JEDI: Model-driven Trace Generation for Cache Simulations

Anirudh Sabnis
Univ. of Massachusetts Amherst

Ramesh K. Sitaraman
Univ. of Massachusetts Amherst & Akamai Technologies

Abstract

A major obstacle for caching research is the increasing difficulty of obtaining original traces from production caching systems. Original traces are voluminous and also may contain private and proprietary information, and hence not generally made available to the public. The lack of original traces hampers our ability to evaluate new cache designs and provides the rationale for JEDI, our new synthetic trace generation tool. JEDI generates a synthetic trace that is "similar" to the original trace collected from a production cache, in particular, the two traces have similar object-level properties and produce similar hit rates in a cache simulation. JEDI uses a novel traffic model called Popularity-Size Footprint Descriptor (pFD) that concisely captures key properties of the original trace and uses the pFD to generate the synthetic trace. We show that the synthetic traces produced by JEDI can be used to accurately simulate a wide range of cache admission and eviction algorithms and the hit rates obtained from these simulations correspond closely to those obtained from simulations that use the original traces. JEDI will be provided to the public as open-source, along with a library of pFD's computed from traffic classes hosted on Akamai's production CDN. This will allow researchers to produce realistic synthetic traces for their own caching research.

ACM Reference Format:
Anirudh Sabnis and Ramesh K. Sitaraman. 2022. JEDI: Model-driven Trace Generation for Cache Simulations. In *Proceedings of the 22nd ACM Internet Measurement Conference (IMC '22), October 25–27, 2022, Nice, France.* ACM, New York, NY, USA, 15 pages. https://doi.org/10.1145/3517745.3561466

1 Introduction

Billions of users access the Internet daily to download many forms of content, including videos, documents, images, web pages, and software. Content caching is ubiquitous and critical to the functioning of the Internet. Most of the content accessed by users are delivered by content delivery networks (or, CDNs) [24, 47] consisting of hundreds of thousands of caches deployed in thousands of datacenters around the world. CDNs enable fast and reliable access to content by caching it in servers proximal to users.

Cache hits and misses. When a user accesses a web page or a video, a request is sent to a proximal CDN server that serves the content if it is present in its cache (called a *cache hit*). If the content is not found in its cache (called a *cache miss*), the CDN server fetches that content over the WAN from an origin server and serves it to the user. The efficiency of a cache is often measured by two different metrics: the *request hit rate* (RHR) which is the fraction of requests for content that resulted in cache hit and the *byte hit rate* (BHR) which is the fraction bytes that were served from cache hits.

Caching benefits. Content caching provides both performance and cost benefits. Caching provides better performance by enabling users to download content from a proximal cache with low latency and higher reliability, allowing web pages to download faster and video streams to play with better quality. Maximizing the RHR is key to maximizing the performance benefit since a cache miss introduces large WAN latencies, resulting in users experiencing slow content downloads. Caching also provides a cost benefit since retrieving the content from a nearby cache avoids fetching that content over the WAN, resulting in a reduction in traffic between the CDN cache and the content provider's origin server, decreasing the bandwidth cost [55] incurred by the CDN for the cache miss traffic. Further, caching reduces the content provider's origin infrastructure cost of serving the cache miss traffic. Maximizing BHR is key to minimizing cost, since BHR weights each cache hit by the size of the object and the traffic (in bits per second) caused by cache misses is directly proportional to the *byte miss rate* that equals 1 - BHR.

Cache admission and eviction. A cache management system has two components designed to maximize RHR and/or BHR: the *cache admission algorithm* decides what objects are admitted into cache and the *cache eviction algorithm* decides what objects will be evicted from cache when it is full. Both cache admission and eviction algorithms have been topics of intense research over the past decades [7, 17, 32, 33, 35, 44, 61]. Caching continues to be a focus for innovative research both within academia and industry as the characteristics of the content and the manner in which it is accessed becomes more diverse and complex.

Content features used in caching. The algorithms that implement cache management policies typically rely on *features* of the objects that are being requested by users to make their decision. Traditionally, cache admission algorithms have used *popularity* or *size* of the object to decide whether or not to admit an object into cache. For instance, a common popularity-based approach implemented by production CDNs such as Akamai is to an admit an object into the disk cache only after it is request k times, typically $k = 2$ and a bloom filter is used to detect the second access of an object [41]. For smaller caches, such as the hot-object cache in memory, algorithms such as AdaptSize [15] use a size threshold and allow only objects smaller than the threshold to be admitted to cache. The use of a size threshold is common for smaller caches since admitting a large object into cache could result in the eviction of numerous smaller objects. In contrast, cache eviction policies have commonly relied on features related to recency or frequency of access [7, 23, 29, 44, 61], evicting those objects that are less recently or less frequently accessed.

Challenges obtaining original traces. Caching research is increasingly important as newer content types and access patterns challenge the effectiveness of existing caching algorithms. However,

IMC '22, October 25–27, 2022, Nice, France

ACM ISBN 978-1-4503-9259-4/22/10…$15.00
https://doi.org/10.1145/3517745.3561466

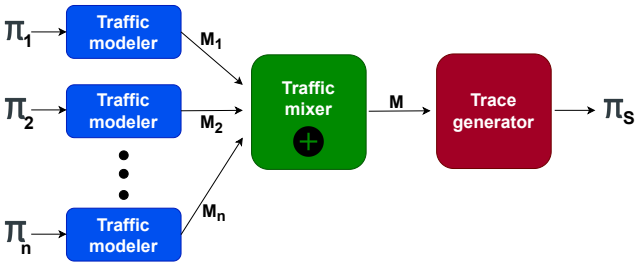


Figure 1: System architecture of JEDI.

a major obstacle for research progress is the increasing difficulty of obtaining *original traces* from production CDN caches. An original trace is a sequence of records where each record corresponds to a request for an object made by an user. Each record contains at least the id (such as url) of the requested object, the size of that object, and the time when it was accessed. Thus, the original trace has all the information needed for performing realistic cache simulations. The reason for the dearth of original traces are manifold. First, the original traces are voluminous, making it hard to collect and distribute. Second, the original traces often have private or proprietary information, making it difficult to share across enterprise boundaries and to provide publicly for researchers. Finally, even if original traces can be obtained, they represent only a few of the possible scenarios that may occur in the field, making it harder to investigate what-if questions that go beyond those scenarios. These challenges motivate our research on synthetic trace generation.

Traffic classes and composability. Modern CDNs serve highly-diverse content that vary in type, size, and access patterns. For ease of traffic management, it is customary to partition the content served by a CDN into *traffic classes* [54] where each class corresponds to a specific content type of a specific content provider, example, images from Facebook, software downloads from Microsoft, and video segments from Netflix each form a traffic class. A large CDN may serve several hundred traffic classes overall and each cache may serve time-varying mix of multiple traffic classes.

Goal of synthetic trace generation. The goal of synthetic trace generation is to produce a synthetic trace Π_s that is "similar" to an original trace Π_o of a traffic class in the following two respects: (i) Traces Π_o and Π_s have similar object-level properties, i.e., the total variation distances [3] of the respective object-size distributions, popularity distributions, and request-size distributions are small, and (ii) Traces Π_o and Π_s have similar cache-level properties, in particular, a cache simulation that uses trace Π_s and another that uses trace Π_o must produce similar values for metrics such as RHR and BHR. In addition, given original traces for multiple traffic classes, the trace generator should be able to generate a synthetic trace corresponding to an arbitrary mix of those classes.

Our approach. We propose a model-based trace generator JEDI (c.f., Fig. 1) that works as follows. Given an original trace Π_o, a *traffic modeler* computes a model that we call Popularity-Size Footprint Descriptor (pFD) that captures key properties of the original trace. The *traffic mixer* enables pFD's of multiple individual traffic classes to be used to derive the pFD of an arbitrary mix of those classes. Finally, the *trace generator* produces a synthetic trace from the pFD of the traffic class (or, mix). Our approach is most similar to TRAGEN [50] that also uses a model-driven approach using two different models called Footprint Descriptor (FD) and Byte-weighted Footprint Descriptor (bFD). Our work is significant advance over TRAGEN as we explain below.

Our contributions. We make the following contributions.

(1) JEDI uses a novel traffic model called Popularity-Size Footprint Descriptor (pFD) that captures both the object-level properties and cache-level properties of an original trace. The important characteristics of our traffic model are: (i) pFD's are succinct and are orders of magnitude smaller than the original production traces. The pFDs computed from original traces of size 100's of GB are around 100MB and hence 1000 times smaller; (ii) pFD's are efficiently computable from an original trace in $O(N \log m)$ where N is the length of the original trace and m is the number of objects in the trace. In our implementation that uses python, a pFD can be computed in around 120 minutes for a trace that consists of 100 million requests; (iii) pFDs are composable i.e., the pFD of a traffic mix can be efficiently computed from the pFDs of individual traffic class by leveraging Fast Fourier Transforms. Our implementation, in python, takes around 30 minutes to compute the pFD of a traffic mix given individual pFDs.

(2) JEDI is the *first* tool that can produce a synthetic trace Π_s that has similar object-level properties *and* cache-level properties as that of the original trace Π_o. In particular, the total variation distance between the object-size distributions, popularity distributions and request-size distributions derived from traces Π_s and Π_o are 1.1×10^{-3}, 5.7×10^{-3}, 1.1×10^{-2} respectively. Further, for cache-level properties, the average difference in RHR (resp. BHR) for Π_S and Π_o is 1.2% (resp. 0.6%) across all cache simulations that we performed. In contrast, the state-of-the-art synthetic trace generation tool TRAGEN does not produce a synthetic trace with similar object-level properties as the original. While TRAGEN produces a synthetic trace that has a similar size distribution as the original trace, the popularity distributions and the request-size distributions of the two traces are not similar. Further, TRAGEN can only produce a synthetic trace that has either a similar RHR or BHR as the original, and not both simultaneously. JEDI meets a higher bar of similarity by *simultaneously* matching three key object-level distributions and two key cache-level metrics of the synthetic and original traces.

(3) Since JEDI produces a synthetic trace that matches the original trace in object-level properties, it can be reliably used to simulate cache admission algorithms that use object size and popularity features for decision making. In particular, we show that traces produced by JEDI provide more accurate simulations of cache admission algorithms that use object size and popularity features than TRAGEN. Likewise, the traces produced by JEDI provide more accurate simulations of cache eviction algorithms that use object size (e.g., GDSF) than TRAGEN. However, JEDI and TRAGEN perform similarly for cache eviction algorithms that rely only on recency features. In summary, synthetic traces produced by JEDI produces more accurate simulations for a broader set of cache admission and eviction algorithms than TRAGEN.

(4) JEDI is publicly available for download as an open-source[1] contribution, along with a library of pFD's computed from traffic classes hosted on Akamai's production CDN. This will allow

[1] It can be downloaded from https://github.com/UMass-LIDS/Jedi

researchers to produce realistic synthetic traces for their own caching research. Our traffic modeler component also allows other researchers to create pFD's for their own original traces.

JEDI limitations. We are able to prove that the trace generation algorithm works for an LRU cache, using theoretical models such as footprint descriptors [54] that capture the caching properties of an LRU cache. However, proving that a trace generation algorithm works for arbitrary (non-LRU) cache algorithms is hard because there are no theoretical models that capture the caching properties of such algorithms. Even though we do not prove that hit rates match for non-LRU algorithms, the synthetic trace generated by JEDI captures essential features of the original trace such as the access patterns (temporal locality), popularity distribution, and the size distribution of the objects in the trace. So, (non-LRU) cache algorithms that use these features to make cache admission and eviction decisions provide similar hit rates on the synthetic trace as the original trace, despite the lack of theoretical guarantee on hit rates. We demonstrate this fact for a wide range of cache algorithms through extensive empirical evaluations (§6).

Relation to prior work. Most prior work on synthetic trace generation propose tools that generate synthetic traces that are representative of web traffic [13, 18, 34, 36, 46]. These tools from past decades do not cater to the diverse traffic classes and traffic mix scenarios that modern content caches serve. Most importantly, these tools fail to generate a synthetic trace that satisfies the cache-level properties of an original trace as we show in §6.5. A recent tool, TRAGEN [50], partially overcomes this challenge by producing a synthetic trace that has the same cache-level properties as the original trace for cache algorithms that use recency as a feature to make eviction decisions. However, recency based cache eviction algorithms form only a subset of the cache algorithms that are used in practice. There exist many cache algorithms that use object-level properties such as popularity and size to make cache admission *and* cache eviction decisions (§10.1 and §10.2). TRAGEN fails to produce a synthetic trace with same cache-level properties as the original trace for these algorithms as it does not incorporate the object-level properties such as popularity distribution and request-size distribution. Further, TRAGEN produces two distinct synthetic traces that have either a similar RHR or a similar BHR as the original trace. Producing a *single* synthetic trace that has similar cache-level properties (RHR and BHR) *and* object-level properties (size distribution, popularity distribution and request-size distribution) as an original trace is the challenge we overcome in this work.

JEDI is the first tool that can produce a single synthetic trace with similar cache-level and object-level properties as the original trace. Thus, the synthetic traces produced by JEDI can be used in the simulation of a wide range of cache algorithms that span (A) popularity and size based admission algorithms; (B) popularity and size based eviction algorithms; and (C) recency based eviction algorithms. This work does not raise any ethical issues.

2 Traffic classes and their properties

We show how the digital content hosted on the internet can be classified into *traffic classes* (c.f., § 2.1) and describe the *object-level properties* and *cache-level properties* (§ 2.2) of a traffic class that are important for realistic trace-based simulations.

Trace	Video (V)	Web (W)	TC	EU
Length (mil. reqs)	596	6167	288	595
Req. rate (reqs/sec)	382	7414	820	382
Traffic (GBps)	1.5	2.29	0.36	1.31
No. of objects (mil.)	127	279	51	99
Avg. obj. size (KB)	1756	291	122	1268
Year collected	2018	2015	2018	2015

Table 1: Trace description

2.1 Traffic classes

There exists a wide variety of digital content on the Internet, for example, images, web pages, videos, 360° videos, software downloads, that is hosted by a multitude of content providers. For the ease of cache management, traffic is often bucketed into traffic classes. Each traffic class refers to a content type hosted by a specific content provider, for example, images from Facebook, videos on Netflix, software downloads from Microsoft. Cache provisioning decisions in a CDN are generally made at the granularity of a traffic class and a large CDN may have a few hundred traffic classes [55]. Each traffic class has unique object-level and cache-level properties.

2.1.1 Original traces for traffic classes. To illustrate the properties of the traffic classes and for other empirical work in this paper, we use the same original traces from Akamai's CDN as that used to evaluate TRAGEN [50]. We provide the details of these traces that were first described in [50] (Table 1). Each trace consists of hundreds of millions of requests that were made for millions of objects for different traffic classes such as web, video, software downloads, media. Each trace is collected over a period of few days. The WEB and VIDEO trace were collected from a CDN edge cluster that consists of 10 servers. The WEB and VIDEO trace contain requests that were made predominantly for the web and video traffic class. The TC and EU traces consists of requests made for a variety of traffic classes such as downloads, media, web and images and are described in Table 3 and Table 2, respectively.

2.2 Object-level and Cache-level Properties

The three *object-level properties* are the object size distribution, popularity distribution, and request size distribution as described below, The two *cache-level properties* of interest are RHR and BHR.

Object size distribution (SZ). The object sizes of content on the Internet vary considerably across and within traffic classes. For instance, web pages or images are generally much smaller as compared to media segments or software downloads. And within a traffic class, say media segments from Netflix, the segments with the lowest quality (bitrate) are much smaller in size as compared to segments with the highest quality. The *object size distribution* SZ(z) of a trace gives us the probability that an object in the trace is of size z. Fig. 3a depicts the object size distribution for the various traffic classes.

Why SZ matters for cache simulations? The SZ of the trace has a direct impact on the observed hit rates. For instance, if the trace consists of requests made to large media objects, then we can only store a small number of objects in cache, leading to cache misses. Further, there also exist many cache algorithms such as ThLRU,

Trace	Media-0	Media-1	Media-2	Media-3	Media-4	Media-5	Media-6	Web-7	Media-8	Web-9
Length (mil. reqs)	32.04	109.3	70.3	91.92	43.98	66.48	36.56	9.73	128.44	6.95
Req. rate (reqs/sec)	20.64	70.44	45.32	59.2	28.33	42.82	23.55	6.248	82.73	5.38
Traffic (MBps)	12	480	13	36	288.3	434.8	26.8	0.8	27.682	0.756
No. of objects (mil.)	15.55	2.66	18.62	39.64	2.31	2.49	14.45	0.028	22.56	0.02
Avg. object size (KB)	679.2	9727	286.4	653	10286	10291	1026	71.65	151.3	69.83

Table 2: EU trace description

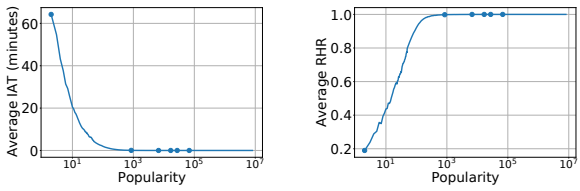
Adaptsize, GDSF (described in § 10.1) that use object size as an admission or eviction criteria. Thus, for the synthetic trace and the original trace to have the same cache-level properties across different cache algorithms, it is desirable that their SZ is similar.

Popularity distribution (POP) We define *popularity* of an object as the number of requests made for it in a given trace [13, 18]. Specifically, the *popularity distribution* POP(p) gives us the probability that an object is requested p times in the duration of the trace. Observe that popularity of an object depends on the length of the trace that is collected. If the length is doubled, the popularity of some objects in the trace could increase. However, we chose not to normalize the popularity by the trace length for the following reason. For sufficiently large traces, such as the ones we have collected and described in Table 1, we find that there exist very few objects that are requested through the span of the trace. Most objects have a short lifespan i.e., the first and last request for an object are not far apart. Thus, if the trace is sufficiently large, POP remains unchanged as its length is increased.

We find that POP varies significantly across different traffic classes (shown in Figure 3b). For the traffic class WEB (Table 1), there exist a few really popular objects that make up to 15% of the requests. The skew in POP of the VIDEO and the EU traffic class is lesser as compared to the WEB traffic class. Nonetheless, in each traffic class, we observe that there exist few objects that are really popular and there exist a large fraction of objects that are accessed only once. For instance, in the VIDEO and EU trace 55% of the objects are requested only once.

Why POP matters for cache simulations? We now analyse the access patterns of requests based on the popularity of the requested object. Figure 2a depicts the average *inter-arrival-time* i.e., the time between consecutive requests to an object based on its popularity. As expected, the inter-arrival-time of popular objects is considerably smaller as compared to the unpopular objects. Hence, popular objects exhibit a higher temporal locality. And on a subsequent request to a popular object, it is likely to be found in the cache and we would observe a cache hit. Figure 2b shows the RHR of the objects based on their popularity. Popular objects have a much higher RHR as compared to unpopular objects. Thus, the POP of a trace indirectly dictates the hitrates we observe on cache simulations. Further, there exist many cache algorithms such as LFU, LRFU, GDSF, Bloomfilter (described in § 10.1 and § 10.2) that use frequency of access as a feature to make eviction or admission decisions. Thus, it is desirable to have POP of the original trace and synthetic trace to be similar.

Request size distribution (REQSZ) The request size distribution REQSZ(z) of a trace gives us the probability that a request is for an object of size z. Note that REQSZ and SZ are not equivalent. An object o of size z and popularity p, is counted only once in the computation of SZ, but is counted p times (for each request for object o) in the computation of REQSZ.



(a) (b)

Figure 2: (a) Popularity vs Avg. inter arrival time; and (b) Popularity vs Avg. RHR

Why REQSZ matters for cache simulations? The REQSZ for the various traffic classes is depicted in Fig. 3c. The REQSZ has the following effect on the RHR and BHR of a trace during a cache simulation. Consider the following scenario. If the popular objects that experience more cache hits are smaller in size as compared to the unpopular objects, the fraction of bytes that would be served from the cache is smaller as compared to the fraction of requests that are served from the cache. Hence, we would observe a smaller BHR than RHR. We observe this phenomena in the VIDEO trace. The BHR of VIDEO trace on an LRU cache of size size 500GB is 0.26, whereas the RHR is 0.42 . Therefore, it is desirable to have REQSZ of the original trace and synthetic trace to be similar.

3 The Traffic Modeler

In this section, we define a novel traffic model called Popularity-Size Footprint Descriptor (pFD) that captures the cache-level properties and object-level properties of traffic classes and their mixes.

3.1 Traffic Model Requirements

A traffic model derived from an original trace should have the following properties.

(1) **Generative.** The traffic model should capture the object-level properties and cache-level properties of a traffic class and those properties must be derivable from the model. Further, the traffic model should serve as sufficient input for generating synthetic traces with those properties.

(2) **Succinct.** The traffic model should be orders of magnitude smaller than the original traces from which it is computed. A small traffic model is easy to store, retrieve and analyze.

(3) **Efficiently computable.** As original traces are voluminous, we should be able to compute the traffic model from the traces in a time and space efficient manner.

(4) **Shareable.** Our traffic model should be shareable across organizations and even made publicly available. Hence, they should contain only aggregate information and not contain any personal identifiable information that may be present in the original trace.

Trace	Download	Image	Media	Web
Length (mil. reqs)	8.06	85.4	49.8	144
Req. rate (reqs/sec)	22.9	243	141	406.7
Traffic (MBps)	70	8	40	250
No. of objects (mil.)	0.32	33	7.1	11.1
Avg. obj. size (KB)	603	20.5	368	255

Table 3: TC trace description

(5) **Composable.** Given traffic models of individual traffic classes, we should be able to *efficiently* compute the traffic model of any traffic mix scenario. Since caches serve a varying mix of traffic classes, composability helps model the various scenarios that could play out in a production environment.

3.2 Popularity-Size Footprint Descriptor

We now describe our traffic model – Popularity-Size Footprint Descriptor (pFD) – and show how it satisfies all five requirements. The pFD of a traffic class is computed from original traces of that traffic class obtained from the production system. An original trace Π is a sequence of requests $\{r_1, \ldots, r_n\}$, where each r_i is a tuple (t_i, o_i, z_i), where t_i is the timestamp at which the request is made, o_i is object identifier that uniquely identifies the requested object, and z_i is the size of the object. Let p_i be the popularity of the object o_i. Now $\theta = \{r_i, \ldots, r_j\}$, with $i < j$, is a *reuse request subsequence* if r_i and r_j are requests made for the same object $o_i = o_j = o$, and o is not requested elsewhere in θ.

The pFD of a trace Π is described as $\langle \lambda, P^r(p, z, s, t), P^a(s, t) \rangle$. Here, λ is the *request rate* of Π i.e., the number of requests per unit time of Π. $P^r(p, z, s, t)$ denotes the *popularity-size reuse subsequence descriptor*. It is the probability that a *reuse request subsequence* $\theta = \{r_i, \ldots, r_j\}$ of Π has the following properties, (1) requests r_i and r_j are made for the same object o and the object o is of size z and has a popularity p; (2) θ consists of s unique bytes i.e., sum of the sizes of the unique objects in θ is s; and (3) θ has an inter-arrival-time of t seconds i.e. $t = t_j - t_i$. The number of unique bytes in a reuse request subsequence is also known as *stack distance* [43]. By convention, reuse request subsequences that begin at the start of a trace and end on the first request to an object are considered to have infinite unique bytes and an infinite inter-arrival-time.

Now, $P^a(s, t)$ denotes the *all subsequence* descriptor and is the probability that *any* request subsequence of Π consists of s unique bytes and has an inter-arrival-time of t seconds. Note that $P^r(p, z, s, t)$ considers only reuse request subsequences whereas, $P^a(s, t)$ considers all request subsequences. And for all practical purposes, the all subsequence descriptor can be approximated as $P^a(s, t) = \sum_{p,z} P^r(p, z, s, t)$, i.e., the statistical properties of a request subsequence and a reuse request subsequence that consists of s unique bytes and are of duration t seconds are similar.

Relation to other Footprint Descriptors. Footprint Descriptors (FDs) are succinct traffic models that capture the caching properties of a trace. The FD approach to cache provisioning was pioneered in [54], FDs are now routinely computed and widely used in making cache provisioning decisions in major CDNs [55]. A limitation of the original FDs is that it can be only be used to compute the RHR of a trace and not the BHR. Work in [50] extends the notion of a FD to a byte-weighted Footprint Descriptor (bFD) that is used to compute the BHR of a trace. Therefore, FD or bFD can compute either RHR or BHR of a trace and not both. Further, FDs and bFDs do not capture finer object-level properties such as object size distribution, popularity distribution and request size distribution of a trace. Our new traffic model – Popularity-Size Footprint Descriptor (pFD) – overcomes these limitations by succinctly capturing object-level properties and cache properties of a trace. By incorporating object-level properties, pFD can be used to compute both the RHR and BHR of a trace.

3.3 How pFD satisfies model requirements

We now show that our traffic model pFD satisfies the requirements that were described in §3.1.

(1) Generative. Cache-level properties and the object-level properties of a trace can be computed from a pFD.

a) Object-level properties. The SZ, POP and REQSZ can be computed from pFD as follows.

$$SZ(z) = \frac{1}{Z}\Big(\sum_{p} P^r(p, z, \infty, \infty)\Big), \tag{1}$$

$$POP(p) = \frac{1}{Z}\Big(\sum_{z} P^r(p, z, \infty, \infty)\Big), \tag{2}$$

$$REQSZ(z) = \sum_{p,s,t} P^r(p, z, s, t), \tag{3}$$

where Z is a normalizing factor,

$$Z = \sum_{p,z} P^r(p, z, \infty, \infty).$$

Recall that, in the trace, the first request made for an object is considered to have an infinite stack distance and an infinite inter-arrival-time. The fraction of such requests is given by $Z = \sum_{p,z} P^r(p, z, \infty, \infty)$. These requests were made for new objects i.e., objects not previously seen in the trace. Now, $P^r(p, z, \infty, \infty)$ gives us the fraction of requests that were made for a new object and the new object has a popularity p and size z. Therefore, the probability that the new object has a popularity p and size z is computed as $\frac{1}{Z}\Big(P^r(p, z, \infty, \infty)\Big)$. Thus, the computation of SZ(z) and POP(p) follow from Eq. 1 and Eq. 2, respectively. The REQSZ(z) is the probability that a request is made for an object of size z and can be obtained from Eq. 3.

b) Cache-level properties. We will now describe how the rHRC and the bHRC of a trace can be computed from a pFD.

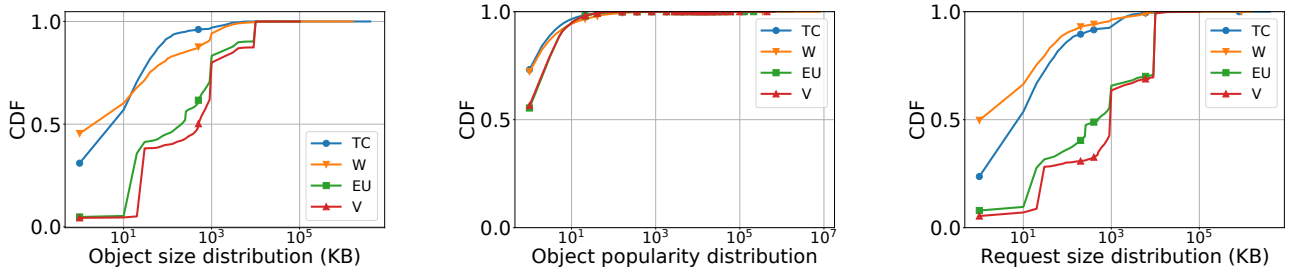
THEOREM 1. *Let rHRC(s) and bHRC(s) be the request hitrate and byte hitrate of trace* Π *for an LRU cache of size s. The rHRC(s) and bHRC(s) are computed from pFD of* Π *as follows,*

$$rHRC(s) = \sum_{p,z,t} \sum_{s' \le s} P^r(p, z, s', t), \tag{4}$$

$$bHRC(s) = \left(\frac{1}{Z}\right) \sum_{p,z,t} z. \sum_{s' \le s} P^r(p, z, s', t), \tag{5}$$

where $Z = \sum_{p,z,s,t} z.P^r(p, z, s, t)$.

PROOF. Consider an LRU cache of size s. Let $\theta = \{r_i, \ldots, r_j\}$ be a reuse request subsequence of trace Π and let r_i (and r_j) be a request for an object o. On request r_i, object o is moved to the most



(a) (b) (c)

Figure 3: Variability in object-level properties of different traffic classes (Table 3)

recently used (MRU) position in the cache. Now, if the sum of the sizes of the unique objects that are requested before r_j exceeds s, then the LRU cache evicts object o. And on request r_j, we would incur a cache miss. Thus, for an LRU cache of size s, r_j is served from cache only if θ contains less than s unique bytes, such an event occurs with probability given by Eq. 4.

Now, let Π^B be a byte sequence that is obtained by replacing each request in Π by its constituent bytes. Let $\beta = \{b_{ij}, \ldots, b_{kl}\}$ be a *reuse byte subsequence* of Π^B i.e., b_{ij} and b_{kl} correspond to the same byte and is not requested elsewhere in β. Let the number of unique bytes in β be s'. We will now derive an expression that gives us the probability that the request for byte b_{kl} will be a cache hit. As previously argued, the request for byte b_{kl} will be a cache hit if $s' \leq s$. To obtain $P(s' \leq s)$, we first define a random variable X that captures the size of the object that byte b_{kl} belonged to. Therefore,

$$P(s' \leq s) = \sum_z P(s' \leq s | X = z)\, P(X = z)$$

Now, consider $P(X = z)$. Let N be the number of requests in Π. The number of requests in Π that were made for an object of size z is $P^r(z).N$. Here $P^r(z) = \sum_{p,s,t} P^r(p, z, s, t)$. Therefore, the number of bytes in Π that belonged to requests that were made for an object of size z is $P^r(z).N.z$. And the total bytes in the trace is given by $\sum_{z'} z'.P^r(z').N$. Thus, $P(X = z) = \frac{P^r(z).z}{\sum_{z'} z'.P^r(z')}$. Hence,

$$\begin{aligned}
P(s' \leq s) &= \sum_z P(s' \leq s | X = z)\, \frac{P^r(z).z}{\sum_{z'} z'.P^r(z')}, \\
&= \sum_z \frac{\sum_{s' \leq s} P^r(z, s')}{P^r(z)}\, \frac{P^r(z).z}{\sum_{z'} z'.P^r(z')}, \\
&= \frac{1}{\sum_{z'} z'.P^r(z')} \sum_z z. \sum_{s' \leq s} P^r(z, s'), \\
&= \frac{1}{\sum_{p,z,s,t} z.P^r(p, z, s, t)} \sum_{p,z,t} z. \sum_{s' \leq s} P^r(p, z, s, t), \\
&= \frac{1}{Z} \sum_{p,z,t} z. \sum_{s' \leq s} P^r(p, z, s, t)
\end{aligned}$$

This completes the proof. □

(2) Succinct. pFD's are much smaller than the original traces. For instance, an original trace of size 100GB results in a pFD of size 100MB, a reduction factor of 1000.

(3) Efficiently computable. Using efficient a B-Tree style data structure [1] to represent the cache, pFD can be computed in $O(N \log m)$ using methods described in [8]. Here, N and m are the length and the number of unique objects, respectively, in the trace.

(4) Shareable. In contrast to original traces, since pFD's contain only aggregate distributions and no object or personal identifiers, they contain no personal identifiable information (PII), making it easier to share across organizations.

(5) Composable. In § 4, we derive a calculus to show that pFD's are composable, leading to design of the traffic mixer.

4 Traffic mixer

CDNs host and deliver thousands of traffic classes through their globally distributed servers. It is customary for a cache to serve varying mixes of traffic classes. An example of a traffic class mix that a production cache may serve is 5Mbps of Hulu videos, 10 Mbps of Microsoft downloads and 20 Mbps of Facebook images. This requires us to efficiently derive the pFD of the traffic mix given pFD's of individual traffic classes. We now describe a *calculus* that enables us to perform addition and scaling operations on the pFD's of traffic classes using efficient Fourier domain operations. *The main advantage of the calculus is that it enables efficiently manipulating concise footprint descriptors without having to operate on the voluminous original traces.*

4.1 Addition operator

Given pFD_1 and pFD_2 of original traces Π_1 and Π_2 respectively, we would like to compute the pFD of the traffic mix $\Pi = \Pi_1 \oplus \Pi_2$ that is obtained by interleaving requests in Π_1 with requests in Π_2 based on time. Let the resultant trace be Π. A key observation made in [54] that facilitates the calculus is that for a reuse request subsequence θ of Π that consists of s unique bytes and is of duration t, some s_1 unique bytes are from Π_1 and the rest $s - s_1$ unique bytes are from Π_2. This holds under the assumption that Π_1 and Π_2 consist of disjoint objects. The disjoint object assumption commonly holds, for example, when Π_1 and Π_2 are traces of different traffic classes. Thus, to compute the probability that the reuse request subsequence θ contains s unique bytes ($P(s|t)$), a product of the probabilities of obtaining $s1$ unique bytes from Π_1, and $s - s_1$ unique bytes from Π_2 ($P_1(s_1|t)P_2(s - s_1|t)$) is computed and summed over all possible s_1. It is expressed by the following,

$$\begin{aligned}
P(s|t) &= \sum_{s_1 \leq s} P_1(s_1|t) P_2(s - s_1|t), \\
&= P_1(s|t) * P_2(s|t),
\end{aligned}$$

where $*$ is the convolution operator. The computation can be sped up from $O(n^2)$ to $O(n \log n)$ using the Fast Fourier Transform [54].

We will now adapt this observation to find the pFD of the interleaved sequence Π. Let pFD of Π be $\langle \lambda, P^r(p,z,s,t), P^a(s,t)\rangle$, pFD_1 of Π_1 be $\langle \lambda_1, P_1^r(p,z,s,t), P_1^a(s,t)\rangle$ and pFD_2 of Π_2 be $\langle \lambda_2, P_2^r(p,z,s,t), P_2^a(s,t)\rangle$. The request rate of Π can be simply computed as $\lambda = \lambda_1 + \lambda_2$. We will now find an expression to quantify $P^r(p,z,s,t)$. Consider the reuse request subsequence $\theta = \{r_i, \ldots, r_j\}$, where $i < j$, of Π. Recall that by definition, the first and last request in a reuse request subsequence is made for the same object. And the expression $P^r(p,z,s,t)$ denotes that probability that θ has the following properties: (1) r_i (and r_j) is made for an object o that has popularity p and is of size z; (2) it consists of s unique bytes; (3) is of duration t seconds. To find $P^r(p,z,s,t)$ we make use of the fact that θ is composed of subsequences θ_1 and θ_2 that are from Π_1 and Π_2, respectively. The first request r_i in θ could belong to either θ_1 or θ_2. Therefore,

$$\begin{aligned} P^r(p,z,s,t) = {} & P(p,z,s,t|r_i \in \theta_1)\, P(r_i \in \theta_1) \\ & + P(p,z,s,t|r_i \in \theta_2)\, P(r_i \in \theta_2) \end{aligned} \tag{6}$$

Now, consider first half of the RHS, say P_1, in Equation 6.

$$\begin{aligned} P_1 &= P(p,z,s,t|r_i \in \theta_1)\, P(r_i \in \theta_1) \\ &= P(s|r_i \in \theta_1, p,z,t)\, P(p,z,t|r_i \in \theta_1)\, P(r_i \in \theta_1) \end{aligned}$$

Here, the term $P(s|r_i \in \theta_1, p,z,t)$ is the probability that θ contains s unique bytes given (1) the first request r_i (and r_j) is from θ_1; (2) r_i is for an object of popularity p and size z; and (3) θ is of duration t seconds. We now use of the observation that among the s unique bytes, some s_1 unique bytes come from θ_1 and the rest $s - s_1$ come from θ_2. As the first request in θ is from θ_1 and is for an object with popularity p and size z, the probability that s_1 unique bytes are from θ_1 is obtained as $P_1^r(s_1|p,z,t)$. The rest $s - s_1$ unique bytes that are from θ_2 could be from any object and the probability of which is given by $P_2^a(s - s_1|t)$. Therefore, we can expand the above equation,

$$P_1 = \left(\sum_{s_1} P_1^r(s_1|p,z,t) P_2^a(s - s_1|t) \right) P(p,z,t|r_i \in \theta_1)\, P(r_i \in \theta_1).$$

Now, $P(p,z,t|r_i \in \theta_1)$ is the probability that (1) r_i is a request for an object with popularity p and size z, and (2) θ_1 is of duration t seconds. This can be computed as $\sum_{s'} P_1^r(p,z,s',t)$. And finally, $P(r_i \in \theta_1)$ can be obtained as $\frac{\lambda_1}{\lambda_1+\lambda_2}$. We now arrive at the final expression by making appropriate substitutions.

$$P_1 = \left(\sum_{s_1} P_1^r(s_1|p,z,t) P_2^a(s - s_1|t) \right) \left(\sum_{s'} P_1^r(p,z,s',t) \right) \left(\frac{\lambda_1}{\lambda_1 + \lambda_2} \right) \tag{7}$$

Now, the computation $\sum_{s_1} P_1^r(s_1|p,z,t) P_2^a(s - s_1|t))$ is identified as a convolution. Therefore,

$$P_1 = \left(P_1^r(s|p,z,t) * P_2^a(s|t) \right) \left(\sum_{s'} P_1^r(p,z,s',t) \right) \left(\frac{\lambda_1}{\lambda_1 + \lambda_2} \right). \tag{8}$$

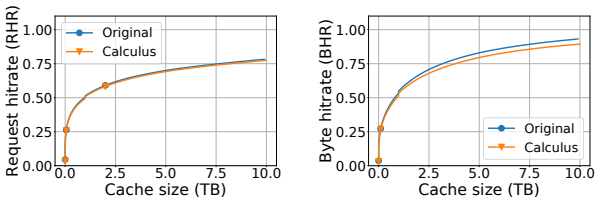
Now, we can leverage tools from Fast Fourier Transform (FFT) to compute the convolution operator. Using FFT, the time complexity of the convolution is reduced from $O(S^2)$ to $O(S \log S)$. The second half of the RHS in Eq. 6, say P_2, can be similarly derived by interchanging subscripts 1 with 2 in Eq. 8. The procedure is in Alg. 1.

Algorithm 1 Addition operator

1: **Input.** $\text{pFD}_1 = \langle \lambda_1, P_1^r(p,z,s,t), P_1^a(s,t)\rangle$ of trace Π_1 and $\text{pFD}_2 = \langle \lambda_2, P_2^r(p,z,s,t), P_2^a(s,t)\rangle$ of trace Π_2. Let Q, Z, S and T be the buckets for p, z, s and t, respectively.

2: **Output.** The $pFD = \langle \lambda, P^r(p,z,s,t), P^a(s,t)\rangle$ of the interleaved trace $\Pi = \Pi_1 \oplus \Pi_2$.

3: $\lambda = \lambda_1 + \lambda_2$
4: **for** $t \in T$ **do**
5: $\quad P^a(t) = \frac{\lambda_1}{\lambda_1+\lambda_2} P_1^a(t) + \frac{\lambda_2}{\lambda_1+\lambda_2} P_2^a(t)$
6: $\quad$ **for** $p \in Q$ **do**
7: $\quad\quad$ **for** $z \in Z$ **do**
8: $\quad\quad\quad P_1^{p,z,t} = \sum_{s'} P_1^r(p,z,s',t)$
9: $\quad\quad\quad P_2^{p,z,t} = \sum_{s'} P_2^r(p,z,s',t)$
10: $\quad\quad\quad P^r(p,z,s,t) = \frac{\lambda_1}{\lambda_1+\lambda_2} P_1^{p,z,t} \left(P_1^r(s|p,z,t) * P_2^a(s|t) \right) +$
11: $\quad\quad\quad\quad \frac{\lambda_2}{\lambda_1+\lambda_2} P_2^{p,z,t} \left(P_2^r(s|p,z,t) * P_1^a(s|t) \right)$
12: $\quad\quad$ **end for**
13: $\quad$ **end for**
14: $\quad P^a(s|t) = P_1^a(s|t) * P_2^a(s|t)$
15: $\quad P^a(s,t) = P^a(t) P^a(s|t)$
16: **end for**
17: $\text{pFD} = \langle \lambda, P^r(p,z,s,t), P^a(s,t)\rangle$
18: **return** pFD



(a) (b)

Figure 4: The rHRC and bHRC as predicted by the pFD calculus aligns with the original

Now, the all subsequence descriptor $P^a(s,t)$ is the probability that any subsequence, say θ of Π, consists of s unique bytes and is of duration t seconds. The computation does not depend on which trace, θ_1 or θ_2, the first request r_i belongs to. This computation is as described in [54] and is mentioned for completeness.

$$P^a(s|t) = P_1^a(s|t) * P_2^a(s|t)$$

Time complexity. The time complexity of Alg. 1 is given by $O(|Q||Z||T||S|\log|S|)$, where Q, Z, S and T are the buckets for p, z, s and t in pFD. The product $|Q||Z||T|$ is for the three for loops in lines 4,6 and 7. And the convolution operator in line 10 is evaluated in $O(S \log S)$.

Empirical evidence. We empirically validate the Addition operator (Alg. 1). We obtain a subsequence (say Π) of the EU trace that consist of requests made to either Media0 and Media1 traffic class and compute the pFD $_{orig}$ of Π. Next, we obtain individual subsequences Π_1 and Π_2 of the EU trace that consist of requests made to traffic classes Media0 and Media1, respectively and compute their individual pFD's. Let the pFD's be pFD_1 and pFD_2. Now, we compute the pFD of the traffic mix, pFD $_{mix}$, using Alg.1. The rHRC and bHRC of the two scenarios is shown in Figure 4.

4.2 Scaling operator

Given a pFD of a trace Π we would like to compute the pFD$_\tau$ of trace Π_τ that is obtained by either increasing or decreasing the request rate of Π by a scaling factor of τ. By scaling Π by a factor τ, we effectively alter the inter-arrival-time between the requests in Π by a factor $\frac{1}{\tau}$. Thus, we compute pFD$_\tau$ as follows,

$$\lambda_\tau = \tau\lambda;\ P^r_\tau(p,z,s,t) = P^r(p,z,s,\frac{t}{\tau});\ P^a_\tau(s,t) = P^a(s,\frac{t}{\tau}) \quad (9)$$

The operator is similar to the scaling operator described in [54].

4.3 Parallelizing pFD operations

On taking a closer look at Equation 8, we observe that to compute the pFD of the traffic mix, the addition operator performs convolutions for every unique pair of p, z and t in the trace. A similar number of computations have to performed to obtain the scaled pFD. This results in a large number of operations that need to be performed. However, these operations are independent and can be performed in parallel. In our implementation (in python), we observe that the addition operator takes around 20 minutes for the experiment depicted in Figure 4 when run on a 56 core machine.

5 Trace Generator

In this section, we describe our algorithm that generates a synthetic trace with similar object-level properties and cache-level properties as the original trace. The procedure is described in Algorithm 2. The inputs to the algorithm are as follows,
(1) A pFD = $\langle \lambda, P^r(p,z,s,t), P^a(s,t)\rangle$ of the original trace Π_o.
(2) Length N of the synthetic trace that is to be generated.
For the output, the algorithm generates a synthetic trace $\Pi_s = r_1, \ldots, r_n$, where each r_i is a request for an object and is represented by a tuple $r_i = \langle t_i, o_i, z_i\rangle$ of timestamp, unique object identifier and object size.

Initialization phase. We first compute the joint POPSZ distribution from pFD using,

$$POPSZ(p,z) = \frac{1}{Z}\Big(P^r(p,z,\infty,\infty)\Big), \quad (10)$$

where $Z = \sum_{p,z} P^r(p,z,\infty,\infty)$. Recall that the first request made for an object in a trace is considered to have an infinite stack distance and an infinite inter-arrival-time. Therefore, Z gives us the probability that the request is for a new object i.e., the object has not yet been seen in the trace. Now, the distribution POPSZ gives us the probability that the object has a popularity p and size z It is computed in line 5 of Alg. 2. An empty list C that represents a cache is initialized in line 7. From lines 10-15, we create objects and assign the object a popularity p and size z by sampling from the joint POPSZ distribution. The process is repeated till the sum of the sizes of the objects in C exceeds the maximum *stack distance* s in $P^r(p,z,s,t)$. The term stack distance is commonly used to denote the number of unique bytes in a reuse request subsequence [43]. Alternatively, a definition that will be relevant to our algorithm is as follows. The $StackDistance[j]$ of an object at position j in C is the sum of the sizes of the objects in positions $[1, j)$ in C.

Trace generation phase. To generate a synthetic trace, we first compute $P^r(s|p,z)$ for each pair of (p,z) in $P^r(p,z,s,t)$. This is done

Algorithm 2 Synthetic trace generator

1: **Input.** (i) A pFD$\langle \lambda, P^r(p,z,s,t), P^a(s,t)\rangle$ (ii) trace length N.

2: **Output.** A synthetic trace $\Pi_s = \{r_1, \ldots, r_N\}$, where $r_i = \langle t_i, o_i, z_i\rangle$ is a tuple of timestamp, object identifier and object size.

3: **Phase 1 - Initialization.**
4: $POPSZ(p,z) = \frac{1}{Z}\Big(P^r(p,z,\infty,\infty)\Big)$.
5: Compute $P^r(s|p,z)$ for each pair of p, z in pFD.

6: $C \leftarrow \{\}$, $C_{size} = 0$.
7: C_{max} is the maximum **finite** s in P^r.
8: $request\_count \leftarrow \{\}$
9: **while** $C_{size} < C_{max}$ **do**
10: Create object o and assign it a popularity p and size z by sampling from $POPSZ(p,z)$.
11: Add object o to the list C.
12: $C_{size} \leftarrow C_{size} + z$.
13: **end while**

14: **Phase 2 - Synthetic trace generation.**
15: $\Pi_s \leftarrow \phi$, $i \leftarrow 0$.
16: **while** $i < N$ **do**
17: Append the first object $o = \langle o_{id}, z\rangle$ in C to the trace Π_s.
18: $req\_count[o]$ += 1
19: Let p_o and z_o be the popularity and size of object o.
20: **if** $req\_count[o] = p_o$ **then**
21: Remove the object o from C.
22: Create new object o' and assign it a popularity p and size z by sampling from $POPSZ(p,z)$.
23: Add object o' at the end of the list C.
24: **else**
25: Sample stack distance s from $P^r(s|p_o, z_o, s < \infty)$
26: Remove o from C.
27: Compute $j = min\{k : StackDistance[k] > s\}$.
28: For each $l > j$ move object at the l^{th} position in C to the $l + 1^{st}$ position.
29: Re-insert object o at position j in C
30: **end if**
31: $i \leftarrow i + 1$
32: **end while**
33: Assign timestamps to each request Π_s using the request rate λ.
34: **return** Π_s

in line 11 and it gives us the probability that a reuse request subsequence $\theta = \{r_i, \ldots, r_j\}$ of Π_o consists of s unique bytes provided request r_i (and r_j) is made for an object of size z and a popularity p.

The trace generation phase runs through lines 13 to 20. We initialize an empty synthetic trace Π_s in line 13. Further, we also maintain a statistic $request\_count$ that counts the number of requests made for each object in Π_s so far. In each iteration i, the object at the first position of C is examined. Let the object be identified as o. A request for object o is appended to the synthetic trace S and we increment $request\_count[o]$. Let the popularity and size of object o be p_o and z_o respectively. Now there are two cases,

(1) If the $request\_count[o]$ equals p_o, then the object is removed from C and a new object o' is inserted at the end of the list C. Object o' is assigned a popularity and size by sampling from the POPSZ distribution.

(2) Otherwise, we sample a stack distance s from $P^r(s|p_o, z_o, s < \infty)$. The object o is removed from C and re-inserted into C at a position $j = min\{k : StackDistance[k] > s\}$.

As a final step, we assign a timestamp to the requests in Π_s using the request rate λ from pFD.

We will now prove that the total variation distance for the *SZ*, *POP* and *REQSZ* of the synthetic trace Π_s and original trace Π_o tends to zero. Further, we will prove that the total variation distance for the rHRC and bHRC of traces Π_s and Π_o for an LRU cache tends to zero.

THEOREM 2. *Given a pFD $\langle \lambda, P^r(p, z, s, t), P^a(s, t)\rangle$ of an original trace Π_o, Alg. 2 produces a synthetic trace $\Pi_s = \{r_1, \ldots, r_N\}$, where the i^{th} request r_i is a tuple $\langle t_i, o_i, z_i\rangle$ of timestamp, object id and object size and N is the synthetic trace length. As $N \to \infty$, the total variation distances for the SZ, POP and REQSZ distributions of traces Π_s and Π_o tends to zero.*

PROOF. Each object o in Π_s is assigned a size z and popularity p by sampling from the joint popularity-size distribution of Π_o. As $N \to \infty$, the number of objects in Π_s tends to ∞. Therefore, as the number of objects in Π_s tends to ∞, the total variation distance of SZ for traces Π_o and Π_s tends to zero. We will now show that the object o is requested p times in Π_s. In each iteration i (line 20) of Alg. 2, we add the first object in the list C to Π_s. We also check if the request count of the object in $\{r_1, \ldots, r_i\}$ equals the assigned popularity. If yes, the object is removed from the list and no further requests are made for it in $\{r_{i+1}, \ldots, r_N\}$. If not, the object is added back into the list at the sampled stack-distance. Now, observe that as $N \to \infty$, the only objects that will have a request count lesser than their assigned popularity are the objects that remain in the list C. The number of such objects is small as compared to the total number of objects in Π_s. Thus, the total variation distance of the POP for traces Π_o and Π_s tends to zero as $N \to \infty$. Now, as each object is assigned a popularity and size from the joint popularity-size distribution of Π_o, the total variation distance of REQSZ for traces Π_s and Π_o tends to zero. □

THEOREM 3. *Given a pFD $\langle \lambda, P^r(p, z, s, t), P^a(s, t)\rangle$ of an original trace Π_o, Alg. 2 produces a synthetic trace $\Pi_s = \{r_1, \ldots, r_N\}$, where the i^{th} request r_i is a tuple $\langle t_i, o_i, z_i\rangle$ of timestamp, object id and object size and N is the synthetic trace length. As $N \to \infty$, the total variation distance for the rHRC and bHRC of traces Π_s and Π_o for an LRU cache tends to zero.*

PROOF. Let $P(p, z, s) = \sum_t P^r(p, z, s, t)$. It is the probability that a reuse request subsequence ρ in Π_o has the following properties: (i) the last (and first) request in ρ is for an object of popularity p and size z, and (ii) the sum of the sizes of unique objects in ρ is s. From Theorem 1, we know that the rHRC and the bHRC of the original trace Π_o for an LRU cache can be computed from $P(p, z, s)$. Now, consider a reuse request subsequence θ in Π_s. Let $P'(p, z, s)$ be the probability that (i) the last (and first) request in θ is for an object of popularity p and size z, and (ii) the sum of the sizes of unique objects in θ is s. We will show that $P'(p, z, s) = P(p, z, s)$. Hence, the total variation distance for the rHRC and bHRC of Π_s and Π_o for an LRU cache tends to zero.

Let r_j be the last request in θ. The probability that r_j is a request for an object with popularity p and size z is obtained as $P'(p, z) = \sum_s P'(p, z, s)$. Now, we know that each object in Π_s is assigned a popularity and size by sampling from the joint popularity-size distribution of the original trace Π_o. Further, in Theorem 2 we showed that if an object is assigned a popularity p, it is requested p times in Π_s. Therefore, the probability that a request r_j in Π_s is for an object of popularity p and size z equals the probability that a request s_j in Π_o is for an object of popularity p and size z. Thus,

$$P'(p, z) = P(p, z) = \sum_s P(p, z, s). \tag{11}$$

We will now show that $P'(s|p, z) = P(s|p, z)$. There are two cases.

Case 1. $s = \infty$. In this case, r_j is the first request for an object in Π_s, i.e., the object has not been requested in $\{r_1, \ldots, r_{j-1}\}$. Recall that the first request for any object in the trace is considered to have an infinite stack distance. Now, as $P'(p, z) = P(p, z)$, the fraction requests made for an object with popularity p and size z in Π_s and Π_o is equal. This implies that the fraction of *first* requests made for an object with popularity p and size z in Π_s and Π_o is equal. Therefore,

$$P'(s = \infty|p, z) = P(s = \infty|p, z). \tag{12}$$

Case 2. $s < \infty$. Let r_j be a request for object o. In this case, object o has been previously requested in Π_s i.e., object o is requested in $\{r_1, \ldots, r_{j-1}\}$. Now, consider the previous request r_i that was made for object o in Π_s. The request r_i was added to Π_s in the i^{th} iteration of Alg. 2 (line 17). In the i^{th} iteration, we sampled a stack distance s from the probability distribution $P(s|p, z, s < \infty)$ (line 25). We will now show that Alg. 2 ensures that the number of unique bytes in $\theta = \{r_i, \ldots, r_j\}$ is s and hence,

$$P'(s|p, z, s < \infty) = P(s|p, z, s < \infty). \tag{13}$$

In the i^{th} iteration, let k be the smallest index in C such that the sum of the sizes of objects in positions 1 to k in C is greater than or equal to s. Let z_k be the size of the object at position k. The object o that was at the first position in C was inserted back into C at a stack distance of at least s and at most $s + z_k$. The subsequent request for object o in Π_s is made on request r_j. Now, the unique objects in $\theta = \{r_i, \ldots, r_j\}$ are the objects present in C from positions 1 to k. Therefore the number of unique bytes in θ is at least s and at most $s + z_k$. Now, since the number of unique objects in reuse request subsequences are typically large, $z_k \ll s$. Therefore, the number of unique bytes in θ tends to s.

Therefore, from Eq. 12 we know that for $s = \infty$, the probability $P'(s = \infty|p, z) = P(s = \infty|p, z)$ and from Eq. 13 we know that for $s < \infty$, $P'(s|p, z, s < \infty) = P(s|p, z, s < \infty)$. This implies,

$$P'(s|p, z) = P(s|p, z) \tag{14}$$

for all possible stack distances s. Now $P'(p, z, s) = P'(s|p, z)P'(p, z)$. Using Eq. 11 and Eq. 14, $P'(p, z, s) = P(s|p, z)P(p, z) = P(p, z, s)$. Now, as the rHRC and bHRC of Π_s and Π_o for an LRU cache can be computed from $P(p, z, s)$, the total variation distance for the rHRC and bHRC of traces Π_s and Π_o for an LRU cache tends to zero. □

Time complexity. The list C in Alg. 2 is implemented as a B-Tree that allows insertion and deletion in $\log m$ time, where m is the number of objects in the list. In each iteration of the algorithm, an object is either removed from the list or re-inserted back into the list.

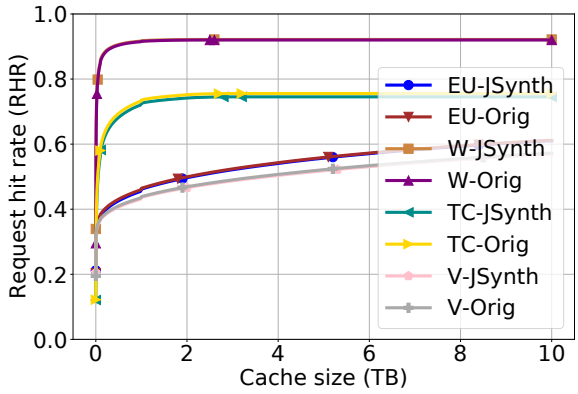


Figure 5: rHRC of original and synthetic traces in Table 1

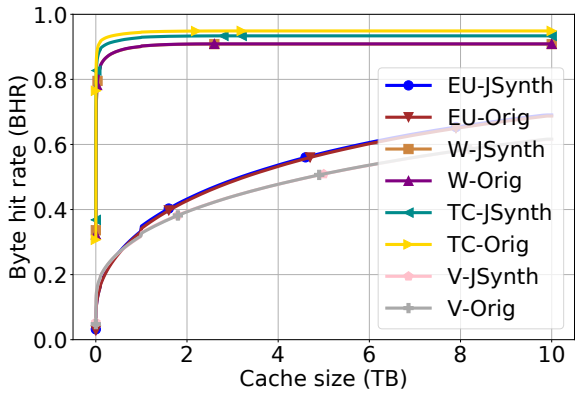


Figure 6: bHRC of original and synthetic traces in Table 1

Now, as the algorithm runs for N iterations, the time complexity of Alg. 2 is $O(N \log m)$.

6 Empirical evaluation

In this section, we show that JEDI produces a synthetic trace that has similar object-level and cache-level properties as the original trace. We use the original production traces described in Table 1 for our evaluation. We use JEDI to first compute the pFD from the original traces and then produce a synthetic trace from it. The trace produced by JEDI will be denoted as JSynth. Further, we evaluate JEDI against TRAGEN, the current state-of-the-art synthetic trace generation tool [50]. To facilitate the comparison, we use TRAGEN to first compute the footprint descriptor traffic models, FD and bFD [50, 54], of the original traces. We then use TRAGEN to produce a synthetic trace from it. We denote the synthetic trace produced from FD and bFD as Tragen-R and Tragen-B, respectively. TRAGEN guarantees that Tragen-R and Tragen-B have similar rHRC and bHRC, respectively, as the original trace for an LRU cache. The traces JSynth, Tragen-R and Tragen-B consist of 200 million requests each. Finally, we show that prior tools other than TRAGEN produce synthetic traces that also fail to satisfy cache-level properties since they use the LRUSM approach.

6.1 JEDI satisfies object-level properties

We show that JEDI produces a synthetic trace with similar object-level properties i.e., SZ, POP and REQSZ as the original trace. The results are shown in Fig. 7. We observe that the both JSynth and Tragen-R have similar SZ as the original trace Fig. 7a. We use the original trace VIDEO for the experiment. The result is as expected, as both TRAGEN and JEDI use SZ to assign object sizes. The total variation distance in SZ of JSynth and original trace is 3.1×10^{-3}.

Next, we observe that the POP of JSynth and the original trace are similar (Fig. 7b) with a total variation distance of 3.66×10^{-3}. However, the POP of Tragen-R differs considerably from that of the original. TRAGEN does not model popularity distribution, and hence, Tragen-R cannot have a similar POP as the original trace.

Further, the REQSZ of JSynth and original trace are similar but the REQSZ of Tragen-R differs from that of the original (Fig. 7c). The total variation distance in the REQSZ of JSynth and the original trace is 4.22×10^{-3}. Again, as TRAGEN does not model popularity distribution, Tragen-R cannot have a similar REQSZ as the original traces. We observe similar results for Tragen-B.

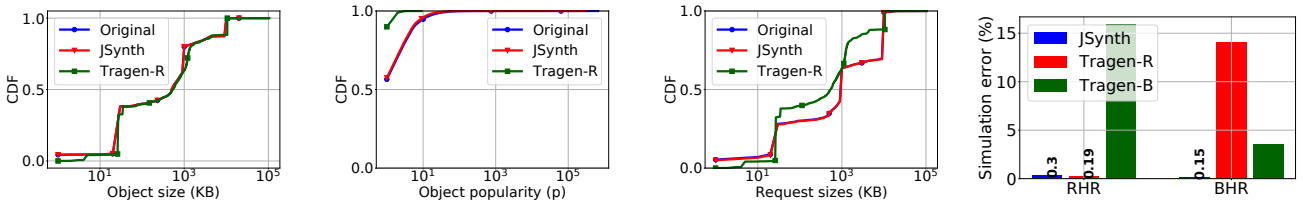
6.2 JEDI satisfies cache-level properties

We now show that JSynth has similar cache-level properties as the original traces. We first show that the total variation distance for the rHRCs and bHRCs of the original traces and JSynth traces is small for an LRU cache. The rHRC(z) (resp. bHRC(z)) of a trace for a cache algorithm A gives the RHR (resp. BHR) for a cache of size z that uses the cache algorithm A. We then show that JSynth and the original traces yield similar RHR and BHR for a wide variety of cache algorithms by implementing the cache algorithms and performing cache simulations.

6.2.1 Hitrace curves for an LRU cache. The bHRC and the rHRC of the original trace and the corresponding JSynth trace for an LRU cache is depicted in Fig. 5 and Fig. 6, respectively. We observe that both the rHRCs and bHRCs of the original traces and JSynth align. We observe a total variation distance of 9.82×10^{-3}, 4.4×10^{-2}, 8.68×10^{-3}, 5.9×10^{-3} in the rHRC for the VIDEO, WEB, EU and TC trace, respectively. And observe a total variation distance of 5.5×10^{-4}, 1.26×10^{-2}, 4.8×10^{-3}, 6.1×10^{-2} in the bHRC for the VIDEO, WEB, EU and TC trace, respectively. As the differences are small in all cases, we conclude that the HRCs of original trace and the synthetic trace produced by JEDI are nearly equal and are hence, similar.

A key difference between JEDI and TRAGEN is that JEDI produces a *single* trace that has similar rHRC and bHRC as the original. Whereas, TRAGEN can produce a synthetic trace that has either a similar rHRC or bHRC as the original, and not both simultaneously. If rHRC of the Tragen-B (resp. Tragen-R) trace is compared with the rHRC (resp. bHRC) of the original trace we observe an error of 16 % (resp. 14 %) on an average across all cache sizes (Fig. 7d).

6.2.2 Validation on other cache algorithms. We now show that the synthetic trace and the original trace yield similar RHR and BHR for a wide variety of cache algorithms. Most cache algorithms that are deployed in production systems use some combination of access patterns (temporal locality), popularity distribution, and the size distribution of the objects to make caching decisions. We choose algorithms from the literature that are commonly-used in practice and that also span the space of features used for decision making. We broadly categorize the cache algorithms as (A) popularity and size based admission algorithms; (B) popularity and size based eviction algorithms; and (C) recency based eviction algorithms. We show that



(a) SZ distribution (b) POP distribution (c) REQSZ distribution (d) Error in RHR and BHR for JSynth, TRAGEN-R and TRAGEN-B

Figure 7: (a), (b) and (c) Comparing object-level properties of synthetic trace and the original trace. (d) Comparing TRAGEN and JEDI

while Tragen-R and Tragen-B traces yield similar RHRs and BHRs for a subset of cache algorithms (mainly recency based eviction algorithms), the trace JSynth yields similar RHRs and BHRs for *all* the cache algorithms we experimented on.

The cache simulation experiments are performed as follows. We first select a subset of the original traces from Table 1 and produce their corresponding synthetic traces JSynth, Tragen-R and Tragen-B. Now, to evaluate the performance of a synthetic trace for a cache algorithm, we run cache simulations using the original traces and the corresponding synthetic traces on multiple cache sizes. Now, the *simulation error* for the synthetic trace is mean of the absolute difference in hitrates. We multiply this quantity by 100 as hitrates are more easily understandable in percentages. The simulation error is thus an aggregate metric that quantifies the performance of a synthetic trace for a cache algorithm.

(A) Popularity and size based admission algorithms. We consider the following algorithms: ThLRU-*z*, Bloomfilter-*n*, ThLRU-Prob, ThLRU-z/Bloomfilter-*n*, Adaptsize. The algorithms are described in the §10.1. Note that each algorithm uses LRU for eviction. We use traces VIDEO and WEB for the experiments.

We first consider cache algorithms that use object size as the admission criteria i.e., objects larger than a size threshold are not admitted into the cache. The algorithms are designed to maximize the RHR [14]. Fig. 8 depicts the simulation error in the RHR for the various cache algorithms. For each algorithm, we run simulations on cache sizes 64GB, 128GB and 256GB that are representative of commonly used RAM cache sizes on production servers. We observe an average simulation error of 0.9%, 3% and 6% for the traces JSynth, Tragen-R and Tragen-B, respectively, across all cache algorithms. The maximum simulation error of 2.5 % is observed for the JSynth trace for Adaptsize. Whereas, a maximum simulation error of 4.7% and 9% is observed for the traces Tragen-R and Tragen-B, respectively, for the cache algorithms ThLRU-Prob and THLRU-8MB. *Therefore, we conclude that the simulation error for the JSynth is much smaller than the simulation error for the traces Tragen-R and Tragen-B for all size-based cache admission algorithms.*

We now evaluate JEDI for cache admission algorithms that use popularity as the admission criteria. The results of our simulations are shown in Fig. 9. We observe that the simulation error for the trace JSynth is lesser than 0.6% across all cache algorithms. The average simulation error for the JSynth trace is 0.5%. However, the simulation error is considerably larger for the Tragen-R and Tragen-B traces, with the minimum simulation error being 11%. As TRAGEN does not aim to produce a synthetic trace with similar popularity distribution as original traces, TRAGEN performs poorly on popularity based admission algorithms. *Therefore,the simulation error for the JSynth*

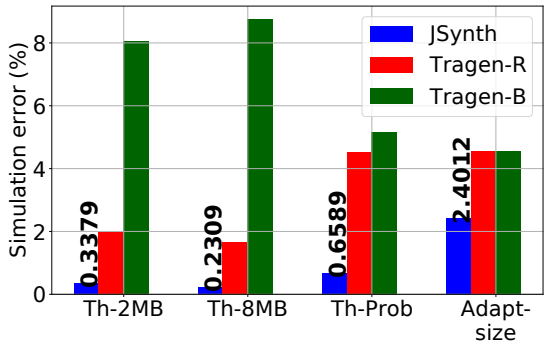
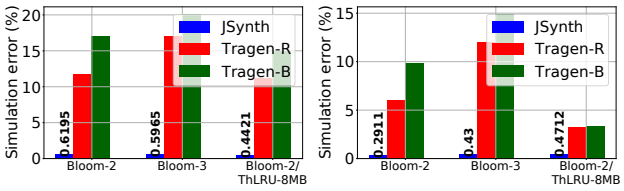


Figure 8: Simulation error in RHR values for Tragen and Synthetic trace for size based admission policies.



(a) Simulation error in RHR (b) Simulation error in BHR

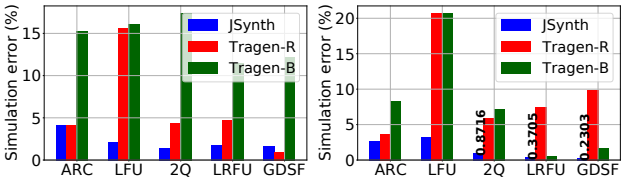
Figure 9: Observed simulation error for cache admission algorithms that use popularity as a criteria

is much smaller than the simulation error for the traces Tragen-R and Tragen-B for all popularity-based cache admission algorithms.

(B) Popularity and size based eviction algorithms. We now evaluate JEDI on cache algorithms that use the popularity and size of the objects to make eviction decisions (Figure 10). We use traces VIDEO and EU for the experiments. We consider following algorithms: GDSF, LRU, LRFU, ARC, 2Q (described in §10.2).

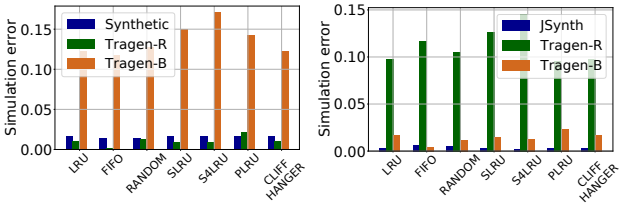
We observe that JSynth produces an average simulation error of 2% and 1.3% in RHR and BHR, respectively, across all the cache algorithms. Whereas, the trace Tragen-R produces an average error of 5% and 9% in RHR and BHR, respectively. And Tragen-B produces a simulation error of 14% and 7.5% in RHR and BHR, respectively. Further, JEDI provides a smaller simulation error as compared to TRAGEN for each of the size and popularity based cache eviction algorithms. *Therefore, we conclude that the JSynth produces a smaller simulation error, on an average by 6%, across all popularity and size based cache eviction algorithms.*

(C) Recency based eviction algorithms. We use traces VIDEO and EU for the experiments. Apart from LRU, we evaluate FIFO, RANDOM, SLRU, PLRU and CLIFFHANGER (§10.3).



(a) Simulation error in RHR (b) Simulation error in BHR

Figure 10: Observed simulation error for cache eviction that use size, popularity and recency for eviction



(a) Simulation error in RHR (b) Simulation error in BHR

Figure 11: Observed simulation error for recency based cache eviction algorithms

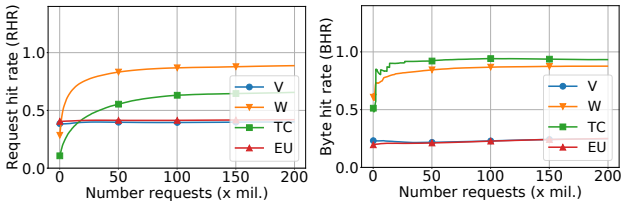
We observe that both JEDI and TRAGEN produce synthetic traces that yield small simulation errors across all the recency-based cache eviction algorithms (Fig. 11). In particular, the trace JSynth and Tragen-R produce an average simulation error of 1.5% and 0.6%, respectively, in the RHR across all the algorithms. And JSynth and Tragen-B produce an average simulation error of 0.5% and 1%, respectively, in the BHR. Thus, JEDI provides similar simulation error as TRAGEN for recency based cache eviction algorithms. However, a key difference between JEDI and TRAGEN, is that JEDI produces a single synthetic trace that has similar RHR and BHR as the original; whereas, TRAGEN produces two distinct synthetic traces Tragen-R and Tragen-B that have similar RHR and BHR, respectively, as the original. Additionally, the simulation error in BHR (resp. RHR) on using the Tragen-R (resp. Tragen-B) is significant and is on an average 12% (resp. 11%) across all cache algorithms. Therefore, Tragen-R and Tragen-B cannot be used interchangeably.

6.3 Determining synthetic trace length

We now answer the following question. *What trace length should be used for cache simulations?* The answer is that trace should be long enough that the hitrates reach a stable value in the duration of a cache simulation that starts with an empty cache. The length varies based on the access patterns in the trace, cache size and the cache algorithm. The observed RHRs and BHRs for the synthetic trace over time on a cache simulation that uses an LRU cache of size 500 GB is shown in Fig. 12. We observe that across the various traces used for our analysis, the hitrates stabilize after around 100 million requests. Hence, we recommend that users of JEDI use a trace length of at least a 100 million requests.

6.4 Synthetic traces for traffic mix scenarios

We now show that JEDI can produce a synthetic trace for any specified traffic mix scenario. As an example, consider a traffic mix scenario consisting of 20 reqs/second of the Media-0 traffic class and 70 reqs/second of the Media-1 traffic class from the EU trace. The traffic mixer module of JEDI computes the pFD of the specified traffic mix and the synthetic trace is generated from it.



(a) (b)

Figure 12: Convergence in RHR and BHR for the synthetic trace.

We compare the HRCs of the synthetic trace with the HRCs of the original EU trace. Fig. 12 depicts our HRCs derived from the pFDs of the synthetic and the original traces. We observe a total variation difference of 0.0067 (resp. 0.01154) in the rHRC (resp. bHRC) of the original and the synthetic trace.

6.5 Comparison with alternate approaches

Prior work in synthetic trace generation for CPU caches and Web caches use the LRUSM algorithm [13, 18]. Upon experimenting with the LRUSM algorithm to produce a synthetic trace, we found that the synthetic trace seldom has the same cache-level properties as the original trace. Fig. 14a depicts the HRCs of the synthetic trace and the original trace for an LRU cache. We observe a large difference in the HRCs of the original and synthetic trace. The experiment was performed using the VIDEO trace (Table 1).

On adapting the LRUSM algorithm for variable-size objects, we find that the BHR of the synthetic trace is much higher than the BHR of the original trace. This is because the synthetic trace contained a large fraction of large objects as compared to the original trace. The results for the VIDEO trace is depicted in Fig. 14b. Therefore, we conclude that the LRUSM approach fails to produce synthetic traces with similar cache-level properties as the original trace.

7 Related work

In this section, we first review prior work that study the characteristics of the Internet traffic and then provide an overview of the various synthetic workload generators that produce synthetic workloads representative of the identified characteristics.

Characterizing the Internet traffic. There exist several studies carried over the past two decades that characterize the workload of various Internet services [5, 9, 11, 12, 19, 27, 31, 42, 52, 53, 57, 60]. A first such study was done by Arlitt et. al. [11]. The work analyses

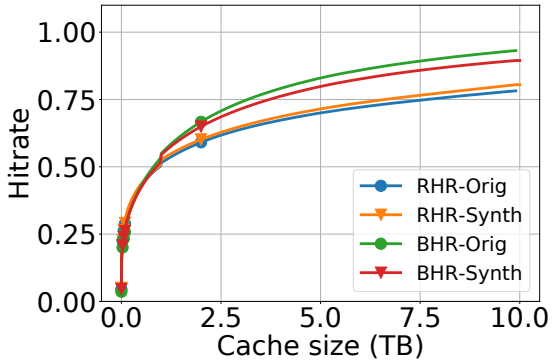


Figure 13: Traffic mix results for the EU trace

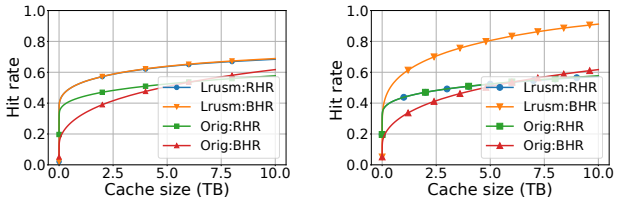


Figure 14: LRUSM approach. (a) HRCs under the LRUSM approach with stack initialized with unit size objects, (b) HRCs under LRUSM approach with stack initialized with object sizes.

six different datasets collected from different Internet web servers to identify ten salient characteristics of the web workload. The work also discusses the impact of the characteristics on caching and performance. The authors revisit the study 10 years hence [58] to show that the previously identified workload characteristics are still in tact. Similar studies that characterized the web workload were performed in [12, 42]. However, in the past decade, the content delivered on the Internet has significantly diversified and there exist several recent works that characterize the diverse workload [27, 31, 53, 60].

The work in [54], characterizes the workload of a content delivery network (CDN). CDNs serve content belonging to a diverse set of traffic classes, each with distinct set of properties such as access patterns, popularity distributions and size distributions. The cache-level properties of a traffic class are captured by a succinct composable model called Footprint Descriptors (FD). Further, the FD of a traffic mix scenario can be computed from individual FDs using the Footprint Descriptor Calculus. A limitation of FD is that it can only be used to compute the RHR of a traffic class. This limitation was addressed in [50], by proposing a byte-weighted Footprint Descriptor (bFD) that can be used to compute the BHR of a traffic class. However, neither FD nor bFD, capture both the bHRC and rHRC of a traffic class. Further, they do not capture finer object-level properties such as size distribution and popularity distribution, that are salient characteristics of any workload.

In this work, we overcome limitations of FD and bFD by proposing a succinct Popularity-Size Footprint Descriptor (pFD) traffic model that captures object-level properties and the cache-level properties of a traffic class. Further, we also derive a calculus to compute the pFD of a traffic mix given the pFDs of individual traffic classes.

Tools for synthetic trace generation. The first synthetic trace generation tools are SpecWeb96 [22] and HttpPerf [46]. Both tools generate representative http request workload with the same size and popularity distributions as the original workload. SURGE [13] produces a synthetic web workload that matches the size distribution, popularity distribution, request size distribution, temporal locality of request accesses, idle periods of individual users. Several other tools that satisfy all or a subset of the mentioned properties are Geist [34], WebPolygraph [49], Globetraff [36], MediSyn [56]. However, the tools do not generate a synthetic trace with similar hitrates as the original. The tool Prowgen [18] identifies specific properties of a workload that impact the hitrates of a proxy cache by generating a synthetic trace. Prowgen (and other tools mentioned above) use the LRUSM algorithm (Section 6.5) and hence cannot generate a synthetic trace with the same cache-level properties as the original.

A recent tool the produces a synthetic trace that has similar cache-level properties as the original is TRAGEN [50]. Specifically, TRAGEN produces a synthetic trace that has either a similar RHR or BHR as the original trace for only caches that use recency based eviction algorithms. Further, TRAGEN does not produce a synthetic trace with finer object-level properties such as popularity distribution and request size distribution. *In this work, we address the limitations of TRAGEN. We propose JEDI that produces a single synthetic trace that produces similar RHR and BHR as the original trace for a wide set of cache algorithms and also satisfies the object-level properties.*

ML-based synthetic trace generation. We are not aware of ML-based synthetic trace generation tools for simulating cache algorithms. However, there exist several tools in the networking community such as STAN [59], DoppelGANger [40], Netshare [62] that generate network packet-level traces that incorporate packet-level features such as source/destination IP, port, start and end times for the packet-flow etc. These tools use modern machine learning techniques like GANs [10, 28] to produce synthetic traces. While the ML-based techniques can be used to generate synthetic traces, they do not provide any theoretical guarantees. Whereas, our work relies on a sound theoretical model – the popularity-size footprint descriptors (pFDs) – that captures the cache-level and object-level properties of a trace.

8 Conclusion

In this work, we design and develop JEDI that significantly advances the state-of-the-art in synthetic trace generation. JEDI is the *first* tool that produces a synthetic trace that simultaneously matches the object-level properties (object size distribution, popularity distribution, request size distribtuion) and the cache-level properties (RHR and BHR) of the original trace. By matching both the object-level properties and the cache-level properties, JEDI is able to produce a synthetic trace that yields similar RHR and BHR as the original trace for a wide range of cache algorithms. JEDI will be seeded with pFD models computed from the original production traces obtained from Akamai's production servers and will allow researchers to produce realistic traces for their own caching research. JEDI is made open-source and is publicly available for download[2]. Further, we believe that the techniques developed in our work are more broadly applicable and can be used to generate synthetic traces for other caching systems such as DNS caches, CPU caches, Memcached [26], and not just CDN caches. Producing synthetic traces for caching systems across other domains is future work.

9 Acknowledgements

We would like to thank our anonymous reviewers and our shepherd Michael Sirivianos, whose valuable feedback and comments immensely improved the paper. This work was supported in part by NSF grants CNS-1763617, CNS-1901137, and CNS-2106463.

References

[1] B tree wikipedia. https://en.wikipedia.org/wiki/B-tree.
[2] Open zfs. https://en.wikipedia.org/wiki/OpenZFS.
[3] Total variation distance. https://en.wikipedia.org/wiki/Total_variation_distance_of_probability_measures.
[4] Zfs caching. https://en.wikipedia.org/wiki/ZFS.

[2]It can be downloaded from https://github.com/UMass-LIDS/Jedi

[5] S. Acharya, B. C. Smith, and P. Parnes. Characterizing user access to videos on the world wide web. In *Multimedia Computing and Networking 2000*, volume 3969, pages 130–141. International Society for Optics and Photonics, 1999.

[6] D. Achlioptas, M. Chrobak, and J. Noga. Competitive analysis of randomized paging algorithms. In *European Symposium on Algorithms*, pages 419–430.

[7] J. Alghazo, A. Akaaboune, and N. Botros. Sf-lru cache replacement algorithm. In *Records of the 2004 International Workshop on Memory Technology, Design and Testing, 2004.*, pages 19–24. IEEE, 2004.

[8] G. Almási, C. Caşcaval, and D. A. Padua. Calculating stack distances efficiently. In *Proceedings of the 2002 workshop on Memory system performance*, pages 37–43, 2002.

[9] V. Almeida, A. Bestavros, M. Crovella, and A. De Oliveira. Characterizing reference locality in the www. In *Fourth International Conference on Parallel and Distributed Information Systems*, pages 92–103. IEEE, 1996.

[10] M. Arjovsky, S. Chintala, and L. Bottou. Wasserstein generative adversarial networks. In *International conference on machine learning*, pages 214–223. PMLR, 2017.

[11] M. F. Arlitt and C. L. Williamson. Internet web servers: Workload characterization and performance implications. *IEEE/ACM Transactions on networking*, 5(5):631–645, 1997.

[12] P. Barford, A. Bestavros, A. Bradley, and M. Crovella. Changes in web client access patterns: Characteristics and caching implications. *World Wide Web*, 2(1):15–28, 1999.

[13] P. Barford and M. Crovella. Generating representative web workloads for network and server performance evaluation. In *Proceedings of the 1998 ACM SIGMETRICS joint international conference on Measurement and modeling of computer systems*, pages 151–160, 1998.

[14] D. S. Berger, R. Sitaraman, and M. Harchol-Balter. Adaptsize: Orchestrating the hot object memory cache in a cdn. In *USENIX NSDI*, pages 483–498, March 2017.

[15] D. S. Berger, R. K. Sitaraman, and M. Harchol-Balter. Adaptsize: Orchestrating the hot object memory cache in a content delivery network. In *14th {USENIX} Symposium on Networked Systems Design and Implementation ({NSDI} 17)*, pages 483–498, 2017.

[16] A. Blankstein, S. Sen, and M. J. Freedman. Hyperbolic caching: Flexible caching for web applications. In *2017 USENIX Annual Technical Conference (USENIX ATC 17)*, pages 499–511, 2017.

[17] J. Boyar, M. R. Ehmsen, and K. S. Larsen. Theoretical evidence for the superiority of lru-2 over lru for the paging problem. In *International Workshop on Approximation and Online Algorithms*, pages 95–107. Springer, 2006.

[18] M. Busari and C. Williamson. Prowgen: a synthetic workload generation tool for simulation evaluation of web proxy caches. *Computer Networks*, 38(6):779–794, 2002.

[19] X. Cheng, C. Dale, and J. Liu. Understanding the characteristics of internet short video sharing: Youtube as a case study. *arXiv preprint arXiv:0707.3670*, 2007.

[20] L. Cherkasova. *Improving WWW proxies performance with greedy-dual-size-frequency caching policy*. Hewlett-Packard Laboratories Palo Alto, CA, 1998.

[21] A. Cidon, A. Eisenman, M. Alizadeh, and S. Katti. Cliffhanger: Scaling performance cliffs in web memory caches. In *13th USENIX Symposium on Networked Systems Design and Implementation (NSDI 16)*, pages 379–392, 2016.

[22] T. S. P. E. Corporation. Specweb96 benchmark. https://www.spec.org/web96/.

[23] A. Dan and D. Towsley. An approximate analysis of the lru and fifo buffer replacement schemes. In *Proceedings of the 1990 ACM SIGMETRICS conference on Measurement and modeling of computer systems*, pages 143–152, 1990.

[24] J. Dilley, B. M. Maggs, J. Parikh, H. Prokop, R. K. Sitaraman, and W. E. Weihl. Globally distributed content delivery. *IEEE Internet Computing*, 6(5):50–58, 2002.

[25] O. Eytan, D. Harnik, E. Ofer, R. Friedman, and R. Kat. It's time to revisit {LRU} vs.{FIFO}. In *12th {USENIX} Workshop on Hot Topics in Storage and File Systems (HotStorage 20)*, 2020.

[26] B. Fitzpatrick. Distributed caching with memcached. *Linux journal*, 2004(124):5, 2004.

[27] P. Gill, M. Arlitt, Z. Li, and A. Mahanti. Youtube traffic characterization: A view from the edge. In *Proceedings of the 7th ACM SIGCOMM Conference on Internet Measurement*, IMC '07, page 15–28, New York, NY, USA, 2007. Association for Computing Machinery.

[28] I. Goodfellow, J. Pouget-Abadie, M. Mirza, B. Xu, D. Warde-Farley, S. Ozair, A. Courville, and Y. Bengio. Generative adversarial networks. *Communications of the ACM*, 63(11):139–144, 2020.

[29] D. Grund. *Static Cache Analysis for Real-Time Systems: LRU, FIFO, PLRU*. epubli, 2012.

[30] C. Hogan and D. Epping. *Essential Virtual SAN (VSAN): Administrator's Guide to VMware Virtual SAN*. VMware Press, 2016.

[31] Q. Huang, K. Birman, R. Van Renesse, W. Lloyd, S. Kumar, and H. C. Li. An analysis of facebook photo caching. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*, pages 167–181, 2013.

[32] S. Jiang and X. Zhang. Lirs: An efficient low inter-reference recency set replacement policy to improve buffer cache performance. *ACM SIGMETRICS Performance Evaluation Review*, 30(1):31–42, 2002.

[33] T. Johnson, D. Shasha, et al. 2q: a low overhead high performance bu er management replacement algorithm. In *Proceedings of the 20th International Conference on Very Large Data Bases*, pages 439–450. Citeseer, 1994.

[34] K. Kant, V. Tewari, and R. K. Iyer. Geist: a generator for e-commerce & internet server traffic. In *ISPASS*, pages 49–56, 2001.

[35] R. Karedla, J. S. Love, and B. G. Wherry. Caching strategies to improve disk system performance. *Computer*, 27(3):38–46, 1994.

[36] K. V. Katsaros, G. Xylomenos, and G. C. Polyzos. Globetraff: a traffic workload generator for the performance evaluation of future internet architectures. In *2012 5th International Conference on New Technologies, Mobility and Security (NTMS)*, pages 1–5. IEEE, 2012.

[37] W. King. Analysis of paging algorithms. In *Proc. IFIP 1971 Congress, Ljubljana*, pages 485–490. North-Holland, 1972.

[38] D. Lee, J. Choi, J.-H. Kim, S. H. Noh, S. L. Min, Y. Cho, and C. S. Kim. Lrfu: A spectrum of policies that subsumes the least recently used and least frequently used policies. *IEEE transactions on Computers*, 50(12):1352–1361, 2001.

[39] Q. Li, X. Liao, H. Jin, L. Lin, X. Xie, and Q. Yao. Cost-effective hybrid replacement strategy for ssd in web cache. In *2015 IEEE International Conference on Computer and Information Technology; Ubiquitous Computing and Communications; Dependable, Autonomic and Secure Computing; Pervasive Intelligence and Computing*, pages 1286–1294, 2015.

[40] Z. Lin, A. Jain, C. Wang, G. Fanti, and V. Sekar. Generating high-fidelity, synthetic time series datasets with doppelganger. *arXiv preprint arXiv:1909.13403*, 2019.

[41] B. M. Maggs and R. K. Sitaraman. Algorithmic nuggets in content delivery. *ACM SIGCOMM Computer Communication Review*, 45(3):52–66, 2015.

[42] A. Mahanti, C. Williamson, and D. Eager. Web proxy workload characterization. *Progress Report, Computer Sciences Dept, Univ. of Saskatchewan*, 1999.

[43] R. L. Mattson, J. Gecsei, D. R. Slutz, and I. L. Traiger. Evaluation techniques for storage hierarchies. *IBM Systems journal*, 9(2):78–117, 1970.

[44] N. Megiddo and D. S. Modha. Arc: A self-tuning, low overhead replacement cache. In *Fast*, volume 3, pages 115–130, 2003.

[45] N. Megiddo and D. S. Modha. ARC: A Self-Tuning, low overhead replacement cache. In *2nd USENIX Conference on File and Storage Technologies (FAST 03)*, San Francisco, CA, Mar. 2003. USENIX Association.

[46] D. Mosberger and T. Jin. httperf—a tool for measuring web server performance. *ACM SIGMETRICS Performance Evaluation Review*, 26(3):31–37, 1998.

[47] E. Nygren, R. K. Sitaraman, and J. Sun. The Akamai Network: A platform for high-performance Internet applications. *ACM SIGOPS Operating Systems Review*, 44(3):2–19, 2010.

[48] D. reason and J. Reineke. Toward precise plru cache analysis. In *10th International Workshop on Worst-Case Execution Time Analysis (WCET 2010)*. Schloss Dagstuhl-Leibniz Center for Computer Science, 2010.

[49] A. Rousskov and D. Wessels. High-performance benchmarking with web polygraph. *Software: Practice and Experience*, 34(2):187–211, 2004.

[50] A. Sabnis and R. K. Sitaraman. Tragen: a synthetic trace generator for realistic cache simulations. In *Proceedings of the 21st ACM Internet Measurement Conference*, pages 366–379, 2021.

[51] K. Saini. *Squid Proxy Server 3.1: beginner's guide*. Packt Publishing Ltd, 2011.

[52] M. Z. Shafiq, A. R. Khakpour, and A. X. Liu. Characterizing caching workload of a large commercial content delivery network. In *IEEE INFOCOM 2016-The 35th Annual IEEE International Conference on Computer Communications*, pages 1–9. IEEE, 2016.

[53] J. Summers, T. Brecht, D. Eager, and A. Gutarin. Characterizing the workload of a netflix streaming video server. In *2016 IEEE International Symposium on Workload Characterization (IISWC)*, pages 1–12. IEEE, 2016.

[54] A. Sundarrajan, M. Feng, M. Kasbekar, and R. K. Sitaraman. Footprint descriptors: Theory and practice of cache provisioning in a global cdn. In *Proceedings of the 13th International Conference on emerging Networking EXperiments and Technologies*, pages 55–67, 2017.

[55] A. Sundarrajan, M. Kasbekar, R. K. Sitaraman, and S. Shukla. Midgress-aware traffic provisioning for content delivery. In *USENIX Annual Technical Conference (USENIX ATC 20)*, pages 543–557. USENIX Association, 2020.

[56] W. Tang, Y. Fu, L. Cherkasova, and A. Vahdat. Medisyn: A synthetic streaming media service workload generator. In *Proceedings of the 13th international workshop on Network and operating systems support for digital audio and video*, pages 12–21, 2003.

[57] M. Wajahat, A. Yele, T. Estro, A. Gandhi, and E. Zadok. Analyzing the distribution fit for storage workload and internet traffic traces. *Performance Evaluation*, 142:102121, 2020.

[58] A. Williams, M. Arlitt, C. Williamson, and K. Barker. Web workload characterization: Ten years later. *Web content delivery*, pages 3–21, 2005.

[59] S. Xu, M. Marwah, and N. Ramakrishnan. Stan: Synthetic network traffic generation using autoregressive neural models. *arXiv preprint arXiv:2009.12740*, 2020.

[60] J. Yang, Y. Yue, and K. V. Rashmi. A large scale analysis of hundreds of in-memory cache clusters at twitter. In *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*, pages 191–208. USENIX Association, Nov. 2020.

[61] Q. Yang, H. H. Zhang, and T. Li. Mining web logs for prediction models in www caching and prefetching. In *Proceedings of the seventh ACM SIGKDD international conference on Knowledge discovery and data mining*, pages 473–478, 2001.
[62] Y. Yin, Z. Lin, M. Jin, G. Fanti, and V. Sekar. Practical gan-based synthetic ip header trace generation using netshare. In *Proceedings of the ACM SIGCOMM 2022 Conference*, pages 458–472, 2022.
[63] J. Yiu. *The Definitive Guide to ARM® Cortex®-M0 and Cortex-M0+ Processors*. Academic Press, 2015.

10 Description of the cache algorithms

10.1 Popularity and size based admission algorithms

(1) **ThLRU-z.** The ThLRU-z algorithm only admits objects that are smaller than the specified threshold z.
(2) **Bloomfilter-n.** The Bloomfilter-n algorithm only admits objects on their n^{th} access [41]. We set n to the values 2 and 3 in our experiments and are abbreviated as Bloom-2 and Bloom-3, respectively, in Figure 9.
(3) **ThLRU-Prob.** ThLRU-Prob admits objects with a probability $e^{-z/c}$, where z is the size of the requested object and c is a user defined parameter. We set $c = 500KB$.
(4) **ThLRU-z/Bloomfilter-n.** The ThLRU-x/Bloomfilter-y admits objects that have a size smaller than x. Further, these objects are admitted only on their n^{th} access. We set $z = 8MB$ and $n = 2$ for our experiments.
(5) **Adaptsize.** Adaptsize is an adaptive size-aware cache admission policy that dynamically tunes the size threshold to maximize the RHR. Adaptsize admits objects with a probability $e^{-z/c}$, where z is the size of the requested object and c is a parameter that is learnt dynamically by solving a markov model [15].

10.2 Popularity and size based eviction algorithms

(1) **GDSF.** The Greedy-Dual-Size-Frequency cache algorithm assigns a priority to each object in the cache. The priority is computed as a function of last access time, frequency and the size of the object. The GDSF algorithm evicts an object from the cache with the least priority. The GDSF has been shown to provide a better performance than LRU for proxy web caches [20]. It is currently implemented in Squid – an open source proxy cache. [51].
(2) **LFU.** The Least-Frequently-Used cache algorithm evicts an object that is requested the least number of times. Despite its popularity, LFU is not used in practice as it performs poorly.
(3) **LRFU.** The Least-Recently/Frequently-Used cache algorithm uses both the recency as well as the frequency of the. objects to make an eviction decision. [38]. LRFU overcomes the pitfalls of the LFU algorithm.
(4) **ARC.** The Adaptive replacement cache algorithm keeps track recently used and frequently used objects and recent eviction history for both. It has been shown to provide a better performance than LRU on a wide range of workloads [45]. ARC is used in many production systems such as Sun Microsystems's ZFS [4], VMware's vSAN [30], OpenZFS [2].
(5) **2Q.** The 2Q cache algorithm maintains two caches. The first cache uses FIFO eviction algorithm and the second uses LRU. On a cache miss, the requested object is added to the FIFO cache and on a subsequent request to the object it is moved to the LRU list [33].

10.3 Recency based eviction algorithms

(1) **FIFO.** The First In First Out eviction algorithm evicts objects from the cache in the order they are inserted, without any regard to the number of accesses or the recency of the objects. FIFO is easy to implement and has been shown to provide similar hitrates as LRU on a variety of production workloads [25]. Further, FIFO provides a better cache performance on SSDs as compared to other cache algorithms [39]. Hence, FIFO is widely used in practice.
(2) **RANDOM.** The RANDOM eviction algorithm evicts a random object from the cache on inserting a new object into the cache. It is easy to implement as it does not maintain access information of the objects in the cache. The RANDOM eviction algorithm and its variants are studied extensively [6, 16, 37] and widely used. For example, in ARM processors [63].
(3) **SLRU (and S4LRU).** The Segmented LRU cache eviction algorithm segments the cache into an equal sizes upper and lower segments that independently use the LRU eviction algorithm. On a cache miss, the requested object is first inserted into the lower segment and moved to the upper segment on a subsequent access. The object that is evicted from the upper segment is inserted into the lower segment. S4LRU operates similar to SLRU but segments the cache into 4 equal size segments. SLRU and S4LRU have been used in Facebook photo caching [31].
(4) **PLRU.** The Pseudo-LRU caching algorithm approximates LRU. PLRU stores objects in the cache as leaf nodes of a binary tree. The non-leaf nodes maintain pointers to the nodes that have not been recently used. The pointers are updated on every access and are used to find a leaf node that has not been recently used. The object in the leaf node is evicted. PLRU is used in TC1798 CPU and POWERPC variants (MPC603E, MPC755, MPC7448) [48].
(5) **CLIFFHANGER.** CLIFFHANGER [21] is a set-associative cache and allocates a segment of the cache for each set of objects. The size of each segment is determined dynamically using the access patterns of the objects across the sets. Each segment uses the LRU eviction algorithm.