Performance Evaluation 70 (2013) 704–719

Contents lists available at ScienceDirect

Performance Evaluation

journal homepage: www.elsevier.com/locate/peva

Generating request streams on Big Data using clustered renewal processes

Cristina L. Abad [a,b,c,*], Mindi Yuan [a], Chris X. Cai [a], Yi Lu [a], Nathan Roberts [b], Roy H. Campbell [a]

[a] University of Illinois at Urbana-Champaign, United States
[b] Yahoo! Inc., United States
[c] Escuela Superior Politécnica del Litoral, Ecuador

ARTICLE INFO

Article history:
Available online 28 August 2013

Keywords:
Big Data
Workload generation
HDFS
Popularity
Temporal locality
Storage

ABSTRACT

The performance evaluation of large file systems, such as storage and media streaming, motivates scalable generation of representative traces. We focus on two key characteristics of traces, *popularity* and *temporal locality*. The common practice of using a system-wide distribution obscures per-object behavior, which is important for system evaluation. We propose a model based on *delayed renewal processes* which, by sampling interarrival times for each object, accurately reproduces popularity and temporal locality for the trace. A lightweight version reduces the dimension of the model with statistical clustering. It is workload-agnostic and object type-aware, suitable for testing emerging workloads and 'what-if' scenarios. We implemented a synthetic trace generator and validated it using: (1) a Big Data storage (HDFS) workload from Yahoo!, (2) a trace from a feature animation company, and (3) a streaming media workload. Two case studies in caching and replicated distributed storage systems show that our traces produce application-level results similar to the real workload. The trace generator is fast and readily scales to a system of 4.3 million files. It outperforms existing models in terms of accurately reproducing the characteristics of the real trace.

1. Introduction

Workload generation is often used in simulations and real experiments to help reveal how a system reacts to variations in the load [1]. Such experiments can be used to validate new designs, find potential bottlenecks, evaluate performance, and do capacity planning based on observed or predicted workloads.

Workload generators can replay real traces or do model-based synthetic workload generation. Real traces capture observed behavior and may even include nonstandard or undiscovered (but possibly important) properties of the load [2]. However, real trace-based approaches treat the workload as a "black box" [1]. Modifying a particular workload parameter or dimension is difficult, making such approaches inappropriate for sensitivity and what-if analysis. Sharing of traces can be hard because of their size and privacy concerns. Other problems include those of scaling to a different system size and describing and comparing traces in terms that can be understood by implementors [2].

Model-based synthetic workload generation can be used to facilitate testing while modifying a particular dimension of the workload, and can model expected future demands. For that reason, synthetic workload generators have been

* Correspondence to: Facultad de Ingeniería en Electricidad y Computación (FIEC), Escuela Superior Politécnica del Litoral (ESPOL), Campus Gustavo Galindo, Km 30.5 Vía Perimetral, Guayaquil, Ecuador.
E-mail addresses: cabad@illinois.edu (C.L. Abad), myuan5@illinois.edu (M. Yuan), xiaocai2@illinois.edu (C.X. Cai), yilu4@illinois.edu (Y. Lu), nroberts@yahoo-inc.com (N. Roberts), rhc@illinois.edu (R.H. Campbell).

0166-5316/$ – see front matter © 2013 Elsevier B.V. All rights reserved.
http://dx.doi.org/10.1016/j.peva.2013.08.006

used extensively to evaluate the performance of storage systems [2,3], media streaming servers [4,5], and Web caching systems [1,6]. Synthetic workload generators can issue requests on a real system [1,2] or generate synthetic traces that can be used in simulations or replayed on actual systems [5,7].

In this work, we focus on synthetic generation of *object request streams*,[1] which may refer to different object types depending on context, like files [3], disk blocks [2], Web documents [6], and media sessions [5].

Two important characteristics of object request streams are popularity (access counts) and temporal reference locality (a recently accessed object is likely to be accessed in the near future) [5]. While highly popular objects are likely to be accessed again soon, temporal locality can also arise when the interarrival times are highly skewed, even if the object is unpopular [8].

For the purpose of synthetic workload generation, it is desirable to simultaneously reproduce the access counts and the request interarrivals of each individual object, as both of these dimensions can affect system performance. However, single-distribution approaches – which summarize the behavior of different types of objects with a single distribution per dimension – cannot accurately reproduce both at the same time. In particular, the common practice of collapsing the per-object interarrival distributions into a single system-wide distribution (instead of individual per-object distributions) obscures the identity of the object being accessed, thus homogenizing the otherwise distinct per-object behavior [7].

As Big Data applications lead to emerging workloads and these workloads keep growing in scale, the need for workload generators that can scale up the workload and/or facilitate its modification based on predicted behavior is increasingly important.

Motivated by previous observations about Big Data file request streams [9–11], we set the following goals for our model and synthetic generation process:

- *Support for dynamic object populations*: Most previous models consider static object populations. Several workloads including storage systems supporting MapReduce jobs [10] and media server sessions [5] have dynamic populations with high object churn.
- *Fast generation*: Traces in the Big Data domain can be large (e.g., 1.6 GB for a 1-day trace with millions of objects). A single machine should be able to generate a synthetic trace modeled after the original one without suffering from memory or performance constraints.
- *Type-awareness*: Request streams are composed of accesses to different objects, each of which may have distinct access patterns. We want to reproduce these access patterns.
- *Workload-agnostic*: The Big Data community is creating new workloads (e.g., key-value stores [12], batch and interactive MapReduce jobs [10], etc.). Our model should not make workload-dependent assumptions that may render it unsuitable for emerging workloads.

In this paper, we consider a stationary segment[2] of the workload and describe a model based on a set of delayed renewal processes (one per object in the stream) in which the system-wide popularity distribution asymptotically emerges through explicit reproduction of the per-object request arrivals and active span (time during which an object is accessed). However, this model is unscalable, as it is heavy on resources (needs to keep track of millions of objects).

We propose a lightweight version of the model that uses unsupervised statistical clustering to identify groups of objects with similar behavior and significantly reduce the model space by modeling "types of objects" instead of individual objects. As a result, the clustered model is suitable for synthetic generation.

We implemented a synthetic trace generator based on our model, and evaluate it across several dimensions. Using a Big Data storage (HDFS [13]) workload from Yahoo!, we validate our approach by demonstrating its ability to approximate the original request interarrivals and popularity distributions (supremum distance between real and synthetic cumulative distribution function, CDF, under 2%). Workloads from other domains were also modeled successfully (1.3–2.6% distance between real and synthetic CDFs). Through a case study in Web caching and a case study in the Big Data domain (load in a replicated distributed storage system), we show how our synthetic traces can be used in place of the real traces (results within 5.5% points of the expected or real results), outperforming previous models.

Our model can accommodate for objects appearing and disappearing at any time during the request stream (making it appropriate for workloads with high object churn) and is suitable for synthetic workload generation; experiments show that we can generate a 1-day trace with more than 60 million object requests in under 3 min. Furthermore, our assumptions are minimal, since the renewal process theory does not require that the model be fit to a particular interarrival distribution, or to a particular popularity distribution.

Additionally, the use of unsupervised statistical clustering leads to autonomic "type-awareness" that does not depend on expert domain knowledge or introduce human biases. The statistical clustering finds objects with similar behavior, enabling type-aware trace generation, scaling, and "what-if" analysis (e.g., in a storage system, what if the short-lived files were to increase in proportion to the other types of files?)

Concretely, the technical contributions of this paper are the following: (1) we present a model based on a set of delayed renewal processes in which the system-wide popularity distribution asymptotically emerges through explicit reproduction of the per-object request interarrivals and active span; (2) we use clustering to build a lightweight clustered variant of

[1] Also called *object reference streams*.

[2] Workloads consisting of a few stationary segments can be divided using the approach in [2]; for more details, see Section 5.1.

$$M = \{\{F_1, F_2, ..., F_n\}, \{T_{1_1}, T_{2_1}, ..., T_{n_1}\}, \{t_1, t_2, ..., t_n\},\ t_{end}\}$$

n: number of objects in request stream
O_i: object i, where $i \in \{1, ..., n\}$
F_i: interarrival distribution for O_i
T_{i_1}: time at which O_i becomes active; i.e., time of 1^{st} access to O_i
t_i: time (since T_{i_1}) at which O_i becomes inactive; i.e., active span
t_{end}: duration of trace (in milliseconds)

Fig. 1. Statistical parameters that describe our system model.

the model, suitable for synthetic workload generation; and (3) we show that clustering enables workload-agnostic type-awareness, which can be exploited during scaling, what-if and sensitivity analysis.

This paper is structured as follows. In Section 2 we describe our system model, and explain why it can approximate an object's access count based on the object's temporal behavior. In Section 3, we show how we can use statistical clustering to reduce the model size and still achieve high accuracy in synthetic trace generation. In Section 4, we evaluate the effectiveness of our model in producing the same results as the real trace using two case studies in the Web caching and Big Data domains. In Section 5 we discuss some benefits and limitations of our model. We discuss related work in Section 6. Finally, we conclude in Section 7.

2. Model

Consider an object request stream that accesses n distinct objects $\{O_1, O_2, \ldots, O_n\}$ during $[0, t_{\text{end}}]$. We model the object request stream as a set of *renewal processes* in which each object (or file, in the context of this paper) has an independent interarrival distribution. The file population may not be static, and not all files may exist (or be active) at time 0. Thus, we consider the case of *delayed renewal processes*, in which the first arrival is allowed to have a different interarrival time distribution. For a brief summary of renewal processes, see the Appendix.

Our model is defined by $\{F_1, F_2, \ldots, F_n\}$, $\{G_1, G_2, \ldots, G_n\}$, and $\{t_1, t_2, \ldots, t_n\}$.[3] F_i is the interarrival distribution of the accesses to object i or O_i. G_i is the interarrival time distribution of the first access to O_i. t_i is the observed *active span* for O_i, or the difference or period between the first and last accesses to O_i.

Each renewal process is modeled after the behavior of a single object in the request stream. For this reason, the interarrival time for the first access to O_i, given by G_i, has only one possible outcome: the time when the first access to O_i was observed in the original request stream, or T_{i_1}. The model is summarized in Fig. 1.

Thus, we have one arrival time process T_i for every object O_i in the system. T_i is the partial sum process associated with the independent, identically distributed sequence of interarrival times; F_i is the common distribution function of the interarrival times. A particular O_i's popularity is given by the counting process N_i. The random counting process is the inverse of the arrival time process.

We use an O_i's corresponding t_i to determine when to stop generating arrivals (sampling from F_i), at which point the number of accesses (N_{t_i}) is evaluated.

Fig. 2 shows the synthetic trace generation algorithm based on our model.

2.1. Convergence to real distributions

In this section, we refer to the distributions obtained from the real trace as the *real distributions*. Mainly, we are concerned with reproducing the per-object interarrival distribution and the popularity distribution. In this section, we explain why these distributions asymptotically converge to the real ones.

2.1.1. Per-object interarrival distribution

The real per-object interarrival distribution is obtained by calculating the time between an access to an object and the previous access to that same object, when the number of accesses to the object is greater than 1.

Let $\hat{F}_n(t)$ be the estimator, or empirical distribution function, obtained by sampling from the real distribution $F(t)$. By the strong law of large numbers, we have that the estimator $\hat{F}_n(t)$ converges to $F(t)$ with probability one as $n \to \infty$.

2.1.2. Object popularity distribution

From the renewal theorem, we know that the expected number of renewals or arrivals in an interval is asymptotically proportional to the length of the interval: $m(t, t + h] \to h/\mu$ as $t \to \infty$, where μ is the mean interarrival time. Since the

[3] In this work, we use real time (in milliseconds). The choice of real time versus virtual time, which is discrete and advances by one unit with each object reference, is typically determined by project goals. Workload generators tend to prefer real time [4,5], while workload characterization projects favor virtual time [8,14].

```
for i ← 1 to n do
  t ← T_{i_1}                                  ┐
  span ← t_i                                   │
  last ← t + span                              │
  while t ≤ t_end and t ≤ last do              │ a renewal
    print t, O_i                               │ process
    interarrival ← sample from F_i             │
    t ← t + interarrival                       │
  end while                                    ┘
end for
sort trace
```

Fig. 2. Synthetic trace generation algorithm.

$$M = \{\{F_1, F_2, ..., F_k\}, \{G_1, G_2, ..., G_k\}, \{H_1, H_2, ..., H_k\}, n, \{w_1, w_2, ..., w_k\}, t_{end}\}$$

n: number of objects in request stream
O_i: object i, where $i \in \{1, ..., n\}$
k: number of clusters (object types)
K_j: cluster j, where $j \in \{1, ..., k\}$
F_j: interarrival distribution for accesses to an object in cluster K_j
G_j: interarrival distribution for first access to objects in cluster K_j
H_j: active span distribution $\forall O_i \in K_j$
w_j: percentage of objects in K_j ; $\sum_{i=1}^{k} w_i = 1$
t_{end}: duration of trace (in milliseconds)

Fig. 3. Statistical parameters that describe the reduced, clustered model; $k \ll n$.

distribution of interarrivals comes from the observed empirical distribution, $\mu = \frac{1}{n}\sum_{i=1}^{n} x_i$, where $\sum_{i=1}^{n} x_i$ is the sum of the observed interarrivals, or the observed active span; since we sample only during the observed active span, $\sum_{i=1}^{n} x_i = h$. Thus, we have $m(t, t+h] \to h/(1/n \times h)$ as $t \to \infty$ or $m(t, t+h] \to n$ as $t \to \infty$, where n is the observed number of renewals.

2.2. Discussion

The asymptotic behavior of the interarrivals and counting process does not imply that a single run of our algorithm will generate a trace in which each object's behavior is statistically reproduced. A way to reach the asymptotic behavior is to perform a Monte Carlo simulation to ensure that the sequence of interarrivals of an O_i contains the expected number of renewals or requests (h/μ).

However, our experimental results (Section 3.5) show that for a large trace with a large number of files, the synthetic distributions can approximate the real ones in a single run.

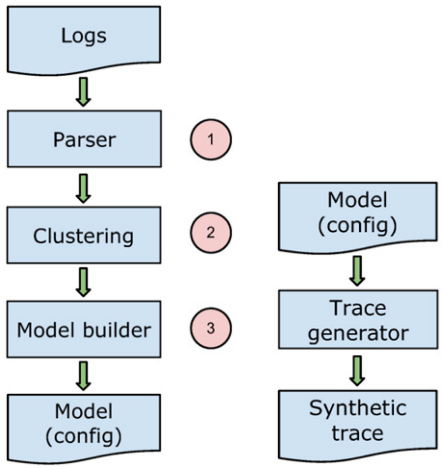
On the other hand, a problem with the model presented in this section is that it is not scalable, as it needs to keep track of millions of distributions. Next, we propose an approach that uses statistical clustering to reduce the model size, making it suitable for synthetic workload generation.

3. Reducing the model size via clustering

In this section, we describe how we can use unsupervised statistical clustering to reduce the state space of our model to a tiny fraction of its original size.

The basic idea is to cluster objects with "similar" behavior so that we only need to keep track of the temporal access patterns of the *cluster*, not those of each object; thus, we reduce the state space, as shown in Fig. 3. The reduction in size that this approach entails is quantified in Section 3.5.

The workflow we describe in this section is as follows (Fig. 4): (a) Build model, once per source workload; (a.1) Parser: Extracts features for clustering; (a.2) Clustering: Use k-means to find similar objects; (a.3) Model builder: Use data-processing



(a) Model building. (b) Trace generation.

Fig. 4. Process of how the model is built (left), and how the synthetic traces are generated (right).

tools, like SQL, and standard techniques to get per-cluster distributions; (b) Generate synthetic trace: Multiple traces can be generated, based on a particular model.

3.1. Dataset description

We analyzed a 1-day (Dec. 1, 2011) *namespace metadata trace*[4] from an Apache Hadoop cluster at Yahoo!. The trace came from a 4100+ node production cluster running the Hadoop Distributed File System (HDFS) [13].[5] We obtained the trace of namespace events by parsing the metadata server audit logs. For the purpose of this paper we analyzed only the open events, which constitute the file request stream. As it came from a Big Data cluster, the 1-day trace is quite big, containing 60.9 million object requests (opens) that target 4.3 million distinct files, and 4 PB of used storage space. Apache Hadoop is an open source implementation of Google's MapReduce [15] framework, used for data-intensive jobs. In prior work, we presented a detailed workload characterization of how the MapReduce jobs operating in this cluster interact with the storage layer [10].

3.2. Clustering similar objects

We begin by explaining what "similar behavior" means in the context of this work. The goal is to cluster objects with the same interarrival distribution, so that sampling from one common (per-cluster) distribution reproduces the per-object interarrivals of the objects in the cluster.

We used *k*-means for the unsupervised statistical clustering. *k*-means is a well-known clustering algorithm that partitions a dataset into *k* clusters [16]. Each observation in the dataset is described by several features or dimensions. The output of the *k*-means algorithm is the center of each cluster, and a list with the cluster identifier to which each of the original observations belongs (the closest center).

We propose the use of two per-object features as input to *k*-means: skewness and average interarrival time (span/accessCount). We next describe the reasoning behind those choices.

It is not possible to know *a priori* if the interarrival of an object can be represented by a known distribution. Thus, comparing two interarrival distributions is not as simple as comparing two distribution parameters, but would rather entail comparing the two empirical distributions in some other way. So that *k*-means can perform that comparison efficiently, it is desirable to summarize each distribution with as few numbers as possible. We choose *skewness* as a metric that can help describe the shape of the per-object interarrival distribution. We use the *Bowley skewness*, or quartile skewness, as the metric: $(Q_1 - 2Q_2 + Q_3)/(Q_3 - Q_1)$, where Q_i is the ith quartile.

Other ways of describing a statistical distribution, such as quartiles or the five-number summary,[6] could be used instead. We did not explore those options, since our experiments show that skewness (combined with the average interarrival time) works well in practice. Furthermore, using multi-value representations of each distribution would require designing a way

[4] We define a *namespace metadata trace* as a storage system trace that contains a snapshot of the namespace (file and directory hierarchy) and a set of events that operate atop that namespace (e.g., open a file, list directory contents) [3]. These traces can be used to evaluate namespace management systems, including their load balancing, partitioning, and caching components.

[5] For a brief description of the design of HDFS, see Section 4.2.

[6] Sample minimum, first quartile, median, third quartile, and sample maximum.

```
for j ← 1 to k do
  for o ← 1 to w_j × n do
    t ← sample first arrival from G_j
    span ← sample active span from H_j
    last ← t + span
    while t ≤ t_end and t ≤ last do
      print t, id(o, j)
      interarrival ← sample from F_j
      t ← t + interarrival
    end while
  end for
end for
sort trace
```

a renewal process

Fig. 5. Synthetic trace generation algorithm for the clustered model.

to find the "distance" between two different multi-value summaries; a task that is nontrivial. In our current implementation, we are using the Euclidean distance on z-score normalized values; this is the default distance metric used by many k-means implementations.

The choice of *average interarrival time* as a feature ensures a strong correlation between the active span or period during which an object is accessed, and the number of accesses in that period.

3.3. Synthetic trace generation

The trace generation algorithm is shown in Fig. 5.

The number of accesses generated for O_i, belonging to cluster j, depends on the per-object interarrival distribution for cluster j (F_j) and the active span distribution for the objects in K_j (H_j). Thus, a good clustering should lead to a strong correlation between the span and access count of the objects in the cluster, which is achieved through the choice of average interarrival time (span/accessCount) as one of the features used in the clustering step.

The number of renewal processes used during trace generation remains the same as in the full model: n, or the number of distinct objects accessed in the request stream. Clustering does not reduce the number of renewal processes used during workload generation; it reduces the number of distributions that we need to keep track of, and thus reduces the memory requirements of the workload generation process. It also reduces the storage space used by the model or configuration file.

Scaling: We can increase the number of objects in the synthetic workload by increasing n; the proportion of each type of object (given by w_i) is maintained. Alternatively, we can increase the number of only one particular type of object by increasing n and doing a transformation on the w_i weights. For example, to double the number of objects of type 1 while keeping the number of objects of types 2 to k unchanged, we can obtain the values of n_{new} and $\{w_{1_{\text{new}}}, w_{2_{\text{new}}}, \ldots, w_{k_{\text{new}}}\}$ as follows:

$$
\begin{aligned}
n_{\text{new}} &= n + w_1 \times n \\
w_{1_{\text{new}}} \times n_{\text{new}} &= 2(w_1 \times n) \\
w_{i_{\text{new}}} \times n_{\text{new}} &= w_i \times n, \quad i \in \{2, \ldots, k\}.
\end{aligned}
$$

3.4. Optimization: Fitting the tail of the popularity distribution

The model described so far is successful at approximating the popularity distribution (see Section 3.5). However, it is not able to reproduce the tail of the distribution with the most popular objects.

A common characteristic of popularity distributions is a small percentage of highly popular objects (e.g., Zipfian popularity distributions) [10,17–19]. These highly popular objects are too few, and their access counts too different from each other, for their asymptotic popularity to be reproduced in one single run of our generation process (see the ragged tail in Fig. 6). While some systems may not be sensitive to the tail of highly popular objects (e.g., Web caching, discussed in Section 4.1), other systems are (e.g., replicated storage, discussed in Section 4.2). To address this problem, we define the following approach and heuristic.

Optimization: Use the full model instead of the clustered model for the highly popular objects located at the tail of the popularity distribution. Before generating the interarrivals for an object in the tail, determine the expected number of arrivals for the object during its span (see Section 2.1). Then, sample sets of arrivals until we find one set whose number of arrivals is within a small %, δ, of the expected number of arrivals. Use that set as the sampled interarrivals for that object, thus simultaneously reproducing the object's access counts, interarrivals, and span.

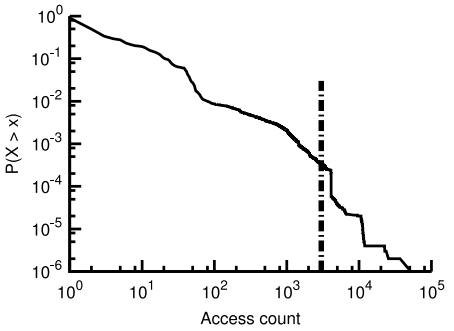
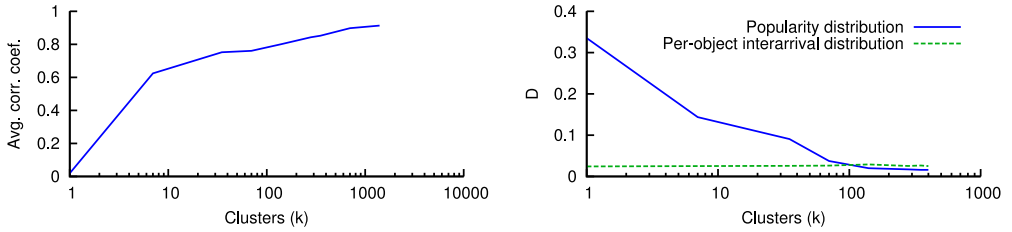


Fig. 6. Complementary CDF of the popularity distribution. The CCDF highlights the tail of very popular objects. The vertical dashed line shows the (heuristical) beginning of the tail.



(a) Correlation coefficient. (b) *D*.

Fig. 7. *Left*: Pearson's correlation coefficient between object popularity (access count) and span, for all objects in a cluster, averaged across clusters. *Right*: Kolmogorov–Smirnov distance (*D*), comparing the real distribution to the synthetic one. *D* converges to 0 if the sample comes from the target distribution.

In our current implementation we keep sampling sets of interarrivals until we find one within $\delta = 0.5\%$ of the expected number of arrivals for that object.

Heuristic: We define the beginning of the tail of highly popular objects as the position where: (1) access counts are held by single files, and (2) distribution has sparse access counts. For our trace, we defined the tail to begin at the 1478th to last file in our trace. Only one file had 3000 accesses; one file had 3001 accesses; and one file had 3010 accesses. No file was accessed 3002–3009 times. This is quite noticeable in Fig. 6, where the tail of the complementary CDF is ragged and not smooth.

3.5. Experimental validation

We implemented the components described in Fig. 4 as follows: (a.1) Apache Pig and bash scripts; (a.2) *R* using the fpc package [20]; (a.3) Apache Pig and bash scripts; and (b.1) Java.

We ran experiments for increasingly larger numbers of clusters and discuss the results in this section. Unless otherwise noted, the optimization to fit the tail was not used in the results shown in this section.

Fig. 7(a) shows Pearson's correlation coefficient between the span and the access count of the objects in the cluster, averaged across all clusters. We can observe that increasing the number of clusters leads to a higher correlation.

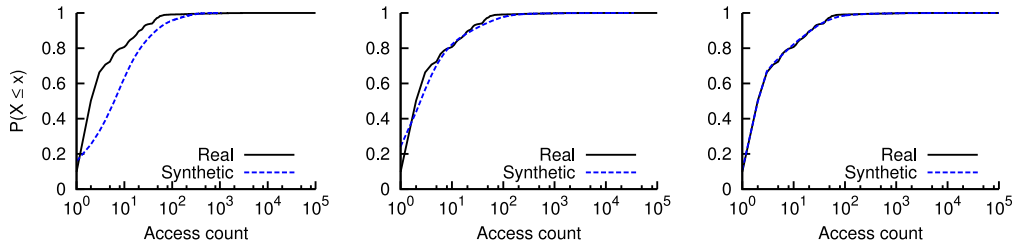
Fig. 7(b) shows how the popularity and per-object interarrival distributions approximate the real distributions. We evaluate the closeness of the approximation using the Kolmogorov–Smirnov distance, *D*, which is the greatest (supremum) distance between the two CDFs.

By comparing Fig. 7(a) and (b), we can appreciate the usefulness of the correlation coefficient between span and access count as a metric to evaluate whether the clustering results will be useful for synthetic trace generation. In our experiments, an average correlation coefficient of 0.76 or higher ($k > 70$) enables us to approximate the popularity distribution within 3% of the real one, and a correlation of 0.8 or higher ($k = 140$) leads to an approximation within 2% of the real CDF.

Fig. 8 provides a visual confirmation of the effectiveness of our approach in approximating the real popularity distribution as the number of clusters increases.

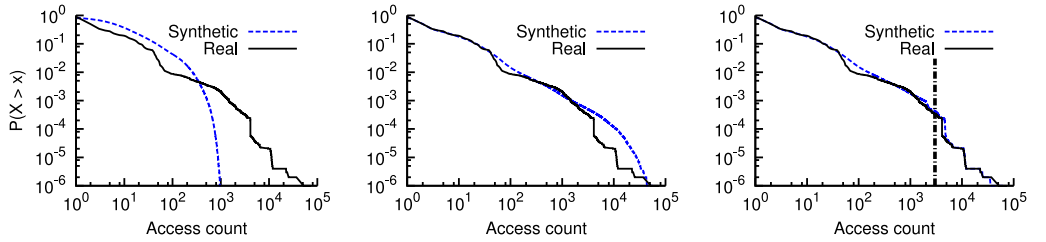
There is a trade-off between approximating the popularity distribution and the size of the model. The extreme cases are when $k = 1$ and when no clustering is performed ($k = n$): we need to keep track of three distributions ($k = 1$)[7] vs. millions of distributions ($k = n$). In our experiments, using 70–400 clusters led to results that approximate the popularity distribution well (with a small *D* value and overlapping CDFs). For our dataset, when $k = 400$, the configuration file can be as small as 24 kB if the distributions are fitted to known distributions and only their parameters are stored, or as big as 190 MB if the distributions are stored as uncompressed empirical histograms (0.0001 precision) or 20 MB if compressed.

[7] F_1, G_1, and H_1.



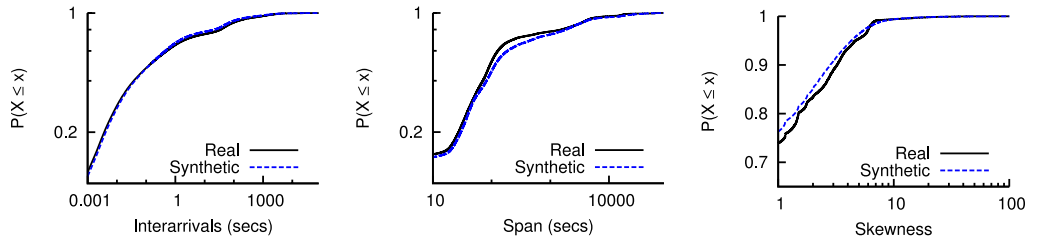
(a) $k = 1$. (b) $k = 7$. (c) $k = 400$.

Fig. 8. Cumulative distribution functions (CDFs) of the real and synthetic popularity distributions, when using 1, 7, and 400 clusters. More clusters lead to a better approximation of the popularity distribution.



(a) $k = 1$. (b) $k = 400$. (c) $k = 350$ w/TF.

Fig. 9. Complementary CDF of the popularity distribution, when using 1 cluster, 400 clusters, and 350 clusters with the tail-fitting (TF) technique. The CCDF highlights the tail of very popular objects. The vertical dashed line in (c) shows the beginning of the fitted tail.



(a) Interarrivals. (b) Span. (c) Skewness.

Fig. 10. Real and synthetic ($k = 400$) per-object interarrival, span and skewness CDFs.

We now evaluate the effectiveness of the technique to fit the tail of the popularity distribution. Fig. 9(a) and (b) show how the basic clustering approach is not able to match the tail of the distribution, even as k increases. Fig. 9(c) shows that we can approximate the tail of the popularity distribution by modeling the top 1478 files using the full model (described in Section 2) instead of the clustered model (described in Section 3). Our simulation results, described in Section 4, show that the current heuristic provides a good approximation of the real workload.

Finally, in Fig. 10, we present a visual confirmation of how we match the interarrivals, span, and skewness (of the per-object interarrival distributions). The results shown are for the case of $k = 400$. As expected, the distributions closely match the real ones.

In addition to the detailed validation performed with the HDFS trace, we used two other traces to validate our model: ANIM and MULTIMEDIA. ANIM is a 24-hour NFS trace from a feature animation company supporting rendering objects, obtained in 2007 (Set 0, available for download at) [21]. MULTIMEDIA is a one-month trace generated using the Medisyn streaming media service workload generator from HP Labs [5].[8] These experiments also produced a close approximation of the popularity distribution: $D = 0.0263$ (ANIM, $k = 500$) and $D = 0.0134$ ($k = 140$).

Running time. On our test system with 12 GB RAM and 2.33 GHz per core (8 cores) running Java code, the synthetic trace generator takes an average of 113 s (std. dev. = 22.01) to generate an unsorted trace (without tail fitting) and an average of 14 min with tail fitting. Note that this implementation can be further improved by using threads so that all the cores are utilized. The current implementation uses only two threads: one for introducing new files and one for generating the accesses to the files. The step of sorting the subtraces with a merge sort takes 6.5 min in our testing environment.

[8] We used the default configuration shipped with Medisyn, with the following changes to generate a large trace: PopularityMaxFreq = 50 000, PopularityNumOfFiles = 1 000 000, PopularityZipfAlpha = 1.61, LifespanclassTrivialClassDistParamUnifb = 30.

3.6. Choosing k

Our results show that increasing the number of clusters, k, leads to a better approximation of the popularity distribution. Our current approach to choose k is as follows:

Step 1: Find a small k for which k-means yields good clustering results. The quality of the clustering can be evaluated using standard techniques, such as cluster silhouette widths or the Calinski and Harabasz index (both of which are included with the fpc package used in our implementation [20]).

Step 2: Use increasingly larger values of k until we find a clustering that is good for the purpose of synthetic workload generation. To determine whether the clustering is useful, we use Pearson's correlation coefficient between the span and the access count of the objects in the cluster. Based on our experimental results, we use the following heuristic: an average correlation value of 0.8 or higher is good for workload generation.

4. Evaluation

In this section, we use two case studies to evaluate how well the synthetic workloads produced by our model can be used in place of the real workload. We first present a case study in Web caching, because it is a well understood and studied problem. Both the file popularity and the short-term temporal correlations have an impact on the performance of a Web cache. We then present a case study from the Big Data domain: an analysis of unbalanced accesses in a replicated distributed storage system supporting MapReduce jobs.

4.1. Case study: Web caching

In this section, we compare the cache miss rates of a server-side Web cache. We use trace-based simulations with the following 24-hour traces:

- *Real*: The original trace, which we want to approximate as closely as possible.
- *Independent Reference Model (IRM)*: Assumes that object references are statistically independent. We obtained this trace by creating a random permutation of the real trace. It is equivalent to a sampling from the popularity distribution.
- *H-IRM*: Obtained by permutating the requests within hourly chunks. Active span of a file can be off by at most one hour (vs 24 h in IRM case).
- *Interarrivals*: Obtained by sampling from a global interarrival distribution, modeled after the per-file interarrivals of the real trace. This is akin to using our model when $k = 1$.
- *Synthetic*: Obtained using the process described in Fig. 5, $k = 400$; the tail was not fitted.
- *Synthetic TF*: Generated using $k = 350$ and the tail-fitting technique.

The accesses and popularity come from the trace studied earlier in this paper (Section 3.1). File sizes are based on the January 2013 page view statistics for Wikimedia projects [22], and matched to the trace based on popularity (i.e., any correlation between size and popularity was preserved). The file sizes are used only to compute the byte miss rate of the cache (not the file miss rates).

We used the simulator developed by P. Cao [23] to simulate a server-side Web cache and evaluated it with the following cache replacement policies: *Least recently used (LRU)*: Evicts the file whose access was furthest in the past; *Size*: Evicts the largest file; and *Lowest relative value (LRV)*: Calculates the utility of keeping a document in the cache using locality, cost, and size information; evicts the document with the lowest value.

Fig. 11 shows that the results obtained using our synthetic trace are almost indistinguishable from the miss rates obtained using the real trace. We quantify the difference or error between the synthetic and expected (real) results using the root mean squared error (RMSE), shown in Table 1. Note that the results obtained using the *Synthetic TF* trace are not plotted in Fig. 11, because they are very close to the *Synthetic* results, and including them in the graph made it hard to differentiate the lines; the corresponding RMSE is shown in Table 1. The small difference between the results of the *Synthetic* and *Synthetic TF* workloads shows that this experiment is not too sensitive to the tail of popular objects.

Our results show that using our synthetic trace yields results comparable to those of the real trace. Furthermore, the low RMSE achieved for all caching strategies outperformed other alternatives. For one experiment (Size eviction), the results using *H-IRM* led to an RMSE smaller than the one resulting from our synthetic trace. However, that same trace (*H-IRM*) performed poorly for the LRU and LRV eviction policies.

Note that just reproducing per-object interarrivals (*Interarrival* trace) led to significantly better results than the independent reference model did (i.e., reproducing only object popularity). As we explained earlier, the *Interarrival* trace was obtained with our model when $k = 1$, and produced a popularity distribution significantly different from the real distribution (see Fig. 9(a)). Clearly, concentrating on reproducing the popularity distribution of object request streams while leaving the per-object interarrival distribution as a lower priority is not acceptable if we want to use synthetically generated workloads in place of real ones.

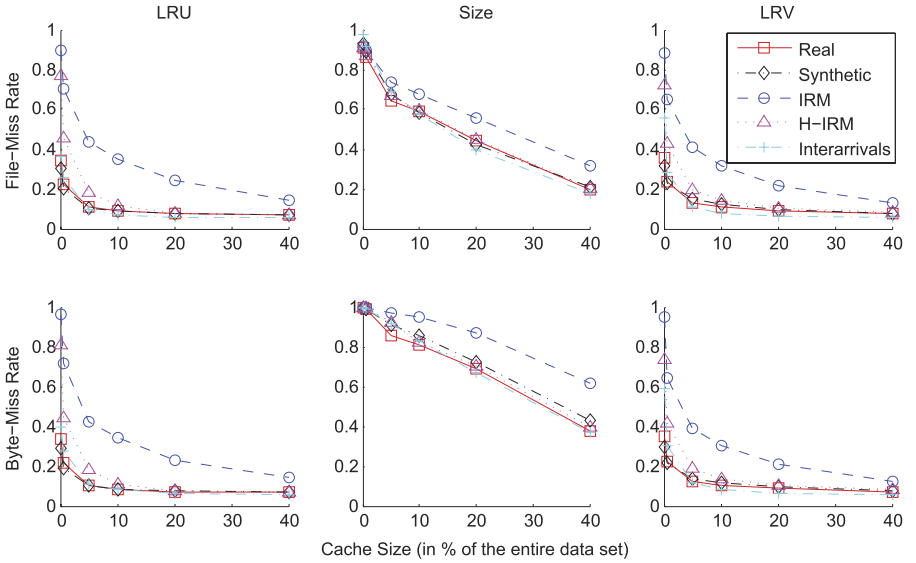


Fig. 11. File and byte cache miss rates for different cache replacement strategies. The synthetic trace generated with our model yields results that deviate very little from the ones using the real trace.

Table 1
Root mean squared error (RMSE) for Fig. 11. The synthetic traces generated by our model (labeled *Synthetic* and *Synthetic TF*) produce a very small RMSE, thus approximating the real results well. Bold values indicate results very close to the real workload (RMSE < 0.05).

		Synthetic	Synthetic TF	IRM	H-IRM	Interarrival
LRU	File miss rate	**0.018**	**0.019**	0.352	0.202	**0.021**
	Byte miss rate	**0.021**	**0.018**	0.375	0.216	**0.037**
Size	File miss rate	**0.023**	**0.041**	0.086	**0.013**	**0.045**
	Byte miss rate	**0.039**	**0.035**	0.145	**0.030**	**0.027**
LRV	File miss rate	**0.018**	**0.019**	0.312	0.172	0.085
	Byte miss rate	**0.022**	**0.020**	0.335	0.180	0.106

4.2. Case study: Replicated storage system

We present a case study on the type of replicated clustered storage systems that are commonly used in MapReduce clusters (e.g., the Google File System [24] or the Hadoop Distributed File System (HDFS) [13]). In this section, we will use specific details related to HDFS, but the general design is present in several systems supporting Big Data applications.

HDFS was designed to run on commodity machines (datanodes), whose disks may fail frequently. To prevent data unavailability due to hardware failures, these clusters replicate each block of data across several datanodes. Files are divided into fixed-size blocks (of 128 MB by default), and each block is replicated thrice by default. The block sizes and replication factors are configurable per file.

HDFS's default replica placement is as follows. The first replica of a block goes to the node writing the data; the second, to a random node in the same rack; and the last, to a random node. This design provides a good balance between being insensitive to correlated failures (e.g., whole rack failure) and minimizing the inter-rack data transmission. A block is read from the closest node: node local, or rack local, or remote.

However, this replica placement policy, combined with the fact that the files have nonuniform popularity, can lead to hotspots, and some nodes can become overwhelmed by data read requests. In another project [25], we are exploring the replica number computation problem and have implemented an HDFS replica placement simulator for that purpose. We use that simulator in this section.

Our trace-based simulator currently implements the default replica placement policy described above. Our goal in this case study is to see if our synthetic traces produce the same unbalanced accesses across nodes as the real trace. We keep track of the number of requests that each node in the cluster receives during the simulation, and use the coefficient of variation ($c_v = \sigma/\mu$, often expressed as a percentage) as a measure of dispersion to quantify the imbalance in the per-node requests. A low coefficient of variation means that the per-node requests are balanced across the nodes in the cluster.

We ran experiments using the *Real*, *IRM*, *Interarrivals*, *Synthetic*, and *Synthetic TF* traces, with one change: we made the simplification of using single-block files. From our traces, we know that 90% of the files have only one block, and we found no correlation between file size and popularity.

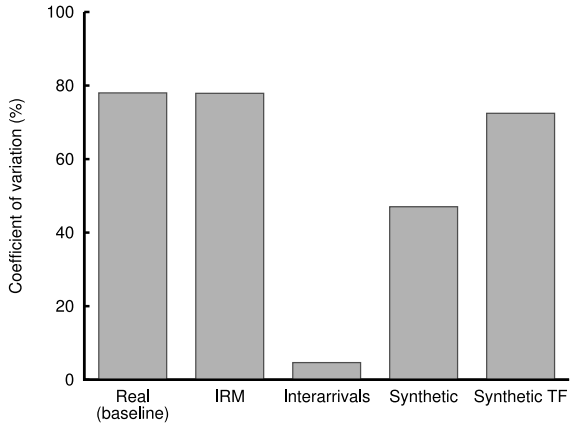


Fig. 12. Coefficient of variation for the number of requests received per node, averaged across 5 runs. Standard error not shown because values are very small (between 0.1 and 0.25% points for all traces). The *Synthetic TF* trace produces an approximation within 5.5% points of the real results.

Note that HDFS deployments run a daily rebalancing process that redistributes the blocks between the nodes to balance the used storage space. The traces used in this experiment were 24-hour traces, so we make the assumption that blocks do not migrate between nodes during the simulation. Additionally, nodes do not fail during the simulation. The simulated cluster has 4096 nodes.

Fig. 12 shows the results of our experiments, averaged across 5 runs. This experiment is sensitive to popularity and not temporal correlations; thus, *IRM* is able to replicate the results of the *Real* trace, while the *Interarrivals* trace differs significantly from the *Real* results. The *Synthetic* trace produced a better approximation; however, the results still differ significantly from the *Real* ones due to the limitations in reproducing the tail of the popularity distribution. The synthetic trace with the tail-fitting technique produces a closer approximation (difference of 5.5% points), thus showing that it can be used to assess the large imbalance in the number of requests received per node.

5. Discussion

In this section, we discuss some benefits and limitations of our model.

5.1. Non-stationary workloads

In this paper, we made no attempt to capture workloads with time-varying behavior. Object request streams typically exhibit non-stationary workloads [2,7]. The implicit assumption in our model is that the workload in the trace is stationary during the length of the trace. Our model can thus be used to capture the behavior of sample periods in a longer and non-stationary workload (e.g., during busiest periods [7]). Alternatively, it could be combined with the model proposed by Tarasov et al. [2], which deals with non-stationary workloads by dividing a time-varying workload into smaller stationary subtraces and extracts the statistical parameters of each subtrace, applying compression techniques to reduce the model size.

5.2. Object type-awareness

Our approach of using clustering to group objects according to their statistical behavior naturally leads to type-aware synthetic workload generation and analysis. Furthermore, these "types" of objects are identified without human intervention. Users do not need previous domain knowledge in order to identify the object types, nor can the process be biased by preexisting misconceptions about the workload.

To illustrate this issue, we provide the results of the clustering of the files in our trace into 7 clusters. Table 2 shows our results. For each cluster, we show the number of files in that cluster, the average span, access count, average interarrival time, skewness (of the per-file interarrival distributions), and a brief description of the "type" of files in the cluster, including real examples of files that belong to those clusters. We can observe that each cluster's averages are clearly distinct from the averages of the other clusters.

Most of the files (69%) belonged to cluster 6. They were very short lived[9] and unpopular, and had around one access per minute. The skewness of these files was the closest to zero, which means that the per-file interarrival distribution is uniformly distributed. A common type of file observed was MapReduce output files; which is to be expected to be the most common type of file in a MapReduce cluster.

[9] In the context of this paper, the life of a file is determined by its span, not by its actual creation and deletion stamps.

Table 2
Clustering results when $k = 7$. Each cluster exhibits distinct characteristics. Bold labels indicate the features used during the clustering process. Time units are in seconds. The values for Span, Access count, Average Interarrival Time (AIT) and Skew are averages for all objects in cluster.

Cluster	# files	Span	Access count	*AIT*	*Skew*	Notes and sample files
1	197 378	488.31	84.26	8.16	6.54	Short-lived, popular, frequently accessed, skewed Sample: 5-min data feeds; MapReduce job.jar
2	548 739	8975.14	9.74	875.77	0.96	Unpopular, rarely accessed Sample: Newly loaded data files
3	12 725	6736.17	1645.84	4.36	26.12	Very popular, frequently accessed, highly skewed Sample: Data schemas; Apache Oozie libraries
4	517 853	4147.81	19.85	234.65	3.08	Median Sample: MapReduce job tokens
5	27 783	28 606.10	2.66	10 863.38	0.21	Long-lived, very unpopular, very rarely accessed Sample: MapReduce job instance configuration
6	2 968 676	182.64	2.58	59.92	0.07	Most common, short-lived, unpopular, unskewed Sample: MapReduce output
7	53 649	26 861.18	5.60	5190.79	0.98	Long-lived, unpopular, rarely accessed Sample: Archived data

Cluster 3 had the fewest files, but these files were very popular and frequently accessed, with a highly skewed interarrival distribution. Files commonly observed in this cluster included data schemas and Apache Oozie libraries.[10]

In the second smallest cluster (cluster 5), the files were very long-lived and very unpopular. Each had fewer than 3 accesses during its long lifetime (28 606 s, on average). MapReduce job instance configuration files were commonly observed in this cluster. Each file was accessed at the beginning of the job and again later on, to archive them or generate cluster-wide job statistics.

The clusters were all quite different; this suggests that the types of files are limited, though the original trace is huge. If files are properly clustered, we can reduce the model space without losing much of the accuracy of the per-file characteristics.

5.3. What-if analysis and scaling

The object type-awareness facilitates "what-if" analysis. By scaling up or down the workload of one type of object (cluster) while leaving the other clusters unaltered, we could experimentally find the answers to questions like "How would the performance of my load balancer be affected if the number of short-lived files were to increase in proportion to the other types of files?"

For a more concrete example, consider the workload studied in this paper: a storage cluster (HDFS) supporting MapReduce jobs. Consider a company *X* that is currently evaluating the possibility of changing its policies regarding the recommended number of reduce tasks per job, so that jobs will now, on average, have more reduce tasks and can process more data in parallel. However, the new policy would lead to an increase in the number of MapReduce output files (since each reducer task writes its output to an individual file). From Table 2, we know that these files typically belong to cluster 6, which constitutes 69% of the files requested at least once during the 1-day trace. While increasing the per-job number of reducers could lead to jobs finishing faster, it could potentially slow things down if the namespace metadata server (called the namenode in HDFS) were to become overloaded with metadata requests. A trace-based simulation or real experiment could be performed using a modified type-aware workload to answer that question.

5.4. Other dimensions of request streams

Our current model does not include other dimensions of request streams that may be domain-specific, such as file size, session duration, and file encoding characteristics. We believe our model can be extended to include some of those dimensions.

As a simple example, consider the issue of file size. Previous studies have found file sizes were uncorrelated with access patterns [5], whereas other workloads exhibit an inverse correlation between size and popularity [26]. For the case of uncorrelated size and popularity, including the size dimension is as simple as randomly sampling from the size distribution to assign a size to a file. For the case when a correlation is found, we believe the feature can be added as one more dimension to the clustering step, so that natural object types are found and their behavior reproduced. Alternatively, the problem can be treated as a matching problem and solved using existing approaches [1]. Exploration of this idea is left for future work.

5.5. Real-time workload generation

To validate our model, we implemented a synthetic trace generator and used the traces to do trace-based simulations. In addition to simulation-based experiments, it may be desirable to do testing on actual systems. A possible way to do this is

[10] Apache Oozie is a workflow manager for MapReduce jobs.

to replay a synthetic trace on a real system by having a client machine issue the requests in the trace at the time indicated in the trace.

However, replaying large synthetic traces in real time is not trivial. It is possible that a single client computer cannot issue the requests in the trace fast enough, and not stress the system in the desired way.

Our algorithm can be adapted to do real-time workload generation by dividing the task of generating the workload (and issuing the corresponding object requests) between multiple clients, and assigning a set of O_is to each client. A workload generating client can have one thread per each renewal process that it is in charge of. We are working on implementing a synthetic workload generator for the Hadoop Distributed File System (HDFS) using this approach.

6. Related work

Several synthetic workload generators have been proposed for Web request streams [1,6]. SURGE [1] generates references to Web documents, matching empirical references of popularity and temporal locality, while reproducing burstiness using an ON–OFF process. The temporal locality of requests is modeled using a *stack distance* model of references, which assumes that each file is introduced at the start of the trace. That approach is suitable for static file populations, but inadequate for populations with high file churn [5]. Our approach considers files with delayed introduction.

ProWGen [6] was developed to enable investigation of the sensitivity of Web proxy cache replacement policies to three workload characteristics: the slope of the Zipf-like document popularity distribution, the degree of temporal locality in the document request stream, and the correlation (if any) between document size and popularity. Instead of attempting to accurately reproduce real workloads, ProWGen's goal is to allow the generation of workloads that differ in one chosen characteristic at a time, thus enabling sensitivity analysis of the differing characteristics. Additionally, through domain knowledge on Web request streams, the authors note that a commonly observed workload is that of "one-timers", or files accessed only once in the request stream. One-timers are singled out as a special type of file whose numbers the user should be able to increase or decrease in relation to other files. In contrast, we were able to approximate the percentage of one-timers in the HDFS workload without explicitly modeling them (9.79% of the real workload vs. 10.69% of our synthetic trace, when $k = 400$).

GISMO [4] and MediSyn [5] model and reproduce media server sessions, including their arrival patterns and per-session characteristics. For session arrivals, both generators have the primary goal of reproducing the file popularity, and distributing the accesses throughout the day based on observed diurnal or seasonal patterns (e.g., percentage of accesses to a file that occur during a specific time slot). Additionally, MediSyn [5] uses a file introduction process to model accesses to new files, and explicitly considers two types of files that differ in their access patterns: regular files and news-like files. Our work allows the synthetic workload generation of objects with different types of behavior without prior domain knowledge.

In earlier work, we developed Mimesis [3], a synthetic workload generator for namespace metadata traces. While Mimesis is able to generate traces that mimic the original workload with respect to the statistical parameters included with it (arrivals, file creations and deletions, and age at time of access), reproducing the file popularity was left for future work.

Chen et al. [27] proposed the use of multi-dimensional statistical correlation (k-means) to obtain storage system access patterns and design insights in user, application, file, and directory levels. However, the clustering was not leveraged for synthetic workload generation.

Hong et al. [28] used clustering to identify representative trace segments to be used for synthetic trace reconstruction, thus achieving trace compression ratios of 75% to 90%. However, the process of fitting trace segments, instead of individual files based on their behavior, does not facilitate deeper understanding of the behavior of the objects in the workload, nor does it enable what-if or sensitivity analysis.

Ware et al. [7] proposed the use of two-level arrival processes to model bursty accesses in file system workloads. In their implementation, objects are files, and accesses are any system calls issued on that file (e.g., read, write, lookup, or create). Their model uses three independent per-file distributions: interarrivals to bursts of accesses, intra-burst interarrival times, and distribution of burst lengths. A two-level synthetic generation process (in which burst arrivals are the first level, and intra-burst accesses to an object are the second level) is used to reproduce bursts of accesses to a single file. However, the authors do not distinguish between the access to the first burst and the accesses to subsequent bursts, so their model cannot capture file churn. Additionally, the authors use one-dimensional hierarchical clustering to identify bursts of accesses in a trace of per-file accesses. The trace generation process is similar to ours: one arrival process per file. However, the size of the systems they modeled (top $\sim$567 files out 8000 total) did not require a mechanism to reduce the model size. We are considering systems two orders of magnitude larger, so a mechanism to reduce the model size was necessary. The approach of modeling intra-burst arrivals independently of inter-burst arrivals can be combined with our delayed first arrival plus clustering of similar objects approach to capture per-file burstiness.

7. Conclusions

We presented a model for analyzing and synthetically generating object request streams. The model is based on a set of delayed renewal processes, where each process represents one object in the original request stream. Each process in the model has its own request interarrival distribution, which combined with the time of the first access to the object plus

the period during which requests to the object are issued, can be used to approximate the number of arrivals or renewals observed in the original trace.

We also proposed a lightweight version of the model that uses unsupervised statistical clustering to significantly reduce the number of interarrival distributions that we need to keep track of, thus making the model suitable for synthetic trace generation.

We showed how our model is able to produce synthetic traces that approximate the original interarrival, popularity and span distributions within 2% of the original CDFs. Through two case studies, we showed that the synthetic traces generated by our model can be used in place of the original workload and produce results that approximate the expected (real) results.

Our model is suitable for request streams with a large number of objects and a dynamic object population. Furthermore, the statistical clustering enables autonomic type-aware trace generation, thus facilitating sensitivity and "what-if" analysis.

Acknowledgments

Part of this work was completed during CA's internship at Yahoo!. RC and CA are supported in part by AFRL grant FA8750-11-2-0084. YL and MY are partially supported by NSF grant CNS-1150080.

Appendix. Background: Renewal processes

A *renewal process* is a stochastic model for events – referred to as *renewals* or *arrivals* – that occur randomly in time. It is a generalization of a Poisson process with arbitrary interarrival times. This model gives rise to several interrelated random processes, including: the sequence of interarrival times $X = (X_1, X_2, \ldots)$, the sequence of arrival times $T = (T_1, T_2, \ldots)$, and a counting process $N = (N_t : t \geq 0)$. X is a sequence of independent, identically distributed random variables. T is the partial sum process associated with the independent, identically distributed sequence of interarrival times X. F is the common distribution function of the interarrival times; $\mu = \mathbf{E}(X)$ denotes the common mean of the interarrival times [29].

The random counting measure is completely determined by the counting process. Furthermore, the arrival process T and the counting process N are inverses of one another, in a sense.

The expected number of arrivals up to time t is known as the *renewal function*, denoted as $m(t)$, which characterizes the renewal process:

$$m(t) = E(N_t), \quad t \in [0, \infty) \tag{1}$$

$$m(t) = \sum_{n=1}^{\infty} F_n(t), \quad t \in [0, \infty) \tag{2}$$

$$\text{where } F_n(t) = P(T_n \leq t), \quad t \in [0, \infty). \tag{3}$$

Asymptotically, the following equations hold:

$$\text{If } \mu < \infty \text{ then } N_t/t \to 1/\mu \text{ as } t \to \infty \text{ with prob. 1} \tag{4}$$

$$m(t)/t \to 1/\mu \text{ as } t \to \infty \tag{5}$$

$$m(t, t+h] \to h/\mu \text{ as } t \to \infty. \tag{6}$$

Eq. (5) is called the elementary renewal theorem. Eq. (6) is called the renewal theorem.

A *delayed renewal process* is a renewal process in which the first arrival time can have a different distribution than the other interarrival times. G denotes the distribution function of X_1. Since only the first arrival time is changed, the asymptotic behavior of a delayed renewal process is the same as the asymptotic behavior of the corresponding regular renewal process.

References

[1] P. Barford, M. Crovella, Generating representative Web workloads for network and server performance evaluation, SIGMETRICS Perform. Eval. Rev. 26 (1) (1998).
[2] V. Tarasov, K. Kumar, J. Ma, D. Hildebrand, A. Povzner, G. Kuenning, E. Zadok, Extracting flexible, replayable models from large block traces, in: Proc. USENIX Conf. File and Storage Tech. FAST, 2012.
[3] C. Abad, H. Luu, N. Roberts, K. Lee, Y. Lu, R. Campbell, Metadata traces and workload models for evaluating Big storage systems, in: Proc. IEEE/ACM Utility and Cloud Comp. Conf., UCC, 2012.
[4] S. Jin, A. Bestavros, GISMO: a generator of Internet streaming media objects and workloads, SIGMETRICS Perform. Eval. Rev. 29 (3) (2001).
[5] W. Tang, Y. Fu, L. Cherkasova, A. Vahdat, MediSyn: a synthetic streaming media service workload generator, in: Proc. ACM Netw. Oper. Sys. Support for Digital Audio and Video, NOSSDAV, 2003.
[6] M. Busari, C. Williamson, ProWGen: a synthetic workload generation tool for simulation evaluation of Web proxy caches, Comput. Netw. 38 (6) (2002).
[7] P. Ware, T. Page Jr., B. Nelson, Automatic modeling of file system workloads using two-level arrival processes, ACM Trans. Model. Comput. Simul. 8 (3) (1998).
[8] R. Fonseca, V. Almeida, M. Crovella, B. Abrahao, On the intrinsic locality properties of Web reference streams, in: Proc. IEEE Intl. Conf. Comp. Comm., INFOCOM, 2003.
[9] B. Fan, W. Tantisiriroj, L. Xiao, G. Gibson, DiskReduce: RAID for data-intensive scalable computing, in: Proc. Petascale Data Storage Wkshp., PDSW, 2009, pp. 6–10.
[10] C. Abad, N. Roberts, Y. Lu, R. Campbell, A storage-centric analysis of MapReduce workloads: file popularity, temporal locality and arrival patterns, in: Proc. IEEE Intl. Symp. Workload Characterization, IISWC, 2012.

[11] Y. Chen, S. Alspaugh, R. Katz, Interactive query processing in Big Data systems: a cross-industry study of MapReduce workloads, in: Proc. Intl. Conf. Very Large Data Bases, VLDB, 2012.
[12] B. Cooper, A. Silberstein, E. Tam, R. Ramakrishnan, R. Sears, Benchmarking cloud serving systems with YCSB, in: Proc. ACM Symp. Cloud Comp., SoCC, 2010.
[13] K. Shvachko, H. Kuang, S. Radia, R. Chansler, The Hadoop distributed file system, in: Proc. IEEE Symp. Mass Storage Syst. and Tech., MSST, 2010.
[14] S. Jin, A. Bestavros, Sources and characteristics of Web temporal locality, in: Proc. IEEE Intl. Symp. Modeling, Analysis and Sim. Comp. Telecomm. Sys., MASCOTS, 2000.
[15] J. Dean, S. Ghemawat, MapReduce: Simplified data processing on large clusters, in: Proc. USENIX Symp. Op. Sys. Design and Impl., OSDI, 2004, pp. 137–150.
[16] J. MacQueen, Some methods for classification and analysis of multivariate observations, in: Proc. 5th Berkeley Symp. Mathematical Statistics and Probability, Vol. 1, 1967, pp. 281–297.
[17] L. Breslau, P. Cao, L. Fan, G. Phillips, S. Shenker, Web caching and Zipf-like distributions: evidence and implications, in: Proc. IEEE Intl. Conf. Comp. Comm., INFOCOM, 1999.
[18] L. Cherkasova, M. Gupta, Analysis of enterprise media server workloads: access patterns, locality, content evolution, and rates of change, IEEE/ACM Trans. Netw. 12 (5) (2004).
[19] G. Dán, N. Carlsson, Power-law revisited: large scale measurement study of P2P content popularity, in: Proc. Intl. Wkshp. Peer-to-Peer Sys., IPTPS, 2010.
[20] C. Hennig, fpc: flexible procedures for clustering, January 2013 [Online]. Available: http://cran.r-project.org/web/packages/fpc (last accessed 23.03.13).
[21] E. Anderson, Capture, conversion, and analysis of an intense NFS workload, in: Proc. USENIX Conf. File and Storage Tech., FAST, 2009.
[22] D. Mituzas, Page view statistics for Wikimedia projects, March 2013 [Online]. Available: http://dumps.wikimedia.org/other/pagecounts-raw/ (last accessed 04.08.2013).
[23] P. Cao, S. Irani, Cost-aware WWW proxy caching algorithms, in: Proc. Usenix Symp. Internet Tech. and Sys., USITS, 1997.
[24] S. Ghemawat, H. Gobioff, S. Leung, The google file system, in: Proc. ACM Symp. Oper. Sys. Principles, SOSP, 2003.
[25] C. Cai, C. Abad, R. Campbell, Storage-efficient data replica number computation for multi-level priority data in distributed storage systems, in: Proc. Wkshp. Reliability and Security Data Analysis, RSDA, co-located with DSN, 2013.
[26] C. Cunha, A. Bestavros, M. Crovella, Characteristics of WWW client-based traces, Tech. Rep. BU-CS-95-010, Boston University, 1995.
[27] Y. Chen, K. Srinivasan, G. Goodson, R. Katz, Design implications for enterprise storage systems via multi-dimensional trace analysis, in: Proc. ACM Symp. Oper. Sys. Principles, SOSP, 2011.
[28] B. Hong, T. Madhyastha, B. Zhang, Cluster-based input/output trace synthesis, in: Proc. IEEE Intl. Perf. Comp. and Comm. Conf., IPCCC, 2005.
[29] K. Siegrist, Virtual laboratories in probability and statistics, 2013, ch. Renewal Processes [Online]. Available: www.math.uah.edu/stat/ (last accessed 23.03.13).

Cristina L. Abad is a Computer Science Ph.D. Candidate at the University of Illinois at Urbana-Champaign, and currently on leave from her position as an Assistant Professor at ESPOL in Guayaquil, Ecuador. She received her Master of Science degree from UIUC in 2003, funded through a Fulbright Fellowship. Her main research interests lie in the area of distributed systems, in particular clustered storage systems and Big Data applications.

Mindi Yuan received his Bachelor Degree from the Department of Electronic Science and Engineering, Nanjing University, in 2010. Now he is a Ph.D. candidate at the Department of Electrical and Computer Engineering, University of Illinois at Urbana-Champaign. His research area is networking and distributed computing.

Chris X. Cai received B.A. degrees in computer science and economics from the University of California, Berkeley. He is currently a Ph.D. student of the Computer Science Department at the University of Illinois, Urbana-Champaign. His research interests include cloud computing, distributed system and networking.

Yi Lu is an assistant professor of Electrical and Computer Engineering at the University of Illinois at Urbana-Champaign. She received her doctorate from the Electrical Engineering Department at Stanford University where she is a recipient of the Stanford Graduate Fellowship. She has received the Sigmetrics Best paper award in 2008, the Performance Best paper award in 2011, and the NSF Career award in 2012. Her work focuses on developing efficient architectures and algorithms for large and complex networks such as clouds and social networks. Her work spans fundamental analysis and algorithm implementation, and emphasizes the design of low-complexity implementable algorithms.

Nathan Roberts received a BS in computer science from the University of Illinois at Urbana-Champaign in 1990. He is a Senior Software Architect with the Hadoop Core Team at Yahoo!. Prior to working on Hadoop, Nathan focused on high performance object storage and Linux kernel internals. Prior to Yahoo!, Nathan was with Motorola for 17 years working on operating system internals for platforms ranging from mobile phones to carrier-grade fault tolerant servers.

Roy H. Campbell is the Sohaib and Sara Abbasi Professor in the Department of Computer Science, Director of Graduate Programs, Director of the NSA Designated Center for Academic Excellence and Research in Information Assurance, Director of the Air Force funded Assured Cloud Computing Center at the Information Trust Institute, and 2013–2014 Chair of the University Senate. He received his Honors B.S. Degree in mathematics, with a Minor in physics from the University of Sussex in 1969 and his MS and Ph.D. degrees in computer science from the University of Newcastle upon Tyne in 1972 and 1976, respectively. Professor Campbell's research interests are the problems, engineering and construction techniques of complex system software. Cloud computing, big data, security, distributed systems, continuous media, and real-time control pose system challenges, especially to operating system designers. He is a Fellow of the IEEE.