 1998 ACM SIGMETRICS Joint International Conference on Measurement and Modeling of Computer Systems*, 1998, pp. 141–150.

[26] 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 (WWC '98)*, 1998, pp. 97–.

[27] A. Riska and E. Riedel, "Long-Range Dependence at the Disk Drive Level," in *Proceedings of the 3rd International Conference on the Quantitative Evaluation of Systems (QEST '06)*, 2006, pp. 41–50.

[28] T. Karagiannis, M. Faloutsos, and M. Molle, "A User-friendly Self-similarity Analysis Tool," *ACM SIGCOMM Computer Communication Review*, vol. 33, no. 3, pp. 81–93, Jul. 2003.

[29] T. Karagiannis, M. Faloutsos, and R. H. Riedi, "Long-range dependence: now you see it, now you don't!" in *Global Telecommunications Conference (GLOBECOM '02)*, vol. 3, Nov 2002, pp. 2165–2169.

[30] T. Akgul, S. Baykut, M. Erol-Kantarci, and S. Oktug, "Periodicity-Based Anomalies in Self-Similar Network Traffic Flow Measurements," *IEEE Transactions on Instrumentation and Measurement*, vol. 60, no. 4, pp. 1358–1366, April 2011.

[31] M. Roughan, A. Greenberg, C. Kalmanek, M. Rumsewicz, J. Yates, and Y. Zhang, "Experience in Measuring Backbone Traffic Variability: Models, Metrics, Measurements and Meaning," in *Proceedings of the 2nd ACM SIGCOMM Workshop on Internet Measurement (IMW '02)*, 2002, pp. 91–92.

[32] "MapKeeper." https://github.com/m1ch1/mapkeeper/.

[33] F. Chang, J. Dean, S. Ghemawat, W. C. Hsieh, D. A. Wallach, M. Burrows, T. Chandra, A. Fikes, and R. E. Gruber, "Bigtable: A Distributed Storage System for Structured Data," *ACM Transactions on Computer Systems (TOCS)*, vol. 26, no. 2, pp. 4:1–4:26, Jun. 2008.

[34] P. O'Neil, E. Cheng, D. Gawlick, and E. O'Neil, "The Log-structured Merge-tree (LSM-tree)," *Acta Informatica*, vol. 33, no. 4, pp. 351–385, Jun. 1996.

[35] "HBase." https://hbase.apache.org/.

[36] P. Wang, G. Sun, S. Jiang, J. Ouyang, S. Lin, C. Zhang, and J. Cong, "An Efficient Design and Implementation of LSM-tree Based Key-value Store on Open-channel SSD," in *Proceedings of the Ninth European Conference on Computer Systems (EuroSys '14)*, 2014, pp. 16:1–16:14.