TRAGEN: A Synthetic Trace Generator for Realistic Cache Simulations

Anirudh Sabnis
Univ. of Massachusetts Amherst

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

Abstract

Traces from production caching systems of users accessing content are seldom made available to the public as they are considered private and proprietary. The dearth of realistic trace data makes it difficult for system designers and researchers to test and validate new caching algorithms and architectures. To address this key problem, we present TRAGEN, a tool that can generate a synthetic trace that is "similar" to an original trace from the production system in the sense that the two traces would result in similar hit rates in a cache simulation. We validate TRAGEN by first proving that the synthetic trace is similar to the original trace for caches of arbitrary size when the Least-Recently-Used (LRU) policy is used. Next, we empirically validate the similarity of the synthetic trace and original trace for caches that use a broad set of commonly-used caching policies that include LRU, SLRU, FIFO, RANDOM, MARKERS, CLOCK and PLRU. For our empirical validation, we use original request traces drawn from four different traffic classes from the world's largest CDN, each trace consisting of hundreds of millions of requests for tens of millions of objects. TRAGEN is publicly available and can be used to generate synthetic traces that are similar to actual production traces for a number of traffic classes such as videos, social media, web, and software downloads. Since the synthetic traces are similar to the original production ones, cache simulations performed using the synthetic traces will yield similar results to what might be attained in a production setting, making TRAGEN a key tool for cache system developers and researchers.

ACM Reference Format:
Anirudh Sabnis and Ramesh K. Sitaraman. 2021. TRAGEN: A Synthetic Trace Generator for Realistic Cache Simulations. In *ACM Internet Measurement Conference (IMC '21), November 2–4, 2021, Virtual Event, USA.* ACM, New York, NY, USA, 14 pages. https://doi.org/10.1145/3487552.3487845

1 Introduction

The volume and diversity of the digital content delivered over the Internet is growing at a rapid pace. Such content include videos, images, webpages, 360° videos, and software downloads. Much of this content is delivered by large distributed networks of caches operated by content delivery networks (CDNs). CDNs deploy hundreds of thousands of servers in thousands of data centers around the world. When a user accesses an object, say a web page or a video segment, the user's request is routed to a proximal CDN server. If that server has the requested object in its cache, a *cache hit* is said to have occurred, and that object is served to the user. Otherwise, a *cache miss* is said to have occurred, the CDN server fetches that object from an (usually distant) origin server over the WAN, and then serves it to the user.

Maximizing cache hits. Obtaining cache hits is the desired goal of caching as the user sees a faster response when the requested object can be downloaded from a proximal cache. A cache miss is undesirable since it causes large latencies due to having to fetch the requested object from an origin server over the WAN. Further, a cache miss causes additional WAN traffic between the CDN's cache and the content provider's origin, increasing the bandwidth cost for both parties.

The key metric for evaluating cache efficiency is its hit rate that come in two flavors. The *request hit rate (RHR)* is the fraction of requests that were cache hits, whereas the *byte hit rate (BHR)* is the fraction of bytes that were served from cache. The former metric correlates with average user performance. The latter correlates with the additional WAN traffic required to serve the cache misses, hence indicative of the additional bandwidth cost. RHR weights all hits equally, whether or not the requested object is big or small. But, the BHR weights each hit by the size of the requested object.

Content caching research. Content caching is ubiquitous and is central to the functioning of the Internet ecosystem. Not surprisingly, algorithms for maximizing the hit rate of a cache has been a subject of intense research over the past few decades. The research has resulted in a vast and growing literature of how to admit and evict objects, so as to optimize the efficiency of the cache [9, 28, 29, 31, 42]. The traditional caching policies LRU, FIFO, LFU, and RANDOM are still commonly used [19, 25] and several variants of these policies have been proposed to improve the cache performance [2, 42, 57]. Adaptive algorithms for caching content in in-memory caches has been explored in [7]. Caching has also been used to balance the load across the backend servers in a cluster [22] and reduce the latency variability in the requests [6], amongst several other applications. There has also been research on using deep-learning to improve the caching policies [34, 44]. With time varying content popularity, new content being published at very frequent rates, and the increasing diversity of the content, caching remains an active research area where new caching policies and architectures are frequently proposed and studied.

Need for realistic traces from production caches. The key enabler of caching system research and development is *cache simulations*. Developers in industry routinely modify caching policies and simulate their impact. Researchers propose new caching policies and architectures and validate their ideas using cache simulations. The efficacy of a caching system greatly depends on the prevailing patterns of how/what objects are requested by users, such patterns

IMC '21, November 2–4, 2021, Virtual Event, USA
© 2021 Copyright held by the owner/author(s). Publication rights licensed to ACM.
ACM ISBN 978-1-4503-9129-0/21/11…$15.00
https://doi.org/10.1145/3487552.3487845

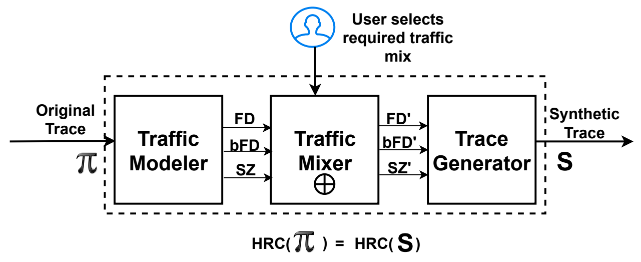


Figure 1: System diagram of TRAGEN.

can be provided by traces that consist of sequences of user requests for objects. In order to design and validate new caching policies and system designs, researchers and developers need such traces to empirically predict the likely performance of the caching system in the field. But, caching research is seriously hampered by the dearth of realistic traces from production caching systems. Production traces are considered private and proprietary and increasingly hard to obtain. Even when obtained, original production traces cannot be made available publicly for other researchers to replicate the work.

Another key obstacle for using original production traces for cache simulations is that the number of caching scenarios that need to be simulated is often large. Consider a CDN server in the field that must cache and serve content that belongs to several different traffic classes, say videos, web, downloads, and images from multiple content providers, in relative proportions that drastically vary over time. It is not possible to find actual production traces for every such scenario that could happen in the field. Further, given that the production traces are voluminous and most developers do not have access to them for reasons of privacy, the ability to generate realistic synthetic traces is important to test the system across several possible traffic mixes and load scenarios.

Our approach. To overcome the challenge posed by the dearth of realistic traces, we propose a tool called TRAGEN (c.f., Figure 1) that generates synthetic traces for a wide range of caching workloads that can be specified by the user of the tool. Formally, a *trace* is a sequence of user requests where each request is a 3-tuple consisting of the time at which the request was made by a user, the unique identifier (say, url) of the object that was requested, and the size of the object. TRAGEN produces a *synthetic trace* S that is "similar" to an *original trace* Π in the sense that the two traces would produce similar hit rates in a cache simulation. Since the synthetic trace has no information about actual objects accessed by users, it can be generated in-place and used for cache simulations without having to transport and store voluminous and sensitive production traces.

TRAGEN can also produce a synthetic trace that is similar to a *mix* of traffic classes, where each traffic class is represented by an original production trace. For instance, it could produce a synthetic trace that is a mix of 10 Gbps of video request traffic and 5 Mbps of download request traffic. The ability to generate synthetic traces for user-specified traffic mixes allows the developer to test their caching systems on a wide variety of possible scenarios likely to occur in the field.

Hit rate curves and trace similarity. The *hit rate curve (HRC)* of a caching system on a given trace is the hit rate achieved by the cache when serving requests in time sequence from the trace, expressed as a function of cache size (c.f., Figure 3). The HRC depends on the caching policy used by the caching system. Further, RHR and BHR may yield different HRCs and we refer to them as rHRC and bHRC respectively.

Our notion of trace similarity is defined in terms of the HRC. Given a caching system that implements a given caching policy (say, LRU), we state that two traces Π and S are *similar* if the HRC of the caching system for trace Π is similar to the HRC of the system for trace S. Thus, cache simulations on traces S and Π would yield similar results, allowing trace S to be used in the simulation instead of trace Π.

TRAGEN architecture. Our trace generator consists of three main modules shown in Figure 1 and are described below.

1) The *traffic modeler* runs periodically on the original traces Π collected from the production system. The original trace could be all the requests served by a given CDN cache over a period of time, such a trace would be a mix of requests from different *traffic classes*, e.g., videos from CNN, or images from Amazon, or software downloads from Microsoft. Alternately, trace Π may contain traces from the production system of a single traffic class. The output of the traffic modeler is a succinct "model" of each traffic class, such a model captures the caching properties of that traffic class. A commonly-used model in large CDNs such as Akamai are footprint descriptor (FD) described in [50] that model the rHRC of the original trace. As FD does not capture byte hit rates, we enhance the FD to a byte-weighted footprint descriptor (bFD) described in Section 2. The traffic modeler also computes the object size distribution (SZ) of each traffic class. Since the traffic modeler works on voluminous production traces collected from the field, it runs infrequently (say, once a week) to create the traffic class models.

2) Given the FDs, bFDs and SZ of each traffic class, the *traffic mixer* component allows the user to specify the mix of traffic that they would like to simulate, e.g., 10 Gbps of video traffic from Amazon mixed with 5 Gbps of download traffic from Microsoft. The traffic mixer uses footprint descriptor calculus to compute the FD or bFD of the traffic class mix.

3) Finally, the *trace generator* uses the FD or bFD of the required mix of traffic classes to generate a synthetic trace that fits requirements. That is, the synthetic trace is similar to the original production traces of the required traffic mix.

Our contributions. Our main contribution is a tool that is publicly available[1] to the research and development community. The tool will be seeded with realistic footprint descriptor models for traffic classes hosted on Akamai's production CDN, allowing users to generate synthetic traces for their experiments for varying caching scenarios according to their requirements. We prove that TRAGEN produces synthetic traces that have a similar hit rate curve as the original trace for caches that use the Least-Recently-Used (LRU) policy. Further, we empirically validate TRAGEN by establishing the similarity of the synthetic and original traces. In particular, we compare the two traces by computing their hit rates and eviction ages.

1) We show that the average difference between the rHRCs and bHRCs of the synthetic and original traces on a LRU cache is 3e-06 and 3.2e-06, respectively, across all traces and cache sizes in our

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

evaluation. Thus, the hit rate curves are nearly identical for the two traces.

2) Using cache simulations, we show that the synthetic traces produced by TRAGEN will yield similar hit rates as the original traces for commonly-used caching policies that include LRU, SLRU [27], FIFO [20, 27], RANDOM [1, 33, 58], PLRU [46], MARKERS [1, 37], and CLOCK [16]. In particular, we show that the synthetic trace yields a RHR (resp., BHR) that differs from the RHR (resp., BHR) of the original trace by 1.5% (resp., 1%) on average across all cache sizes, caching policies and traces in our evaluation.

3) We show that on an average the eviction age of the synthetic trace differs from the original trace by 1.8% on a LRU cache across all cache sizes in our evaluation.

Limitations of TRAGEN. TRAGEN is guaranteed to produce synthetic traces that have similar caching behavior to the original traces for the set of caching policies that we could theoretically or empirically validate. Based on our work, we *conjecture* that TRAGEN will work well for the class of policies that primarily use criteria related to recency of access for eviction, many commonly-used policies belong in this class.

The main limitation of our work is that we offer no explicit guarantees for *arbitrary* caching policies that may use entirely different criterion. In fact, it is not clear if there exists *universal* trace generators that can provably work for arbitrary caching policies, while still producing synthetic traces that are different from the original. The space of possible caching policies is large and include ones that control both the admission and eviction of content into cache, while our validated cache policies perform only eviction. Our current approach is to continue validating TRAGEN for more policies and making changes to the algorithms as needed to widen the scope. We expect this evolution to continue as more developers and researchers use our tool.

Relation to prior work. Prior work in this area have proposed synthetic workload generators for Web traffic [5, 12, 17, 30, 32, 43, 47]. These tools from the past decades, however, do not cater to multiple traffic classes and traffic class mixes seen in modern content caching scenarios. Further, they lack a provable guarantee that the generated traces have the same caching properties as the original ones. They also do not support traffic mix scenarios that are common in content caching, an important requirement since a CDN cache is shared across multiple diverse traffic classes. Further, prior work assume a fixed object catalogue which is not true in practice, since new content is continually generated, and old content fall out of use, e.g., the news story on the front page of CNN. In particular, as we show in Section 5.3, the prior work that use the LRUSM algorithm [5, 40] do not produce traces that have the same caching properties as the original trace. Finally, the prior work often consider caches of small size (in MBs) that do not scale to modern caches that are many magnitudes larger (in TBs).

Roadmap. In Section 2, we describe the Traffic Modeler that captures the cache properties of the production traces. In Section 3, we describe the Traffic Mixer that computes a model for a user specified traffic mix. In Section 4.1, we describe our Trace Generator and provide formal guarantees for its correctness and in Section 4.2 we describe our tool that implements the Trace generator and will be made available for public use. In Section 5, we provide empirical results and also show that alternate approaches for trace generation do not perform well. We end with related work in Section 6 and conclude in Section 7.

2 Traffic modeler

In this section, we describe traffic classes and the footprint descriptor model that is used to capture the caching properties of a given traffic class. To model the RHR we leverage the tool of Footprint Descriptor (FD) from the work in [50] and extend the model to a byte-weighted Footprint Descriptor (bFD) to capture the BHR. The output of the traffic modeler is a *model* of the original production trace that is a three-tuple consisting of its FD, bFD, and its object size distribution. The model is a succinct representation of the caching characteristics of the voluminous original trace from which it is derived.

2.1 Traffic classes

The content accessed by users on the Internet is very diverse, each with a unique set of characteristics. For the purposes of cache management, the content is usually bucketed into traffic classes. A traffic class is a type of content from a content provider and is treated as a unit in the cache provisioning process [51], for example, media (i.e., videos) from Hulu, software downloads from Microsoft, images from Amazon, and web pages from CNN. Each traffic class has a distinct set of characteristics such as its object size distribution, specific access patterns, and its popularity distribution. Further, each traffic class may need a different cache size in order to provide a minimum hit rate guarantee. The request HRCs (rHRC) and byte HRCs (bHRC) of the traffic classes Download, Media, Web and Images is depicted in Figure 2a and Figure 2b, respectively. The rHRC for the image traffic shows that despite providing a large cache space we cannot obtain a RHR of above 0.6. This may be attributed to the fact that a significant number of image objects are accessed only once and the cache incurs a compulsory miss on a request for these objects. Such an access pattern may arise with a large product catalog where a significant fraction of products are unpopular and their images are seldom accessed. Thus, the caching properties of each traffic class is different from the other. Figure 2c shows the variability in the object sizes for the various traffic classes. As expected, objects in the Media traffic class tend to be larger as compared those in the Web traffic class.

2.2 Footprint descriptors (FD)

A footprint descriptor (FD) is a succinct space-time representation of a trace from which its rHRC and other caching properties can be derived. A traffic class can be characterized by collecting typical original traces of that class from the production system and computing their FDs. FDs were first proposed in [50] and are now used in production CDNs to provision traffic classes to servers [51].

Our definition and presentation of FDs closely follows [50]. Let trace $\Pi = \{r_1, \ldots, r_n\}$, be a sequence of requests, where each request r_i is a tuple $\langle t_i, o_{id}, z_i \rangle$ of timestamp, object identifier and object size. Now, let $\theta = \{r_i, \ldots, r_j\}$, where $i < j$, be a request subsequence consisting of consecutive requests in Π. The subsequence θ is denoted as a *reuse request subsequence* if the requests r_i and r_j are made for the same object that is not requested elsewhere in θ.

A FD of a trace Π is a tuple $\langle \lambda, P^r(s,t), P^a(s,t) \rangle$ where (i) λ is the request rate (number of requests per second) of Π ; (ii) $P^r(s,t)$ is the

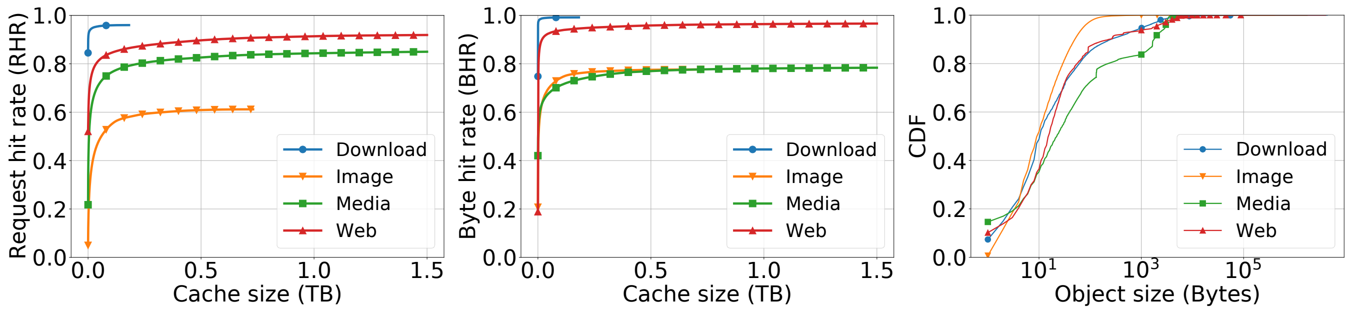


Figure 2: For the different traffic classes in TC trace (Table 2) (a) Variability in request HRCs; (b) Variability in the byte HRCs; (c) Variability in object size distribution (SZ)

reuse-subsequence descriptor function that captures the probability that a *reuse request subsequence* θ of Π contains s unique bytes and is of duration t seconds. (iii) $P^a(s,t)$ is the all-sequence descriptor function that captures the probability that *any request subsequence* θ of Π contains s unique bytes and is of duration t seconds. Note that $P^r(s,t)$ considers only the reuse subsequences, whereas $P^a(s,t)$ considers all possible subsequences of Π. The number of unique bytes s in θ is the sum of the sizes of the unique objects in θ, and the duration t is the difference in the timestamp $(t_j - t_i)$ of the last request r_j and the first request r_i. Note that the request sequence θ that starts at the beginning of the trace and ends in the first request for an object is considered a reuse subsequence with infinite unique bytes and infinite duration. The number of unique bytes in a reuse request subsequence is also known as *stack distance* [41]. The following theorem is from [50].

THEOREM 1. *Let rHRC(s) be the request hit rate of trace Π for an LRU cache of size s. The rHRC(s) is computed from the FD of Π as follows.*

$$rHRC(s) = \sum_t \sum_{s' \le s} P^r(s', t).$$

Observe that the expression in RHS sums the reuse sequence distribution function $(P^r(s',t))$ across all possible time durations. In essence, the RHS captures the probability that a reuse request subsequence, $\theta = \{r_i, \ldots, r_j\}$, contains at most s unique bytes, thus incurring a cache hit on the request r_j. Note that if θ contained more than s bytes, the LRU policy would have evicted the object requested by r_i when the request r_j is made, resulting in a miss.

FD calculus. A key property of FDs is that there is an efficient calculus to evaluate the cache properties of any traffic class mix. For instance, given footprint descriptors of two traces Π_1 and Π_2, the calculus can be used to compute the FD of the traffic mix of Π_1 and Π_2 obtained by interleaving the two traces by their timestamps. The computation of the FD of the traffic mix uses convolution and can be computed quickly using a Fast Fourier Transform [50]. When synthetic traces are generated for a traffic mix, TRAGEN uses the FD calculus to compute the FD of traffic class mix and then uses this FD to generate a synthetic trace for the mix.

2.3 Byte-weighted footprint descriptor (bFD)

RHR can be derived from FD as shown in Theorem 1, but the BHR cannot be derived from it. However, we know from practice that the RHR and BHR can be significantly different. Consider the following scenario. Let the smaller objects in a trace exhibit higher temporal locality as compared to the large objects, thus incurring more cache hits as compared to the larger objects. We can then expect the BHR of the trace to be smaller than the RHR. In fact, we observe a similar difference in the rHRC and bHRC of the VIDEO trace (Table 1) in Figure 3. Thus, to capture the BHR properties of a trace, we define a new type of footprint descriptor called Byte-weighted Footprint Descriptor (bFD).

A bFD operates on a byte sequence as opposed to a request sequence. Let $\Pi^B = \{b_{11}, \ldots, b_{1z_1}, \ldots, b_{n1}, \ldots, b_{nz_n}\}$ be the byte sequence that is obtained from a request trace $\Pi = \{r_1, \ldots, r_n\}$ by replacing each request r_i by the sequence of bytes $b_{i1}, \ldots, b_{iz_i}$ in its requested object, where byte b_{ij} corresponds to the j^{th} byte of request $r_i \in \Pi$ and z_i is the size of the requested object. Now, let $\beta = \{b_{ij}, \ldots, b_{kl}\}$, where $i < k$, correspond to a *byte subsequence* in Π^B. The byte subsequence β is called a *reuse byte subsequence* if the first byte (b_{ij}) and the last byte (b_{kl}) in β correspond to the same byte that does not occur elsewhere in β.

A bFD is a tuple $\langle \lambda, P^{rb}(s,t), P^{ra}(s,t) \rangle$ where (i) λ is the traffic rate i.e., the number of *bytes* requested per second. (ii) $P^{rb}(s,t)$ is the reuse byte subsequence descriptor function that captures the joint probability that a *reuse byte subsequence* consists of s unique bytes and is for a duration t seconds, and, (iii) $P^{ra}(s,t)$ is the all byte subsequence descriptor function that captures the joint probability that a *byte subsequence* consists of s unique bytes and is for a duration t seconds.

The computation of $P^{rb}(s,t)$ is done as follows. We maintain a counter $C(s,t)$ that counts the number of reuse byte subsequences

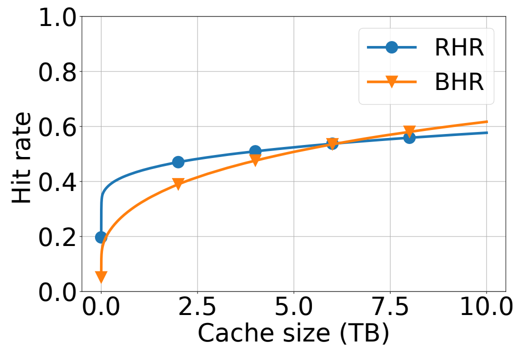


Figure 3: The rHRC and bHRC for the VIDEO trace

that contain s unique bytes and is of duration t seconds. We can obtain $C(s,t)$ by enumerating all reuse byte subsequences θ of Π and incrementing the appropriate counters. To understand the computation of $C(s,t)$, consider a reuse byte subsequence $\beta = \{b_{i1}, \ldots, b_{j1}\}$ that corresponds to a reuse request subsequence $\theta = \{r_i, \ldots, r_j\}$. If the number of unique bytes in θ is s, and the duration of θ is t, then counter $C(s,t)$ is incremented z_i times, where z_i is the size of request r_i. Thus, the computation accounts for all the bytes of request r_i, i.e., all reuse byte sequences of the form $\beta = \{b_{ik}, \ldots, b_{jk}\}$, $1 \leq k \leq z_i$. By updating counter $C(s,t)$ for all possible reuse sequences θ of Π we obtain $P^{rb}(s,t) = \frac{C(s,t)}{|\Pi^B|}$, where $|\Pi^B|$ is the total number of bytes in Π^B.

The following theorem applies to LRU whose variants are widely used in many production systems like Akamai [38].

THEOREM 2. *Let bHRC(s) be the byte hit rate of trace Π for an LRU cache of size s. The bHRC(s) is computed from the bFD of Π as follows.*

$$bHRC(s) = \sum_t \sum_{s' \leq s} P^{rb}(s', t).$$

PROOF. Our proof is similar to the proof of Theorem 1 in [50]. Consider a byte reuse sequence $\beta = \{b_{ij}, \ldots, b_{kl}\}$, where b_{ij} and b_{kl} are requests for the same byte. For a cache size s running LRU, the request for the byte b_{kl} is a hit if and only if the number of unique bytes accessed in β is less than the cache size s. The probability of the occurrence is obtained by the expression $bHRC(s) = \sum_t \sum_{s' \leq s} P^{rb}(s', t)$. □

Time complexity. Using efficient data structures and stack based algorithms FD and bFD can be computed in $O(N \log m)$, where N is the length of the trace and m is the number of unique objects in the trace [3].

3 Traffic mixer

We describe the component that computes the model of a user-specified traffic mix from the models of the individual traffic classes. A traffic mix is specified as a list of traffic classes $\Gamma = \{\tau_1, \ldots, \tau_n\}$ and a weight vector $\mathbf{W} = \{w_1, \ldots, w_n\}$ that describes the traffic contribution of each traffic class. The traffic contribution can be specified using the required request-rate (requests/second) or the required byte-rate (GBps) for each traffic class. The traffic mixer uses the footprint descriptor calculus [50] to compute the FD_{mix} (resp. bFD_{mix}), i.e., FD (resp., bFD) of the traffic mix. Further, the traffic mixer also computes the object size distribution of the traffic mix.

3.1 Footprint descriptor calculus

We will now describe the FD calculus that is described in [50] and show that it extends to bFD as well. Consider two traces Π_1 and Π_2 and their respective footprint descriptors $FD_1 = \langle \lambda_1, P_1^r(s,t), P_1^a(s,t) \rangle$ and $FD_2 = \langle \lambda_2, P_2^r(s,t), P_2^a(s,t) \rangle$. Let Π be the trace that is formed by interleaving Π_1 and Π_2 by time. A key observation that facilitates the calculus is that for a subsequence θ of Π, of duration t and unique bytes s, some s_1 bytes could be from Π_1 and the rest $s - s_1$ bytes from Π_2, assuming Π_1 and Π_2 contain disjoint objects (The disjoint object assumption holds in many common situations, including when Π_1 and Π_2 are different traffic classes). Thus, to compute a descriptor function $P(s|t)$ for Π from the descriptor functions $P_1(s|t)$ and $P_2(s|t)$ for Π_1 and Π_2 respectively, the convolution operator is used to enumerate and add the probabilities of all possible ways of obtaining s_1 unique bytes from Π_1 and the remaining $s - s_1$ unique bytes from Π_2. Thus,

$$P(s|t) = P_1(s|t) * P_2(s|t)$$
$$= \sum_{s_1=0}^{S-s_1} P_1(s_1|t) P_2(S - s_1|t),$$

where * is the convolution operator. We can see that the same argument follows for the byte sequence Π^B and hence the footprint descriptor calculus that works for a FD also works for a bFD. We will briefly describe the basic operations in the calculus using byte sequences.

(i) *Addition.* Given two byte sequences Π_1^B and Π_2^B and their bFDs, bFD_1 and bFD_2, if Π^B represents a sequence with Π_1^B and Π_2^B interleaved by time, then the addition operator provides the byte footprint descriptor bFD_{mix} of the interleaved sequence Π^B. By using the Fourier transform to evaluate the convolution operator, the addition operator runs in $O(TS \log S)$ time, where T and S are the number of time and stack distance buckets in $P^{rb}(s,t)$.

(ii) *Subtraction.* Given a byte sequence Π^B and its corresponding bFD, the subtraction operator provides a means to compute the bFD of the trace that is formed by removing all the requests (i.e., corresponding bytes) that are made for a subset of objects from Π^B. If Π_1^B is the sequence that is removed from Π^B, the byte footprint descriptor of the resultant sequence $\Pi_2^B = \Pi^B \ominus \Pi_1^B$ can be computed using the subtraction operator. By using the inverse fourier transform, the subtraction operator runs in $O(TS \log S)$ time.

(iii) *Scaling.* Given a byte sequence Π^B and its corresponding bFD, the scaling operator provides a means to compute the byte footprint descriptor of a trace whose traffic rate is intensified or rarefied i.e., the traffic volume λ is altered. The scaling operator runs in $O(TS)$ time.

We experimentally verify that the calculus works for bFD. We consider the trace EU that is described in Table 3. We compute bFD of each individual traffic class in the trace and use the calculus to find the bFD of the traffic mix. We verify that the bHRC computed using the trace that corresponds to the traffic mix aligns with the bHRC predicted by the calculus. The result of the experiment is depicted in Figure 4. The Media0+Media1(trace) curve depicts the bHRC computed from the subsequence of trace EU that consists of objects from the Media0 and Media1 traffic class. The Media0+Media1(calculus) curve depicts the bHRC computed using the addition operator from the calculus.

3.2 Object size distribution of a traffic mix

A traffic mix is specified by a list of traffic classes $\Gamma = \{\tau_1, \ldots, \tau_n\}$ and a weight vector $\mathbf{W} = \{w_1, \ldots, w_n\}$ that specifies the traffic contribution of each traffic class in GBps. To compute the object size distribution of a traffic mix, we first compute an object weight vector $\mathbf{O}$ that provides us the ratio of the number of objects per traffic class that is to be present in the produced synthetic trace.

For each traffic class, we first compute the expected *unique byte rate* (U_i), which is the number of unique bytes requested per unit time. Now, we can compute the expected *unique object rate* i.e., the

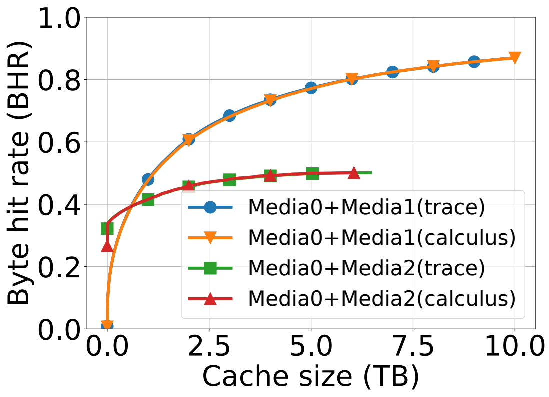


Figure 4: The bHRC for the traffic mixes in the EU trace as predicted by the calculus aligns with the original.

number of unique objects requested per unit time, for the traffic class as U_i/s_i^{avg}. Here, s_i^{avg} is the average object size of the traffic class. The ratio of unique object rate across the specified traffic classes gives us the object weight vector $\mathbf{O}$. The procedure is described in Algorithm 1.

The computation of unique byte rate U_i is done as follows. Let U_{orig} be the expected unique byte rate of a traffic class τ_i and bFD of τ_i is a tuple $\langle \lambda, P^{rb}(s,t), P^{ab}(s,t) \rangle$. As described in lines 6-10 of Algorithm 1, we can compute U_{orig} from the bFD using $U_{orig} = \sum_t \sum_s P^{rb}(s,t) \left(\frac{s}{t}\right)$. Recall that $P^{rb}(s,t)$ is a joint probability distribution that a reuse byte subsequence has s unique bytes and duration t seconds. Now, U_i can be computed as $\frac{w_i}{\lambda} \times U_{orig}$ (line 13), where w_i is the traffic volume specified by the user and λ is the traffic volume of the original trace.

The object size distribution, SZ, of the traffic mix can be computed by weighting the SZ of individual traffic classes by a weight proportional to its contribution in the object weight vector.

4 Trace Generator

The trace generator produces a synthetic trace with same request hit rate curve (rHRC) *or* byte hit rate curve (bHRC) as the original trace or a user-specfied traffic mix.

4.1 Trace generation algorithm

Algorithm 2 performs trace generation and is described below.
Input. The algorithm is provided with a model of a traffic class or traffic mix:
(1) a FD $\langle \lambda, P^{r}(s,t), P^{a}(s,t) \rangle$,
(2) a bFD $\langle \lambda, P^{rb}(s,t), P^{ab}(s,t) \rangle$,
(3) an object size distribution SZ, and
(4) the number of requests N to be generated.
Output. A synthetic trace $S = \{r_1, \ldots, r_N\}$, where each $r_i = \langle t_i, o_i, z_i \rangle$ is a tuple of timestamp, object id, and object size.
Note: The algorithm uses either the FD or bFD depending on whether RHR or BHR is required. Let $P(s) = \sum_t P^{r}(s,t)$ or $P(s) = \sum_t P^{rb}(s,t)$ be the marginal distribution from the FD or bFD, depending on the input. .

Algorithm 1 Object Weight Estimator

```
1:  Input. (i) A list of byte footprint descriptors $bFD = \{bFD_1, \ldots, bFD_n\}$
    of each traffic class $\tau_i$, (ii) a weight vector $W = \{w_1, \ldots, w_n\}$, where
    $w_i$ specifies the traffic volume (in GBps) for the traffic class $\tau_i$, and
    (iii) size distribution of the objects $SZ = \{SZ_1, \ldots, SZ_n\}$ of each traffic
    class $\tau_i$.

2:  Output. An object weight vector $\mathbf{O} = \{o_1, \ldots, o_n\}$ that specifies the
    ratio of number of objects per traffic class.

3:  $\mathbf{O} \leftarrow \{\}$
4:  for $bFD_i \in bFD$ do // $FD_i = \langle \lambda, P^{rb}(s,t), P^{ab}(s,t) \rangle$
5:      $U_{orig} = 0$
6:      for $s \in S$ do // $S$ be the stack distance buckets in $P^{rb}(s,t)$.
7:          for $t \in T$ do // $T$ be the time buckets in $P^{rb}(s,t)$.
8:              $U_{orig} += P^{rb}(s,t).\frac{s}{t}$
9:          end for
10:     end for
11:     Let $s_i^{avg}$ be the average object size of the traffic class $\tau_i$
12:     Let $\lambda_i$ be the traffic volume (in GBps) of traffic class $\tau_i$
13:     $U_i = \frac{w_i}{\lambda_i} \times U_{orig}$
14:     $o_i \leftarrow \frac{U_i}{s_i^{avg}}$
15:     Append $o_i$ to $\mathbf{O}$
16: end for
17: return $\mathbf{O}$
```

Initialization. An empty list C that represents a cache is initialized in line 5. Through lines 7-11, we iteratively create new objects, assign them a size that is sampled from the object size distribution SZ, and append them to the list. We repeat till the sum of the sizes of the objects exceeds the maximum stack distance in $P(s)$. The maximum stack distance is the maximum number of unique bytes in any request or byte subsequence of the original trace. Each entry in C is thus a tuple $\langle o_{id}, z \rangle$ of object id and size.
Synthetic trace generation. The trace generation algorithm runs from line 17 to line 27. In each iteration i, the object at the first position in the list C, say $o = \langle o_{id}, z \rangle$, is appended to the trace S that is being produced. Now, a stack distance s_i is sampled from $P(s)$ and the list is manipulated based on the value of s_i. There are two cases:

(i) if s_i is finite (lines 17-21): the object o is removed and re-inserted back at a position j in C, by moving the objects at positions $\geq j$ by a step. The location j is decided as follows. We find the *first* position in the list, say k, such that the sum of the sizes of the objects at locations from 1 to k in C is greater than s_i. Let the sum of the sizes of the objects be S_k and the size of the object at position k be z_k. Now, if $s_i \leq S_{k-1} + \frac{z_k}{2}$, i.e., the stack distance s_i falls on the first half of the object at position k, then j is set to $k-1$ and set to k otherwise. This ensures that object o is re-inserted at a location that is as close as possible to stack distance s_i.

(ii) if s_i is ∞ (lines 21-25): object o is removed from C and a new object o' is inserted at the end of the list.

Now in line 28, a timestamp t_i is assigned to each request r_i in the synthetic trace S. We assign timestamp to the synthetic trace based on the byte rate λ obtained from the bFD.
Time complexity. We implement the list C as the leaves of a B^+-tree [15], and thus, the complexity of the algorithm is $O(N \log m)$, where

N is the length of the synthetic trace and m is the number of unique objects in the synthetic trace. The algorithm runs for N iterations and in each iteration it takes $\log(m)$ time to insert the object back into the list at the sampled stack distance (line 19).

We will now formally prove that the algorithm produces a synthetic trace that has approximately the same HRCs as predicted by the footprint descriptors. In particular, if Footprint Descriptor (FD) is provided as the input, the algorithm produces a trace with approximately the same rHRCs (Theorem 3). When Byte-weighted Footprint Descriptor (bFD) is provided as the input, the algorithm produces a trace with approximately the same bHRCs (Theorem 4).

Algorithm 2 Synthetic trace generator

```
1:  Input. (i) A Footprint Descriptor (FD) $\langle \lambda, P^r(s,t), P^a(s,t) \rangle$, (ii) Byte-
    weighted Footprint Descriptor (bFD) $\langle \lambda, P^{rb}(s,t), P^{ab}(s,t) \rangle$, (iii) Ob-
    ject size distribution ($SZ(z)$) and (iv) trace length $N$.
2:  Output. A synthetic trace $S = \{r_1, \ldots, r_N\}$, 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:  $P(s) = \sum_t P^r(s,t)$ or $P(s) = \sum_t P^{rb}(s,t)$ depending on whether RHR
    or BHR is required.
5:  $C \leftarrow \{\}$, $C_{size} = 0$.
6:  $C_{max}$ is the maximum finite $s$ in $P(s)$.
7:  while $C_{size} < C_{max}$ do
8:      Create object $o$ and assign a size $z$ sampled from $SZ$.
9:      Add object $o$ to the list $C$.
10:     $C_{size} \leftarrow C_{size} + z$.
11: end while
12: Phase 2 - Synthetic trace generation.
13: $S \leftarrow \phi$, $i \leftarrow 0$.
14: while $i < N$ do
15:     Append the first object $o = \langle o_{id}, z \rangle$ in $C$ to the trace $S$.
16:     Sample stack distance $s$ from $P(s)$.
17:     if $s$ is not $\infty$ then
18:         Remove $o$ from $C$.
19:         Compute $j = min\{k : \sum_{i=1}^{k} z_k \geq s\}$; where $z_i$ is the size of the
    object at $C[i]$.
20:         Re-insert object $o$ at position $j$ in $C$ by moving objects at posi-
    tions $\geq j$ by a step.
21:     else// $\infty$ means a new object was introduced in the trace.
22:         Remove the object $o$ that is at the first index in $C$.
23:         Create new object $o'$ and assign it a size $z$ sampled from $SZ$.
24:         Add object $o'$ at the end of the list $C$.
25:     end if
26:     $i \leftarrow i + 1$
27: end while
28: Assign timestamps to requests in $S$ using $\lambda$ from bFD.
29: return $S$
```

Theorem 3. *Given a FD, $\langle \lambda, P^r(s,t), P^a(s,t) \rangle$, and a size distribution SZ of an original trace Π, Algorithm 2 produces a synthetic trace $S = \{r_1, \ldots, r_N\}$, where 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 $N \rightarrow \infty$, the rHRC of traces S and Π for an LRU cache are approximately equal.*

Proof. Let $P(s) = \sum_t P^r(s,t)$, where $P^r(s,t)$ is the reuse subsequence descriptor function of trace Π. Consider the synthetic trace $S = \{r_1, \ldots, r_N\}$. In each iteration i of the algorithm, we sample a stack distance s_i from $P(s)$ and request r_i is added to the synthetic trace S. Let request r_i be made for an object o. We know s_i can either be a finite quantity or be ∞. We consider both cases and show that in either case s_i is approximately represented in the synthetic trace.

Case 1: s_i is finite. In the i^{th} iteration, let k be the smallest index in C such that the sum of the sizes of the objects from position 1 to k in C is greater than or equal to s_i. Let z_k be the size of the object at position k. As o is inserted at the position k in C, it is inserted at a stack distance that is at least s_i and at most $s_i + z_k$ (line 19, Algorithm 2). Now, if r_j is the subsequent request in S that was made for object o, the request subsequence $\theta = \{r_i, \ldots, r_j\}$ is a reuse request subsequence in S. The unique objects in θ are the objects present at positions 1 to k in C in the i^{th} iteration of the algorithm. We know the sum of the sizes of these objects is at least s_i and at most $s_i + z_k$. Since the number of objects in the trace is typically large, the reuse request sequences often have many unique objects, hence $s_i \gg z_k$. Therefore, the number of unique bytes in θ is approximately s_i and the sampled stack distance s_i is represented by the request subsequence θ in S.

Case 2: s_i is ∞. If we sampled a stack distance that is ∞, the algorithm discards the object o from the list and introduces a new object at the end of the list. Recall that in the computation of a Footprint Descriptor (FD), the first access to an object is counted as infinite stack distance (see Section 2.2).

Thus, in each iteration, the sampled stack distance is approximately represented in the synthetic trace. As $N \rightarrow \infty$, the distribution of unique bytes across the reuse subsequences in S approximately converges to the $P(s)$ of trace Π. Now, since the rHRC can be computed from $P(s)$ using Theorem 1, both traces Π and S have approximately the same rHRC. ◻

Theorem 4. *Given a bFD, $\langle \lambda, P^{rb}(s,t), P^{ab}(s,t) \rangle$ and a size distribution SZ of an original trace Π, Algorithm 2 produces a synthetic trace $S = \{r_1, \ldots, r_N\}$, where 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 $N \rightarrow \infty$, the bHRC of traces S and Π for an LRU cache are approximately equal.*

Proof. Given a bFD $\langle \lambda, P^{rb}(s,t), P^{ab}(s,t) \rangle$, Algorithm 2 computes $P(s) = \sum_t P^{rb}(s,t)$ and uses $P(s)$ for sampling the stack distance. Using Theorem 3, we know that the rHRC of the synthetic trace S approximately equals $\sum_{s' \leq s} P(s')$, i.e., rHRC of S approximately equals the bHRC of Π. We will now show that rHRC and bHRC of S are equal, and hence, bHRC of S approximately equals bHRC of Π.

Let $rhrc(s)$ (resp., $bhrc(s)$) be the probability that a reuse request subsequence (resp., reuse byte subsequence) in S contains exactly s unique bytes. We will show that $rhrc(s) = bhrc(s)$.

Let k be an object of size z_k in S and θ_k be the set of reuse request subsequences of S that end in object k. The set θ_k consists of reuse request subsequences such that the subsequence either (i) starts at the beginning of the trace and ends in the first access for object k, or (ii) that begin and end in object k. Now, the expected number of reuse request subsequences in θ_k that contain s unique bytes is given by

$rhrc(s).|\theta_k|$ and the number of reuse byte subsequences that contain s unique bytes and end in a byte that belongs to object k is obtained as $rhrc(s).|\theta_k|.z_k$. Therefore, if K is the set of all objects that are requested in S, the expected number of reuse byte subsequences in S that contain s unique bytes is given by,

$$B_s = rhrc(s) \sum_{k \in K} |\theta_k|.z_k. \tag{1}$$

Now, the term $|\theta_k|.z_k$ in the above expression gives us the number of reuse byte subsequences that end in any byte of object k, and thus, the summation across all objects $\sum_{k \in K} |\theta_k|.z_k$, gives us the total number of reuse byte subsequences in S. Let $B = \sum_{k \in K} |\theta_k|.z_k$. Equation 1 can be simplified as,

$$rhrc(s) = \frac{B_s}{\sum_{k \in K} |\theta_k| z_k} = \frac{B_s}{B} = bhrc(s)$$

Since bHRC and rHRC of S can be computed from $bhrc(s)$ and $rhrc(s)$, respectively, the bHRC of S equals rHRC of S. As rHRC of S approximately equals the bHRC of Π, the bHRC of S approximately equals the bHRC of Π.

□

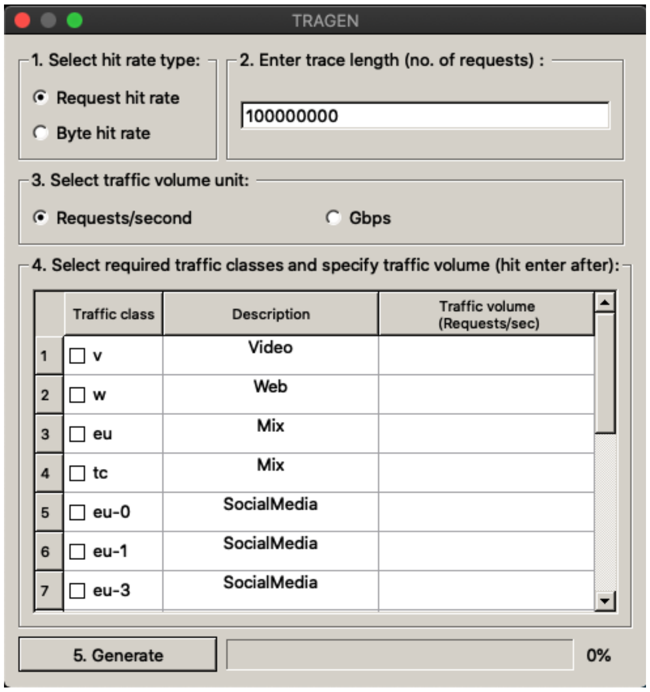
4.2 How to use TRAGEN

The tool is written in python with around 2000 lines of code. The tool can be accessed through a GUI or a command line interface. A screenshot of the GUI is shown in Figure 5. The user is expected to fill in the following details:

(1) **Select hit rate type.** Select if the synthetic trace is to have the same RHR or BHR as the original.
(2) **Enter trace length.** Specify the number of requests in the synthetic trace.
(3) **Select traffic volume unit.** Select if the traffic volume field in the third column of the table will be input as requests/second or Gigabits per second (Gbps).
(4) **Select required traffic classes and specify the traffic volume.** Select traffic classes from the first column of the table and specify a traffic volume for the selected traffic classes in the third column of the table. The synthetic trace will be similar to original production traffic with the specified mix. The second column provides a description of each choice. Each choice is either a pure traffic class such as video, web, or social media traffic class. Or, it is a traffic mix itself, e.g., EU a mix of all traffic served by a cache located in Europe in the production CDN.
(5) **Generate.** Hit the generate button and TRAGEN will start producing the synthetic trace.

Based on the selected hit rate type, the tool uses the FD or the bFD calculus to generate the properties of the traffic mix. The tool then implements the algorithm defined in Section 4.1 to produce a synthetic trace. A command line version of the tool will also be made available.

Performance. The tool implements a B^+-tree [15] to represent the cache (list C in Algorithm 2). Both insert and delete take time $O(\log m)$, where m is the number of objects in the cache. In each iteration of the Algorithm 2, we incur a deletion and insertion, therefore if N is the length of required trace, the tool runs with time complexity $O(N \log m)$. The tool takes around 580 to 640 seconds


TRAGEN

1. Select hit rate type: [✓ Request hit rate] Byte hit rate

2. Enter trace length (no. of requests) : 100000000

3. Select traffic volume unit: [✓ Requests/second] Gbps

4. Select required traffic classes and specify traffic volume (hit enter after):

	Traffic class	Description	Traffic volume (Requests/sec)
1	☐ v	Video	
2	☐ w	Web	
3	☐ eu	Mix	
4	☐ tc	Mix	
5	☐ eu-0	SocialMedia	
6	☐ eu-1	SocialMedia	
7	☐ eu-3	SocialMedia	

5. Generate 0%

Figure 5: TRAGEN GUI

to generate a trace of 10 million requests for the four traffic classes in Table 1.

Downloading TRAGEN. We have made the tool open-source and is publicly available for download[2]. Apart from the code that is used to generate traces, we have also released the code that computes the FDs, bFDs and SZ from original traces. *So, in addition to our traffic models, users can compute and seed TRAGEN with traffic classes from their own environment.*

5 Empirical evaluation

We empirically validate TRAGEN by showing that it produces a synthetic trace that is similar to the original production trace, for a range of commonly-used caching policies and for a range of traffic classes from Akamai's production CDN. We collect four production traces from the Akamai CDN, each consisting of hundreds of millions of requests made for a few million objects. The traces are described in Table 1. The VIDEO and WEB traces are collected from CDN servers that are predominantly serving video and web traffic, respectively. The EU and TC traces are collected from CDN servers that serve a mix of traffic. The EU trace consists of 10 different traffic classes with varying characteristics (Table 3), while TC trace consists of requests for Download, Images, Media and Web (Table 2). In each experiment, we generate a synthetic trace of 100 million requests, unless otherwise specified.

5.1 TRAGEN validation

We first show results individually for traces in Table 1 and then show results for the traffic mixes.

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

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

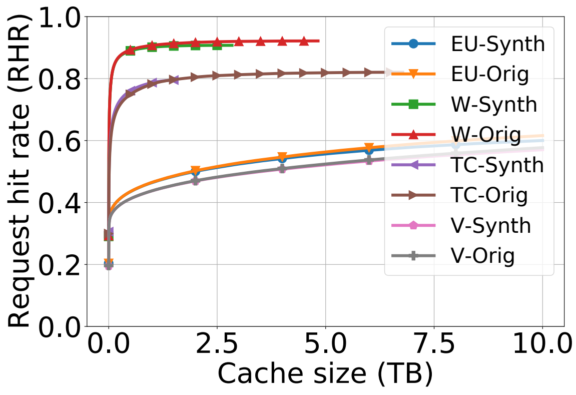


Figure 6: rHRC of original and synthetic traces in Table 1

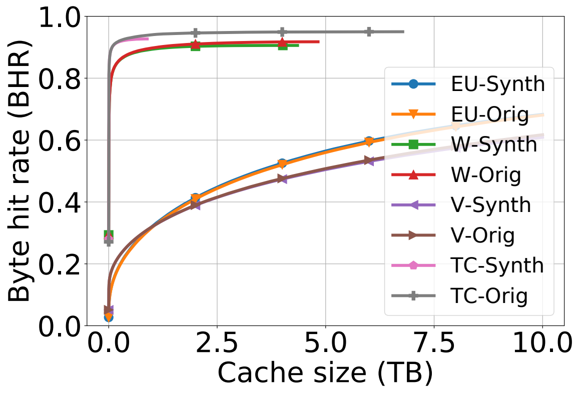


Figure 7: bHRC of original and synthetic traces in Table 1

Validation by comparing hit rate curves. We show that the HRCs of original and synthetic trace are similar for a LRU cache. For rHRCs, we use FDs of the original traces in Table 1 to produce synthetic traces. To show that the bHRCs are equal, we use bFDs to produce synthetic traces. The rHRCs and bHRCs of the original and the synthetic traces are shown in Figure 6 and Figure 7, respectively. The curves are computed at intervals of 200 MB cache size. We compute *average difference* defined as $\sum_{s\in C} |HRC_{original}(s) - HRC_{synthetic}(s)|/|C|$, where C is the set of all cache sizes the hit rates are computed for. We observe an average difference of 2.4e-07, 8.2e-07, 1.6e-06, and 4.6e-07 in the rHRCs and an average difference of 4.6e-07, 1.4e-06, 1.1e-06, and 5.6e-07 in the bHRCs of traces VIDEO, WEB, TC and EU, respectively. Thus, the average difference is extremely small and the HRCs of the original and synthetic traces are nearly identical.

Validation by simulating different caching policies. We empirically validate that TRAGEN produces a synthetic trace that will yield similar RHRs and BHRs as the original trace by implementing and running cache simulations for commonly-used caching policies that are listed below.

(1) **FIFO.** The First In First Out caching policy evicts objects from the cache in the order they were inserted. FIFO is easily implementable, provides comparable hit rates as LRU in practice [20], and provides better performance on SSDs as compared to other caching policies [36]. Hence, we expect FIFO to be widely used.
(2) **RANDOM.** RANDOM caching policy evicts a random object from cache upon the insertion of a new object. Due to its simplicity, RANDOM caching policy and its variants are widely studied [1, 33] and used in practice (for instance, in ARM Processors [58]).
(3) **SLRU (and S4LRU).** Segmented LRU divides the cache into two segments that individually run the LRU caching policy. Upon a cache miss, the requested object is first inserted into the lower segment. On a subsequent request, if the requested object is in the lower segment, it is moved to the upper segment. SLRU is used as the caching policy for Facebook photo caching [27]. S4LRU is similar to SLRU, but divides the cache into four segments.
(4) **MARKERS.** The markers caching policy runs in phases. At the beginning of each phase, all objects in the cache are unmarked. On a cache hit, the requested object is marked. On a cache miss, the requested object is inserted into the cache and marked. Upon insertion, one of the unmarked objects is evicted. A new phase begins when all objects in the cache are marked [1, 37].
(5) **CLOCK.** The clock caching policy maintains a circular list of the objects that are present in cache and an iterator that points to the last examined object in the list. To evict an object, a R (referenced) bit is inspected at the iterators location. If R is 0, the object at the iterators location is evicted. If not, R is unset and the iterator is incremented to point to the next object. The process repeats till an object is evicted from the cache [16].
(6) **PLRU.** The Pseudo-LRU caching policy is a tree based policy that approximates LRU. PLRU arranges objects in the cache as leaves of a binary tree and maintains pointers in the non leaf nodes to point to an object that has not been recently used. This object is evicted upon an insertion of a new object into the cache. These pointers are updated on every cache access and eviction. PLRU is used in the TC1798 CPU and several POWERPC variants (MPC603E, MPC755, MPC7448) [46].

We verify that an LRU cache yields the same RHR and BHR for the original and synthetic traces, across all the traces and cache sizes that we tested on. The results for the VIDEO and EU trace are shown in Figure 10. This validates Theorem 3 and Theorem 4. The RHRs of the synthetic trace and the original trace are similar for the other caching policies with a maximum difference of around 5% that is observed for the EU trace on a cache of size 2 TB that uses CLOCK caching policy. For the other caching policies, the difference in RHRs for the original and synthetic trace is below 2% across all cache sizes. We observe similar results in the BHRs for the tested caching policies. We observe a maximum difference of around 3.5% in the BHRs for the VIDEO trace on a cache size of 2TB that uses the MARKERS caching policy. The difference is smaller than 2% for other caching policies for the VIDEO and EU trace.

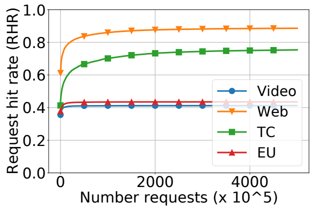
Thus, TRAGEN produces a synthetic trace that yields a RHR (resp., BHR) that differs from the RHR (resp., BHR) of the original trace by 1.5% (resp., 1%) on average and at most 5% (resp., 3%) in the worst-case, across all cache sizes, caching policies and traces in our evaluation.

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

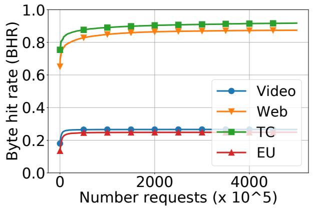
Table 2: Trace description for the TC trace

Eviction age. We also compare the expected eviction age of the original trace and of the synthetic trace for a LRU cache. Eviction age is an important caching metric since it measures how long an object stays in cache after its last access – a smaller eviction age means that the content in the cache is churning too quickly, leading to worse cache performance. Upon assigning a timestamp to the synthetic trace based on the byte rate of the original, the average eviction age of the original and synthetic trace align for various cache sizes (Figure 9). Particularly, eviction age of the synthetic trace differs from the original trace by 1.8% on an average across all cache sizes for the VIDEO trace.

Evaluating traffic mixes. We now show that our tool can generate a synthetic trace for any specified traffic mix. We consider two case studies. First, we consider the EU trace and a traffic mix specified by the Media-0, Media-1 and Media-2 traffic classes with traffic volumes 40, 140 and 90 requests/second, respectively. We then generate a trace as described in Section 4. The result is seen in Figure 11a. The curves RHR-Gen (resp., BHR-Gen) and RHR-Orig (resp., BHR-Orig) depict the rHRC (resp., bHRC) of the synthetic trace and the original trace as predicted by FD (resp., bFD) calculus. Next, we consider a traffic mix of the Image, Media and Web traffic classes from the TC trace with traffic volumes 80, 8 and 250 MBps, respectively. We observe similar results (Figure 11b).



(a)



(b)

Figure 8: Converging to the required hit rates for an LRU cache of size 500GB. (a) Cumulative RHR; (b) Cumulative BHR.

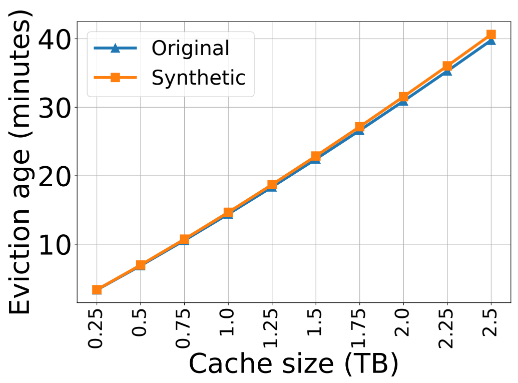


Figure 9: Mean eviction age across cache sizes for VIDEO trace.

5.2 Determining the synthetic trace length

For using TRAGEN for cache simulations, it is necessary to determine how long a cache simulation should run, starting from an empty cache. The answer depends on how quickly the hit rates converge to a stationary value, such convergence depends on the caching policy, cache size, and the nature of the trace. As an example, we explore the convergence of the hit rate of the synthetic trace produced by TRAGEN to the stationary RHR and BHR for an LRU cache of size 500 GB. For the WEB and TC trace, the convergence to a stationary RHR is slower and takes up to 20 million requests. However, for the traces VIDEO and EU the convergence occurs within a million requests. A possible explanation is that for the EU and VIDEO trace, a cache size of 500 GB is small as compared to the overall footprint of the trace. Whereas, for the WEB and TC trace a cache size of 500 GB is large enough. Results are shown in Figure 8a. Similarly, we explore the convergence of BHR for the various traces in Figure 8b.

5.3 Comparison with alternate approaches

We will now discuss two alternate approaches that can be used to generate synthetic traces, LRUSM and Naive Merge, and show why both approaches fail to produce synthetic traces that have similar hit rates as the original. Thus, neither approach allows the synthetic trace to take the place of an original trace in realistic cache simulations.

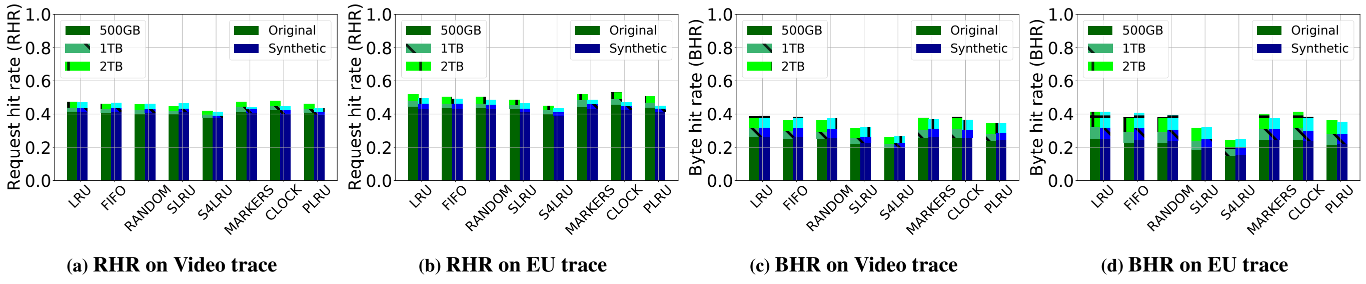
5.3.1 LRUSM Algorithm. The LRUSM algorithm has been used extensively in the synthetic trace generation community for CPU caches and Web Caches. LRUSM generates synthetic traces that capture the temporal locality of the original trace [5, 12, 26]. All prior work in the synthetic trace generation literature that capture temporal correlations use the LRUSM algorithm [5, 12].

We experimented with the LRUSM algorithm to produce a synthetic trace. We find that the synthetic trace in almost all cases does not have the same HRCs as the original trace. For instance, Figure 12a depicts the HRCs for the VIDEO trace (Table 1). Thus, we conclude that LRUSM algorithm fails to generate a synthetic trace with similar hit rate curves as the original trace.

Adapting LRUSM to variable-sized objects. The LRUSM algorithm considers unit size objects to produce a synthetic trace, and then assigns each object a size from the object size distribution. If a FD is provided as an input, the LRUSM algorithm can be updated as follows. The LRU stack is initialized with objects and are assigned sizes from the object size distribution. In each iteration, on sampling a stack distance s from $P(s)$, where $P(s) = \sum_t P^r(s,t)$, the object that falls at a stack distance s is added to the synthetic trace and re-inserted at the top of the stack. Now, observe that the probability of s falling on a large object is higher as compared to a small object.

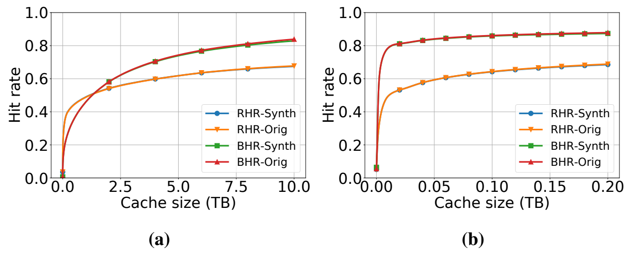
Trace	Media-0	Media-1	Media-2	Media-3	Media-4	Media-5	Media-6	Web-7	Media-8	Web-9
Length (mil. reqs)	32.04	109.3	70.3	91.92	43.98	66.48	36.56	9.73	128.44	6.95
Req. rate (reqs/sec)	20.64	70.44	45.32	59.2	28.33	42.82	23.55	6.248	82.73	5.38
Traffic (MBps)	12	480	13	36	288.3	434.8	26.8	0.8	27.682	0.756
No. 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 3: Trace description for the EU trace



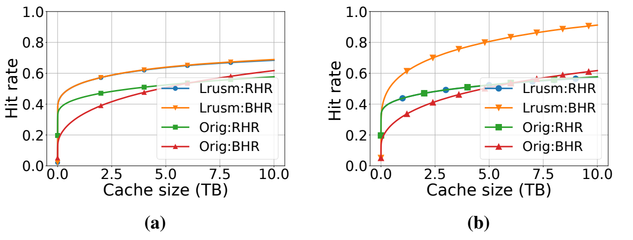
(a) RHR on Video trace **(b) RHR on EU trace** **(c) BHR on Video trace** **(d) BHR on EU trace**

Figure 10: Observed RHRs and BHRs of the Original and Synthetic trace for the various caching policies and cache sizes for the Video and EU trace.



(a) **(b)**

Figure 11: Traffic mix results for (a) EU Trace, and (b) TC trace



(a) **(b)**

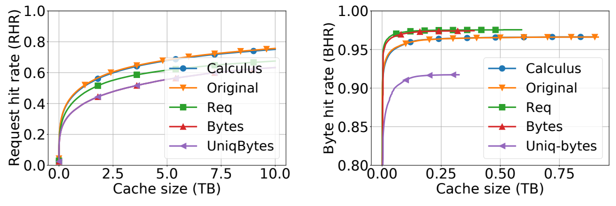
Figure 12: LRUSM approach. (a) HRCs under the LRUSM approach with stack initialized with unit size objects, (b) HRCs under LRUSM approach with stack initialized with object sizes.

Thus, the synthetic trace is likely to have a much higher proportion of large objects. This in turn, causes the BHR of the synthetic trace to be much higher than the RHR of the original trace. The results for the VIDEO trace can be seen in Figure 12b.

5.3.2 Naive Merge.

TRAGEN uses FD (resp., BFD) calculus to compute a model for traffic class mixes and uses that model for synthetic trace generation. We now ask if a naive approach (which we call Naive Merge) could be used to derive synthetic traces for traffic class mixes instead. Naive Merge uses TRAGEN to generate synthetic traces for individual traffic classes and then merges them in time-order by assigning each request a time stamp.

We consider three ways of assigning timestamps to the synthetic trace, (i) request rate (number of requests per second), (ii) byte rate (number of bytes requested per second), and (iii) unique byte rate (number of unique bytes requested per second) of the traffic class.



(a) **(b)**

Figure 13: Approach naive merge. (a) rHRCs of the synthetic traces for the EU trace (b) bHRCs of the synthetic traces for the TC trace.

Figure 13a and Figure 13b depict our findings. For the result in Figure 13a, we consider traffic classes eu-0 and eu-1 from the EU trace (Table 3). We observe that the rHRC of the trace computed using Footprint Descriptor (FD) of the traffic mix and the one computed using the calculus align. Whereas, the rHRC of the trace computed by using any of the merge techniques are not equal to the rHRC of the original trace. To compare bHRCs, we repeat the experiment using the traffic classes Download and Images from the TC trace (Table 2). Again, we observe similar results, results seen in Figure 13b. Thus, simpler timestamp-based approaches to traffic mixing does not produce realistic synthetic traces, providing support for our use of FD and bFD calculus.

6 Related work

Synthetic workload generation has been an active research area in several disciplines of computer science. We briefly review the relevant literature on workload characterization and generation in internet based systems.

Tools for synthetic trace generation. The tool SpecWeb96 [17] is amongst the first tools to generate a synthetic web workload. The tool generates a sequence of HTTP GET requests such that the requests satisfy the size and popularity distribution of an expected web trace. The tool is however dormant. HttpPerf [43] is a similar tool that is used to generate a trace to test the performance of a web server. SURGE [5] is a more realistic web workload generator and generates a trace that matches the empirical measurements of popularity distribution, object size distribution, request size distribution, relative object popularity, temporal locality of reference, idle periods of individual users. Some of the other tools with similar capabilities are Web Polygraph [47], Globetraff [32], Geist [30]. Unlike our work, these tools do not generate a synthetic trace that is similar to the original trace in terms of hit rates.

The work in [45] captures the correlations between requests and the popularity distribution of the original workload and generates a trace that satisfies both. The work does not consider the size distribution of the objects. The tool MediSyn [52], generates a trace with properties specific to streaming media characteristics such as file duration, encoding bit rate, session duration and non-stationary popularity of media accesses, but does not consider the cache hitrates.

The tool closest to our work is ProwGen [12]. ProwGen was used to study the workload characteristics that impact the cache hit rate of a proxy server by generating a synthetic trace that closely satisfies the workload characteristics. The tool however is limited to generating web traffic traces and is unable to generate a trace with the same cache hit rates as the original trace as it relies on the LRUSM algorithm. We have shown in Section 5.3 that the LRUSM algorithm fails to produce a synthetic trace that is similar to original production traces. Further, the tool considers very small cache sizes (order of MBs) and runs in $O(n^2)$, which makes it impractical considering the current workload and much larger cache sizes. A comprehensive survey of the available workload generation tools and their capabilities is provided in [18].

Unlike prior work, TRAGEN is the first tool to produce synthetic traces with similar hit rates as the original traces for a cross-section of modern traffic classes. This makes TRAGEN suitable for realistic cache simulations. Further, unlike prior work that produce traces that cater only to small cache sizes and trace lengths, TRAGEN incorporates better data structures to produce sufficiently long traces that satisfy the caching properties of large caches.

Characterizing the workload of Internet services. There exist several work that characterize the workload of the Internet and the services based on it [4, 14, 39, 48, 55, 56]. In 1997, Arlitt et al. provided an extensive study of the web workload characteristics using data sets obtained from 6 websites and identified 10 invariant characteristics of the workload. The authors revisited the study in 2007 [55] to find that the invariants still hold. A similar study was performed by Mahanti et al. in [39]. In both the studies, the workload was predominantly Web traffic. However, in recent times, with applications such as streaming, online gaming, social media and software downloads that use the Internet, the Internet traffic has become highly diversified [14, 48, 49, 56]. CDNs like Akamai, serve multiple traffic classes from their servers [50] and each traffic class is shown to have unique access patterns and content properties. The caching properties of each traffic class was captured with a succinct representation of Footprint Descriptor and the caching properties of a traffic mix is obtained using the Footprint Descriptor calculus [50]. *Considering the vast diversity in the Internet traffic and given that a production cache serves various time-varying mixes of the traffic classes, the flexibility that TRAGEN provides in generating synthetic traces for any prescribed traffic mix is essential.*

Stack distance distribution. Stack distance or reuse distance as a metric has been a useful tool to capture the temporal locality and the cache properties for CPU caches and web caches [8]. The stack distance distribution of a trace can be used to compute the rHRC of an LRU cache for the trace. Over time, several methods have been proposed to speed up the computation of the stack distance distribution [3, 54]. However, the metric can only provide the rHRC of the trace. *We introduce the byte-weighted footprint descriptor that extends the footprint descriptor to capture the bHRC of the trace.*

The Independent Reference Model (IRM). The IRM model has been widely used to describe the request process in several applications. The IRM model assumes that each request references an object and the reference is independent of prior requests. Further, each object is assigned a popularity that fits a zipfian distribution [10, 24, 40]. Under this model, there exists considerable work that quantifies the expected hit rates the requests would incur [13, 21]. However, it is well known that real traffic does not follow IRM that completely ignores temporal locality [23, 35, 53] and thus researchers have proposed statistical models such as shot noise model [35], markovian arrival process [11]. But they are not known to be accurate and it is also not known if they capture the expected HRCs accurately. Hence, we rely on a more robust model of footprint descriptors that capture caching properties without making any statistical assumptions and is now used in industry.

7 Conclusion

We design and implement TRAGEN, the first tool to produce synthetic traces that are similar to original production traces in terms of hit rates and eviction ages. TRAGEN supports user-specified traffic mixes that allow developers and researchers to generate a wide range of realistic workloads for cache simulations.

TRAGEN is available to the public and comes with footprint descriptor models of major traffic classes from a large production CDN. This allows users to generate realistic synthetic traces that accurately represent the immense variety of content access patterns on the internet. Further, we provide the tools for users to generate footprint descriptor traffic models from their own original traces. This allows researchers and developers to use TRAGEN for simulations in their own caching application domains. Thus, TRAGEN provides a platform for industry and academia to publish traffic models (FDs and bFDs) from their own caching systems, allowing other researchers and system designers to compute similar synthetic traces, while preserving the privacy of the original production traces.

TRAGEN is guaranteed to produce synthetic traces that have similar caching behavior to the original traces for the set of caching policies that we could theoretically or empirically validate. Providing strict guarantees for a broader set of caching policies is future work.

8 Acknowledgements

We would like to thank our anonymous reviewers for their valuable feedback. We would like to thank our shepherd Paul Barford for his insightful comments that improved the quality of the paper immensely. We also thank Aditya Sundarrajan for his help in the initial phases of the project. This work was supported in part by NSF grants CNS-1763617 and CNS-1901137.

References

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

[2] 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.

[3] 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.

[4] 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.

[5] 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.

[6] D. S. Berger, B. Berg, T. Zhu, S. Sen, and M. Harchol-Balter. Robinhood: Tail latency aware caching–dynamic reallocation from cache-rich to cache-poor. In *13th {USENIX} Symposium on Operating Systems Design and Implementation ({OSDI} 18)*, pages 195–212, 2018.

[7] 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.

[8] K. Beyls and E. D'Hollander. Reuse distance as a metric for cache behavior. In *Proceedings of the IASTED Conference on Parallel and Distributed Computing and systems*, volume 14, pages 350–360. Citeseer, 2001.

[9] 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.

[10] L. Breslau, P. Cao, L. Fan, G. Phillips, and S. Shenker. Web caching and zipf-like distributions: Evidence and implications. In *IEEE INFOCOM'99. Conference on Computer Communications. Proceedings. Eighteenth Annual Joint Conference of the IEEE Computer and Communications Societies. The Future is Now (Cat. No. 99CH36320)*, volume 1, pages 126–134. IEEE, 1999.

[11] P. Buchholz, P. Kemper, and J. Kriege. Multi-class markovian arrival processes and their parameter fitting. *Performance Evaluation*, 67(11):1092–1106, 2010.

[12] 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.

[13] H. Che, Y. Tung, and Z. Wang. Hierarchical web caching systems: Modeling, design and experimental results. *IEEE journal on Selected Areas in Communications*, 20(7):1305–1314, 2002.

[14] 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.

[15] D. Comer. Ubiquitous b-tree. *ACM Computing Surveys (CSUR)*, 11(2):121–137, 1979.

[16] F. J. Corbato. A paging experiment with the multics system. Technical report, MASSACHUSETTS INST OF TECH CAMBRIDGE PROJECT MAC, 1968.

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

[18] M. Curiel and A. Pont. Workload generators for web-based systems: Characteristics, current status, and challenges. *IEEE Communications Surveys & Tutorials*, 20(2):1526–1546, 2018.

[19] 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.

[20] 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.

[21] R. Fagin and T. G. Price. Efficient calculation of expected miss ratios in the independent reference model. *SIAM Journal on Computing*, 7(3):288–297, 1978.

[22] B. Fan, H. Lim, D. G. Andersen, and M. Kaminsky. Small cache, big effect: Provable load balancing for randomly partitioned cluster services. In *Proceedings of the 2nd ACM Symposium on Cloud Computing*, pages 1–12, 2011.

[23] M. Garetto, E. Leonardi, and S. Traverso. Efficient analysis of caching strategies under dynamic content popularity. In *2015 IEEE conference on computer communications (INFOCOM)*, pages 2263–2271. IEEE, 2015.

[24] 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.

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

[26] R. Hassan, A. Harris, N. Topham, and A. Efthymiou. Synthetic trace-driven simulation of cache memory. In *21st International Conference on Advanced Information Networking and Applications Workshops (AINAW'07)*, volume 1, pages 764–771. IEEE, 2007.

[27] 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.

[28] 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.

[29] 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.

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

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

[32] 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.

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

[34] V. Kirilin, A. Sundarrajan, S. Gorinsky, and R. K. Sitaraman. Rl-cache: Learning-based cache admission for content delivery. *IEEE Journal on Selected Areas in Communications*, 38(10):2372–2385, 2020.

[35] E. Leonardi and G. L. Torrisi. Least recently used caches under the shot noise model. In *2015 IEEE Conference on Computer Communications (INFOCOM)*, pages 2281–2289. IEEE, 2015.

[36] 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.

[37] T. Lykouris and S. Vassilvtiskii. Competitive caching with machine learned advice. In *International Conference on Machine Learning*, pages 3296–3305. PMLR, 2018.

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

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

[40] A. Mahanti, C. Williamson, and D. Eager. Traffic analysis of a web proxy caching hierarchy. *IEEE Network*, 14(3):16–23, 2000.

[41] 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.

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

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

[44] A. Narayanan, S. Verma, E. Ramadan, P. Babaie, and Z.-L. Zhang. Deepcache: A deep learning based framework for content caching. In *Proceedings of the 2018 Workshop on Network Meets AI & ML*, pages 48–53, 2018.

[45] K. Psounis, A. Zhu, B. Prabhakar, and R. Motwani. Modeling correlations in web traces and implications for designing replacement policies. *Computer Networks*, 45(4):379–398, 2004.

[46] 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.

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

[48] 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.

[49] 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.

[50] 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.

[51] 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.

[52] 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.

[53] S. Traverso, M. Ahmed, M. Garetto, P. Giaccone, E. Leonardi, and S. Niccolini. Unravelling the impact of temporal and geographical locality in content caching systems. *IEEE Transactions on Multimedia*, 17(10):1839–1854, 2015.

[54] C. A. Waldspurger, N. Park, A. Garthwaite, and I. Ahmad. Efficient {MRC} construction with {SHARDS}. In *13th {USENIX} Conference on File and Storage Technologies ({FAST} 15)*, pages 95–110, 2015.

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

[56] J. Yang, Y. Yue, and K. 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, 2020.

[57] 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.

[58] J. Yiu. *The Definitive Guide to ARM® Cortex®-M0 and Cortex-M0+ Processors*. Academic Press, 2015.