Footprint Descriptors: Theory and Practice of Cache Provisioning in a Global CDN

Aditya Sundarrajan
University of Massachusetts Amherst

Mingdong Feng
Akamai Technologies

Mangesh Kasbekar
Akamai Technologies

Ramesh K. Sitaraman
University of Massachusetts Amherst & Akamai Technologies

ABSTRACT

Modern CDNs cache and deliver a highly-diverse set of traffic classes, including web pages, images, videos and software downloads. It is economically advantageous for a CDN to cache and deliver all traffic classes using a shared distributed cache server infrastructure. However, such sharing of cache resources across multiple traffic classes poses significant cache provisioning challenges that are the focus of this paper.

Managing a vast shared caching infrastructure requires careful modeling of user request sequences for each traffic class. Using extensive traces from Akamai's CDN, we show how each traffic class has drastically different object access patterns, object size distributions, and cache resource requirements. We introduce the notion of a footprint descriptor that is a succinct representation of the cache requirements of a request sequence. Leveraging novel connections to Fourier analysis, we develop a footprint descriptor calculus that allows us to predict the cache requirements when different traffic classes are added, subtracted and scaled to within a prediction error of 2.5%. We integrated our footprint calculus in the cache provisioning operations of the production CDN and show how it is used to solve key challenges in cache sizing, traffic mixing, and cache partitioning.

ACM Reference format:
Aditya Sundarrajan, Mingdong Feng, Mangesh Kasbekar, and Ramesh K. Sitaraman. 2017. Footprint Descriptors: Theory and Practice of Cache Provisioning in a Global CDN. In *Proceedings of CoNEXT '17, Incheon, Republic of Korea, December 12–15, 2017,* 13 pages.
DOI: 10.1145/3143361.3143368

1 INTRODUCTION

Much of the world's online content are delivered by Content Delivery Networks (CDNs). Modern CDNs cache and deliver a highly-diverse set of traffic classes with different content types such as web pages, images, videos and software downloads. CDNs use a large network of cache servers to deliver content from locations that are proximal to the user. Users download content from these cache servers. If the requested content is found in the cache (i.e., cache hit), the user experiences a faster response time. A large CDN such as Akamai has a platform of hundreds of thousands of cache servers deployed in thousands of locations around the world.

A large CDN may host content from tens of thousands domains belonging to web sites of thousands of content providers. Further, each content provider may host different types of content, including web, downloads, videos, and images[1]. Requests from users accessing the content provider's web sites are routed by the CDN to an appropriate cache server that can serve the content using a process called *mapping* [6].

The content traffic served by a CDN can be classified into *traffic classes*. Each traffic class is a set of domains that host a specific content type belonging to one or more content providers with similar requirements. For instance, a traffic class could be web content from a specific news site, or image content from a specific e-tailer, or iOS downloads from Apple, or video content from Hulu, etc. Thus, a major CDN may have several hundred traffic classes, each with different access characteristics and performance requirements.

A CDN performs cache provisioning and request routing at the granularity of a traffic class. Thus, a key decision a CDN must make is which subset of its hundreds of thousands of servers must serve which traffic classes. Different traffic classes may have different caching characteristics and different performance requirements. A traffic class consisting of web content from an e-tailer may require fast response times and high cache hit rates to aid more sales conversions, and the object sizes are smaller. In contrast, a traffic class consisting of background software downloads has large object sizes, but can tolerate lower hit rates and slower response times.

1.1 Cache Provisioning

Cache provisioning is the process of determining which traffic classes are hosted in which cache servers of the CDN, given a vast platform of cache servers with varying amounts of cache space available at each server. Despite the potentially diverse requirements for each traffic class, it is economically and operationally advantageous for the CDN to use a single shared platform of servers to serve all the traffic classes and to have each cache serve multiple traffic classes. However, such sharing across multiple traffic classes poses significant challenges that are the focus of this paper.

[1]Content providers often segregate their content by type and place them on different domains for better content management and delivery. For instance, a content provider may have a different domain for each content type, such as `{www,video,image,download}.foo.com`.

CoNEXT '17, Incheon, Republic of Korea

978-1-4503-5422-6/17/12…$15.00
DOI: 10.1145/3143361.3143368

Provisioning traffic classes and controlling the sharing of the available cache space between those classes to maximize cache hit rates is an important challenge with direct impact on the cost-performance tradeoff of the CDN. For example, servers hosting an aggressive mix of traffic classes relative to available cache space may end up providing poor cache hit rates. This may violate the performance requirements for some classes, and raise bandwidth cost due to elevated cache-miss traffic. Conversely, servers hosting a conservative mix of traffic classes may end up underutilizing their resources, which makes the CDN buy more servers than necessary.

The goal of cache provisioning is to *model* the caching requirements of traffic classes and to *predict* the best way to assign traffic classes to cache servers, so as to optimize the use of the cache resources and provide an acceptable hit rate at a reasonable cost. To do so, the process takes as input the sizes of the caches available in servers across the CDN, and the characteristics of the request sequences for each traffic class. The process outputs the set of servers that serve each traffic class. Cache provisioning is an offline planning step but it must be performed regularly, since new traffic classes are added or removed to the system and caching characteristics of existing classes may change. Once cache provisioning is complete, its output is used by a mapping system (cf. Section 2.1) to route the requests of each traffic class to one of the provisioned cache servers in real-time.

1.2 Challenges and Contributions

The main conceptual challenges in cache provisioning and our contributions in addressing those challenges are below.

1) To provide effective cache provisioning, we need to first understand the diversity of traffic classes hosted on a modern CDN, and how they vary in terms of user request patterns, content popularity, object sizes, and caching requirements. *In Section 3, we provide the first detailed characterization of traffic classes on a modern CDN.*

2) The user requests for each traffic class must be modeled efficiently from the traces. While traces may contain hundreds of millions of requests, the model must be *concise*, and must be able to *predict* the resource-performance tradeoffs for caches that serve that traffic class. *In Section 4.1, we propose the novel notion of a* footprint descriptor (FD) *that is computed efficiently from user request traces of that traffic class. Using footprint descriptors, we can derive the full tradeoff between cache size and hit rate for each traffic class.*

3) A main goal of cache provisioning is to answer important "what-if" questions through modeling and prediction. Examples of such questions include: what would the hit rate be when multiple traffic classes are mixed together and served by a single shared cache? How should you partition a cache across multiple traffic classes, so that each class receives its target hit rate? How would the hit rates change if the traffic volume of a traffic class is increased? *In Section 4.3, we develop a calculus for footprint descriptors that lets us perform addition, subtraction, and scaling operations on request sequences. The calculus lets us model, predict, and answer the key "what-if" questions that arise in the cache provisioning context. For instance, the calculus allows us to efficiently compute the footprint descriptor for a mix of traffic classes, given the footprint descriptors of each individual class in the mix.*

4) Cache provisioning must be able to process and manipulate traffic class models in an *efficient fashion. In Section 4.3, we show an intriguing connection to Fourier Analysis that lets us visualize and manipulate footprint descriptors. Specifically, we show how Fast Fourier Transform (FFT) can be used to transform footprint descriptors to the "frequency" domain. Analogous to how signal processing can be speeded up by using Fourier Transforms, we show how footprint descriptors can be efficiently manipulated in the frequency domain.*

5) The models used in cache provisioning should provide predictions that are accurate enough to use in production CDN operations. *In Section 4.5, we highlight the need for footprint descriptor calculus through simulations using traces from production servers. We also compare our predictions with hit rates from the production network and show that the prediction error is at most 2.5% in the scenarios considered. In Section 5, we show how footprint descriptors are used to solve key challenges in CDN operations.*

Footprint descriptor modeling versus cache simulations. In theory, one could evaluate the hit rates of different traffic class mixes by experimentally simulating cache operations on each request trace mix. But, simulating various combinations of several hundred traffic classes for different cache sizes is unscalable and prohibitively expensive, even for an offline computation, since it must be repeated periodically (say, every few days). With our approach, the footprint descriptor is computed for each traffic class only once (Section 5.3 shows how to compute FDs efficiently using a map-reduce paradigm) and traffic mixes are evaluated rapidly using footprint descriptor calculus. *The power of footprint descriptors is that it needs to be computed only once for each traffic class from the voluminous traces. Various operations on traffic classes can then be performed rapidly using the calculus without costly cache simulations of traffic class mixes.*

Roadmap. The rest of the paper is organized as follows. In Section 2, we provide background on cache provisioning in CDNs and how footprint descriptors fit into the complex environment of CDN operations. In Section 3, we describe the characteristics of the different traffic classes hosted on the CDN. In Section 4, we introduce the notion and develop the theory of footprint descriptors. In Section 5, we show how footprint descriptors can be used in a production setting for CDN cache operations. In Section 6 we review prior work and conclude in Section 7.

2 BACKGROUND

A large CDN such as Akamai has a network of a few hundred thousand cache servers deployed in *clusters* that are located in thousands of data centers and in most major countries around the globe. In this section, we describe how the mapping, caching, and cache provisioning systems interact with each other to serve content to users. When a user accesses content using a URL (say, `https://domain/path`), the *mapping system* [6, 19] is responsible for routing the user's request to a cache server that can serve the request. Note that the domain of the request belongs to a traffic class, e.g., the domain `downloads1.foo.com` might belong to a traffic class of software downloads from `foo.com`. The cache server that is picked by mapping to serve the request must host the traffic class to which the request belongs. The (offline) cache provisioning process determines which set of cache servers host each traffic class.

Mapping uses that information to route requests in real-time to one of the servers in that set. Once the request is received by a cache server, it is served by the caching system running on that server. We provide some additional relevant details below.

2.1 The Cache Provisioning Process

The cache provisioning process works in an offline fashion undertaking the complex task of periodically deciding which traffic classes are hosted on which subset of the CDN's servers. Let $\mathcal{T}$ be the set of all traffic classes and $\mathcal{N}$ be the set of all cache servers of the CDN. Formally, the cache provisioning process computes a function $\pi : \mathcal{T} \rightarrow 2^{\mathcal{N}}$, where each traffic class $\tau \in \mathcal{T}$ is hosted on a chosen subset of the servers $\pi(\tau) \in 2^{\mathcal{N}}$. The cache provisioning process should accurately predict the effects of traffic mixing under varying traffic conditions and availability of caching resources. The goal of cache provisioning is maximizing metrics that are important to the CDN, such as the aggregate hit rate of each cache server and the specific hit rates of each traffic class that it hosts.

Currently, cache provisioning in production CDNs happens in an adhoc fashion where a human operator frequently modifies the traffic class assignment π based on past experience, as the characteristics of traffic classes change, or when new traffic classes are added or removed, or when server clusters in data centers are deployed or deconstructed. Our goal is to develop models and prediction tools that a human operator could use to quickly answer "what-if" questions on how traffic class mixes behave when they share the same server and what hit rates each class gets individually and in aggregate. In this paper, we propose footprint descriptors that succinctly model request sequences of different traffic classes and a footprint descriptor calculus that accurately predicts the outcome of traffic mixing. We show how footprint descriptors can be used to address the complex challenges listed in Section 1.2. We also discuss many such scenarios through case studies in Section 5.

Our current work is focused on providing tools for a human operator who determines the function π, rather than eliminating the human operator from the cache provisioning process altogether. The holy grail of cache provisioning is to compute π automatically without human intervention at all. Extending footprint descriptors to such a scenario is left as future work.

Once cache provisioning is complete, the *mapping system* routes user requests to their appropriate servers in *real time*. The mapping system assigns each request for a domain in traffic class τ to a live server in $\pi(\tau)$ that is proximal to the user. The reader is referred to [6, 19] for a more detailed description of mapping system architecture.

2.2 The Caching System

After the user receives a server IP from the mapping system, the user requests the content from that assigned server. Each server has a cache for storing content. If the server has the requested content in its cache (i.e, a "cache hit"), it is delivered to the user. If the content is not available in cache (i.e., a "cache miss"), the server fetches it from the content provider's origin that has the original copy of the content.

The caching system consists of hundreds of thousands of servers deployed around the world, each server implementing a content cache. The servers are heterogenous and implement caches of varying sizes. Each cache server implements a cache replacement algorithm. Most production CDNs and caching software use extensions or variants of LRU, including Akamai [15], Nginx [17], and Varnish [13]. The LRU cache replacement algorithm works as follows. When an object is accessed, it is placed in cache. If the cache is full, the least-recently-used object is evicted from cache. Our calculus models an LRU cache, though it can also be extended to a broader class of stack algorithms that satisfy the inclusion property [16].

The primary metric of cache efficiency is its hit rate. *Hit rate*[2] is the percentage of the requested bytes that were found in cache. That is, hit rate is percentage of requests for objects that were cache hits, weighted by object size. A cache hit is highly desirable, since it does not incur the additional latency of fetching the requested object from elsewhere. Thus, the user sees faster response times on a cache hit. Further, a cache hit does not incur any forward traffic to origin to fetch the content, reducing the "midgress" traffic costs for the CDN [21]. Another key goal of a CDN is to decrease the traffic on the origin site, since it reduces the operating costs for the content provider. A measure of that goal is the *origin offload factor*, which is simply the ratio of the content traffic served to users and the content traffic served by origin. It is easy to see that the origin offload is simply $1/(1 - \text{hit rate})$, i.e., the offload increases with hit rate. Thus, from multiple perspectives, a key goal of a CDN operator is to optimize the cache hit rates for each server and for each of the traffic classes that it hosts.

3 TRAFFIC CLASS CHARACTERISTICS

Each domain hosted on the CDN can be thought of generating a *request sequence* that consists of users requesting content from that domain. A domain also belongs to a traffic class, where each traffic class is a set of domains from a set of similar content providers, usually serving a specific content type. A request sequence for a traffic class is simply a sequence of requests received for some domain within that class. The major traffic classes in a modern CDN have content types that are either web sites, videos, images, or downloads. A large CDN may host tens of thousands of domains from thousands of content providers that form several hundred traffic classes. The main challenge in cache provisioning is the diversity of access patterns, object sizes, and resource requirements across different traffic classes.

3.1 Trace Collection

To illustrate this diversity of traffic classes in a quantitative fashion, we collected extensive traces from Akamai's production CDN for four representative traffic classes from 2 production cache servers in Akamai's CDN. The data set contains anonymized logs of content accessed by end-users. Each line in the production trace corresponds to a single request and contains a timestamp, the requested URL (anonymized), and the size of the object.

The four representative traffic classes each represent a major content type: web, downloads, videos and images. The web request trace is for HTML objects and associated objects such as css and

[2]This is also called the byte hit rate. There is another notion of hit rate called the object hit rate which is the percentage of requests that were found in cache. In this paper, we only focus on byte hit rates, though our work easily extends to object hit rates.

javascript files. The downloads request trace contains predominantly large objects consisting of software updates from a content provider. The image trace contains images embedded in web pages. The video trace contains video-on-demand (VOD) objects. Typically the objects belonging to the download and video traffic classes are several GB in size. But, in our traces these objects are smaller because a CDN fragments such large files to smaller chunks to avoid caching the entire object. CDNs typically cache only the byte-range that is requested and a few extra bytes, anticipating future requests (spatial locality). Videos are normally served in chunks that correspond to a few seconds of the video. Hence, CDNs typically only cache those chunks that are requested to avoid polluting the cache with content that has not been requested.

We collected the web and download traces from one production server and the image and video traces from the other production server. The characteristics of these traces are described in Table 1. We also collected additional web, download, image and video traces from two more servers to evaluate the accuracy of our cache models. These traces are described in Section 4.5.

Traffic class	Web	Download	Image	Video
Length of trace (days)	2.5	2.5	3.5	3.5
Arrival rate (req/s)	520	77	52	57
Traffic volume (Mbps)	333.0	216.5	8.4	361.5
Object count (millions)	10.3	0.7	1.3	6.1
Average object size (MB)	0.21	2.32	0.03	1.53

Table 1: Characteristics of the chosen traffic class traces.

3.2 Analysis of Traffic Classes

We compare and contrast the four chosen traffic classes based on their popularity distribution, object size distribution and cache hit rate. Figure 1 shows the popularity distribution of each traffic class. For our download traffic class, we see that 92% of all requests are for only 10% of the objects. For our web traffic class, 87% requests are for 10% of the objects. Both of these traffic classes have a long tail of popularity, indicating the presence of a large amount of unpopular content. For our image traffic class, 90% requests are for 30% of the objects. The *footprint* of a set of objects is the total bytes that need to be stored in cache to serve those objects from cache. From the traces we collected, we observe that we can achieve a high cache hit rate with a small footprint for our image class due to smaller object sizes. Finally, 90% of the requests in our video class are for 65% of the requested objects (i.e. requested video chunks), indicating that a larger footprint needs to be cached to achieve a good hit rate.

Figure 2 shows the CDF of the object size distribution for each traffic class. The x-axis is shown in log scale for clarity. In general, we see that the image and web traffic classes have predominantly small objects and both the download and video traffic classes have predominantly large objects. The extreme variability in object sizes across traffic classes makes it challenging to manage cache

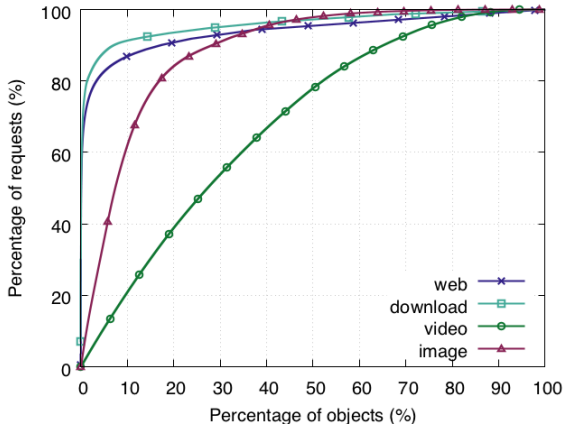

Figure 1: Popularity distribution of the 4 traffic classes.

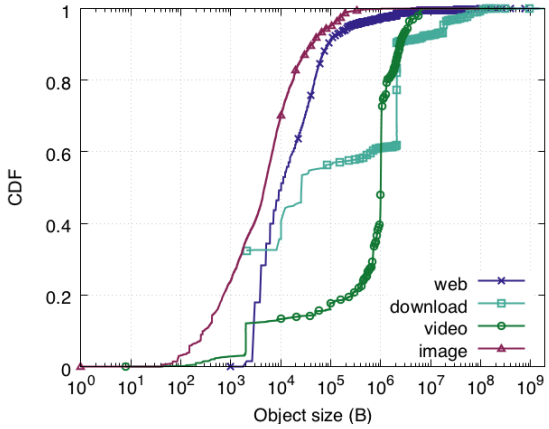

Figure 2: Object size distribution of the 4 traffic classes.

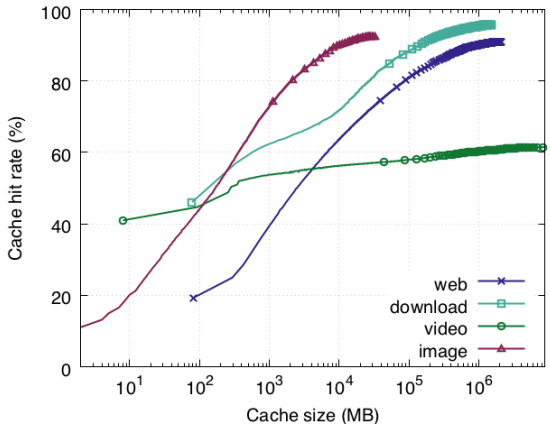

Figure 3: Hit rate curves of the 4 traffic classes.

resources across the CDN because different traffic classes require different amounts of cache space to achieve the same hit rate.

We compare the cache hit rate of the different traffic classes in Figure 3 by plotting their *hit rate curves (HRCs)* which gives the hit rate as a function of cache size. These hit rate curves were derived using footprint descriptors as shown in Section 4.2. The x-axis is shown in log scale for clarity. Note that we need a relatively smaller

cache to achieve a large cache hit rate for the image traffic class, when compared to the video traffic class which needs a much larger cache for the same hit rate. For example, to achieve a hit rate of 60%, the image class requires a cache space of about 2 GB, whereas the video class requires a cache space of about 1 TB.

The extreme variability in popularity, object size and caching performance highlights the importance of efficient cache provisioning when caching content belonging to different classes in a shared server. Note that two traffic classes of the same content type but different content providers may have different access characteristics and performance requirements. Thus, cache provisioning is done on a per-traffic-class basis, rather than on content types.

4 THEORY OF FOOTPRINT DESCRIPTORS

We now describe a concise space-time representation of a traffic class called footprint descriptor (FD) and derive a calculus for evaluating traffic mixes. Let ρ be a *request sequence* $\langle r_1, r_2, \cdots r_n \rangle$, corresponding to a traffic class τ, where each request r_i represents a user requesting an object belonging to that traffic class. Each request r_i has associated with it the timestamp t_i when the request was made, a unique identifier id_i of the object (such as its URL), and the size of the object s_i. We denote a subsequence ρ' of ρ to be the sequence of consecutive requests $\langle r_i, r_{i+1}, \cdots r_j \rangle$ of ρ, for some $i \leq j$. We call ρ' a *reuse subsequence* if the same object is accessed in the first and last request of the subsequence, but is not accessed elsewhere in ρ'. It is known that reuse subsequences have great significance in evaluating caching properties in other contexts [16]. They play an important role in FDs and their calculus as well.

4.1 Footprint Descriptors (FD)

A typical request stream ρ may have tens of millions of requests. We would like to efficiently summarize the attributes of ρ using the notion of a footprint descriptor FD, so that we may answer questions regarding the cacheability of ρ using FD. To that end, we define the footprint descriptor FD of ρ as the tuple $\langle \lambda, P^r(s,t), P^a(s,t) \rangle$, where λ is the traffic volume (in bits requested per second), $P^r(s,t)$ is the reuse-sequence descriptor function, and $P^a(s,t)$ is the all-sequence descriptor function. We describe each component of FD below.

1) The traffic volume λ is the average number of bits requested per second in the request sequence ρ. For a sequence ρ with n requests, the traffic volume $\lambda = (\sum_i b_i)/(t_n - t_1)$, where b_i denotes the number of bits requested by the i^{th} request and t_i denotes the time of the i^{th} request.

2) The reuse-sequence descriptor $P^r(s,t)$ is a *"space-time"* description of the reuse-subsequences ρ' of the ρ. In particular, it provides the joint probability distribution of the unique bytes s and the duration t for reuse subsequences ρ' of ρ. The unique bytes s accessed in ρ' is simply the sum of the sizes of all the unique objects requested in ρ'. The duration of t of ρ' is the difference in timestamps of the first and the last request in ρ'. Then, $P^r(s,t)$ is the probability that s unique bytes of content is requested in some reuse sequence ρ' of duration t. Given a request sequence ρ, $P^r(s,t)$ can be estimated by enumerating all its reuse sequences ρ' and tallying its unique bytes s and duration t. Note that the unique bytes and duration on the first access of any object is infinity. This accounts for the cold cache miss rate.

3) The all-sequence footprint descriptor $P^a(s,t)$ computes a similar statistic, but using any subsequence ρ' of ρ, i.e., ρ' is not necessarily a reuse subsequence and the first and last request of ρ' can be arbitrary. $P^a(s,t)$ is the probability that s unique bytes of content is requested in some subsequence ρ' of duration t. Given a request sequence ρ, $P^a(s,t)$ can be estimated by enumerating all its subsequences ρ' and tallying the unique bytes s and duration t.

4.2 Estimating cache properties from footprint descriptors

A footprint descriptor FD is a succinct representation of a request sequence that allows us to predict the hit rate performance that can be achieved for that sequence. We now show that the hit rate curve (HRC) of a request sequence can be derived from its FD in the context of the commonly-implemented Least-Recently-Used (LRU) caching algorithm. Most production CDNs use extensions of LRU, including Akamai [15].

Theorem 4.1. *The hit rate curve $HRC(s)$ for a request sequence ρ is a function that provides the hit rate achieved for ρ by an LRU cache of size s. The function $HRC(s)$ can be computed from the reuse-sequence descriptor $P^r(s,t)$ as follows.*

$$HRC(s) = \sum_{s' \leq s} \sum_{t} P^r(s', t).$$

Proof. Let $\rho' = \langle r_i, r_{i+1}, \cdots r_j \rangle$ be a reuse sequence of the request sequence ρ. That is, r_i and r_j are consecutive requests for the same object. For any cache of size s that uses LRU, the request r_j experiences a cache hit if and only if the unique bytes requested in ρ' is at most s, i.e., if the unique bytes is more than s the object requested by r_i that enters the cache will get evicted by the time the next request for the same object arrives at r_j. Thus, the hit rate $HRC(s)$ is simply the probability that a reuse sequence has unique bytes that is at most s, which in turn equals $\sum_{s' \leq s} \sum_{t} P^r(s', t)$. □

Besides LRU, the above theorem can be extended to other stack algorithms using the well-known relationship between unique bytes in a reuse sequence (called the stack distance) and hit rate [16].

4.3 A Calculus of Footprint Descriptors

The power of footprint descriptors is that it can support operations on request sequences that are important for cache provisioning. We present three key operations, addition, subtraction, and scaling. In Section 5, we show key applications of these operations in cache provisioning in the production network.

4.3.1 Addition. Let ρ_1 and ρ_2 be two request sequences that are independent and share no common objects[3]. The addition operator $\oplus$ can be applied to the two sequences to obtain a new sequence ρ which we represent as $\rho = \rho_1 \oplus \rho_2$. Request sequence ρ is obtained by interleaving ρ_1 and ρ_2 in accordance with the time stamp for the requests. We now show how the footprint descriptor $FD = \langle \lambda, P^r, P^a \rangle$ for ρ can be derived from the footprint descriptor $FD_1 = \langle \lambda_1, P^r_1, P^a_1 \rangle$ for ρ_1 and FD_2 for $\rho_2 = \langle \lambda_2, P^r_2, P^a_2 \rangle$.

[3]The assumption that two traffic classes share no common objects is reasonable in practice, since the objects belong to different domains from possibly different content providers. Such objects are treated as being different by the caching system.

The traffic volume λ of ρ is simply the sum of the traffic volumes of ρ_1 and ρ_2, i.e.,

$$\lambda = \lambda_1 + \lambda_2 \tag{1}$$

To compute the descriptor functions, we introduce some notation. Given a descriptor function $P(s,t)$, let $P(s \mid t)$ denote the conditional probability of unique bytes s given time duration t and let $P(t)$ denote the marginal distribution, i.e., $P(t) = \sum_s P(s,t)$. Thus,

$$P(s,t) = P(s \mid t)P(t). \tag{2}$$

The key observation of our calculus is that when two request sequences are combined, i.e., $\rho = \rho_1 \oplus \rho_2$, and we examine a subsequence ρ' of ρ of duration t with s unique bytes, the unique bytes s in ρ' either come from ρ_1 or ρ_2. Since the ρ_1 and ρ_2 have non-overlapping sets of objects, some s_1 come from ρ_1 and the remaining $s - s_1$ must come from ρ_2. Thus, to compute a descriptor function $P(s \mid t)$ for ρ from the descriptor functions $P_1(s \mid t)$ and $P_2(s \mid t)$ for ρ_1 and ρ_2 respectively, we can use the convolution operator to enumerate and add up the probabilities of all possible ways of obtaining s_1 unique bytes from ρ_1 and the remaining $s - s_1$ unique bytes from ρ_2. That is,

$$\begin{aligned} P(s \mid t) &= P_1(s \mid t) * P_2(s \mid t) \\ &= \sum_{s_1=0}^{s} P_1(s_1 \mid t) P_2(s - s_1 \mid t). \end{aligned}$$

Using this observation, we now compute $P^r(s \mid t)$ of ρ from $\langle P_1^r(s \mid t), P_1^a(s \mid t)\rangle$ of ρ_1 and $\langle P_2^r(s \mid t), P_2^a(s \mid t)\rangle$ of ρ_2 as follows. Let ρ' be a reuse sequence of ρ, i.e., the first and the last request of ρ' is for the same object. Let ρ' have s unique bytes and duration t. ρ' can be broken up into two subsequences ρ_1' of ρ_1 and ρ_2' of ρ_2. With probability $\frac{\lambda_1}{\lambda_1+\lambda_2}$ the first (and, last) request of ρ' is derived from ρ_1. That is, ρ' is composed of a reuse sequence ρ_1' and an arbitrary sequence ρ_2'. Similarly, with probability $\frac{\lambda_2}{\lambda_1+\lambda_2}$, ρ' is composed of a reuse sequence ρ_2' and an arbitrary sequence ρ_1'. Thus,

$$\begin{aligned} P^r(s \mid t) &= \frac{\lambda_1}{\lambda_1+\lambda_2}\left(P_1^r(s \mid t) * P_2^a(s \mid t)\right) \\ &\quad + \frac{\lambda_2}{\lambda_1+\lambda_2}\left(P_1^a(s \mid t) * P_2^r(s \mid t)\right), \end{aligned} \tag{3}$$

where $*$ denotes the convolution operator.

We can also compute $P^a(s \mid t)$ from $P_1^a(s \mid t)$ and $P_2^a(s \mid t)$. The computation is analogous to the above, except that ρ' can be an arbitrary sequence of ρ, not necessarily a reuse sequence. Since ρ' is composed of two arbitrary subsequences of ρ_1 and ρ_2, our computation involves only one convolution below.

$$P^a(s \mid t) = P_1^a(s \mid t) * P_2^a(s \mid t). \tag{4}$$

Note that the convolution operator $*$ arises naturally in our calculus, allowing us to leverage the powerful tools of Fourier analysis for the efficient computation of footprint descriptors. Putting together Equations 1, 2, 3, and 4 above, we can compute the FD of ρ from the FDs of ρ_1 and ρ_2 as we show in more detail in Algorithm 1.

Time Complexity. We can use Fourier analysis to speedup the computation of the addition operation. Let S and T be the maximum value buckets for s and t respectively. The addition operation can be performed in $O(TS \log S)$ time, since we need to perform 3 convolution operations in total for Equations 3 and 4 for every value of t, where each convolution takes $O(S \log S)$ time using Fast Fourier Transform algorithm (FFT) and t takes on T values.

Algorithm 1 Addition algorithm

Input: $FD_1 = \langle \lambda_1, P_1^r, P_1^a\rangle$, $FD_2 = \langle \lambda_2, P_2^r, P_2^a\rangle$, S and T be the buckets for s and t respectively
Output: $FD = \langle \lambda, P^r, P^a\rangle$

1: $\lambda = \lambda_1 + \lambda_2$
2: **for all** t $\in T$ **do**
3: $\quad P^r(t) = \frac{\lambda_1}{\lambda_1+\lambda_2} P_1^r(t) + \frac{\lambda_2}{\lambda_1+\lambda_2} P_2^r(t)$
4: $\quad P^a(t) = \frac{\lambda_1}{\lambda_1+\lambda_2} P_1^a(t) + \frac{\lambda_2}{\lambda_1+\lambda_2} P_2^a(t)$
5: $\quad$ **for all** s $\in S$ **do**
6: $\quad\quad P^r(s \mid t) = \frac{\lambda_1}{\lambda_1+\lambda_2}\left(P_1^r(s \mid t) * P_2^a(s \mid t)\right) + \frac{\lambda_2}{\lambda_1+\lambda_2}\left(P_1^a(s \mid t) * P_2^r(s \mid t)\right)$
7: $\quad\quad P^r(s,t) = P^r(s \mid t)P^r(t)$
8: $\quad\quad P^a(s \mid t) = P_1^a(s \mid t) * P_2^a(s \mid t)$
9: $\quad\quad P^a(s,t) = P^a(s \mid t)P^a(t)$

Inferring the individual hit rates of ρ_1 and ρ_2 after addition. Let the hit rate curves $HRC_1'(s)$ and $HRC_2'(s)$ represent the post-addition individual hit rate curves of ρ_1 and ρ_2 within $\rho_1 \oplus \rho_2$, i.e. $HRC_i'(s)$ is the post-addition hit rate of ρ_i when the traffic mix occupies cache capacity s. Then, $HRC_1'(s)$ and $HRC_2'(s)$ can be computed as follows.

$$\begin{aligned} HRC_1'(s) &= \sum_{s' \le s} \sum_t P^r(s' \mid t) P_1(t). \\ HRC_2'(s) &= \sum_{s' \le s} \sum_t P^r(s' \mid t) P_2(t). \end{aligned} \tag{5}$$

4.3.2 Subtraction. The subtraction operation models the cache provisioning operation of removing some traffic classes from the list of traffic classes served by a cache server. The result of that operation is that the request stream corresponding to those traffic classes are subtracted out. Given a request sequence ρ_1 that is a subsequence of ρ, we define $\rho_2 = \rho \ominus \rho_1$ to be the sequence obtained when the requests of ρ_1 are removed from ρ. We show how the FD of the resultant sequence ρ_2 can be obtained from the FDs for ρ and ρ_1. Note that we relate the request streams with the addition operator, i.e., $\rho = \rho_1 \oplus \rho_2$. Thus, the FD of ρ_2 can be computed by simply "inverting" Equations 1, 3, and 4 that we derived earlier for addition. By inverting Equation 1, we get

$$\lambda_2 = \lambda - \lambda_1 \tag{6}$$

The key idea for finding the descriptor functions is that the convolution $A = B * C$ can be inverted by using the frequency domain, i.e., $C = \mathcal{F}^{-1}(\mathcal{F}(A)/\mathcal{F}(B))$, where $\mathcal{F}$ and $\mathcal{F}^{-1}$ are Fourier transform and its inverse respectively. Thus, inverting Equation 4 we get

$$P_2^a(s \mid t) = \mathcal{F}^{-1}(\mathcal{F}(P^a(s \mid t))/\mathcal{F}(P_1^a(s \mid t))). \tag{7}$$

To find $P_2^r(s \mid t)$, we use Equation 3 to first compute $P_2^r(s \mid t) * P_1^a(s \mid t)$. Then, since we know $P_1^a(s \mid t)$, we can compute $P_2^r(s \mid t)$ by using the Fourier transform and its inverse as above. We provide the details of the subtraction algorithm in Algorithm 2.

$P_2^r(s \mid t)$ on line 8 in Algorithm 2 is computed from Equations 3 and 7.

Time Complexity. Let S and T be the maximum value buckets for s and t respectively. The subtraction operation can be performed

Algorithm 2 Subtraction algorithm

Input: $FD = \langle \lambda, P^r, P^a \rangle$, $FD_1 = \langle \lambda_1, P_1^r, P_1^a \rangle$, S and T be the buckets for s and t respectively
Output: $FD_2 = \langle \lambda_2, P_2^r, P_2^a \rangle$

$\lambda_2 = \lambda - \lambda_1$
for all $t \in T$ **do**
  $P_2^r(t) = \left(P^r(t) - \frac{\lambda_1}{\lambda_1+\lambda_2} P_1^r(t)\right) \frac{\lambda_1+\lambda_2}{\lambda_2}$
  $P_2^a(t) = \left(P^a(t) - \frac{\lambda_1}{\lambda_1+\lambda_2} P_1^a(t)\right) \frac{\lambda_1+\lambda_2}{\lambda_2}$
  for all $s \in S$ **do**
    $P_2^a(s \mid t) = \mathcal{F}^{-1}(\mathcal{F}(P^a(s \mid t))/\mathcal{F}(P_1^a(s \mid t)))$
    $P_2^a(s, t) = P_2^a(s \mid t) P_2^a(t)$
    $P_2^r(s, t) = P_2^r(s \mid t) P_2^r(t)$

in $O(TS \log S)$ time, since t takes on T values, we need to perform $O(1)$ Fourier (or, inverse Fourier) transforms for each value of t, and each Fourier (or, inverse Fourier) transform takes $O(S \log S)$ time using FFT.

4.3.3 Scaling. Suppose we wish to increase or decrease the traffic volume for a request sequence ρ_1. This operation is called scaling and we express the new request sequence $\rho = \rho_1 \otimes \tau$, where τ is the factor by which the volume is increased (decreased). We model the volume increase (decrease) as scaling the time variable, i.e., the time stamp of each request in ρ_1 is divided by the factor τ. This has the effect of decreasing (increasing) the inter-arrival times for the requests by τ when $\tau > 1$ ($\tau < 1$). We compute the FD of ρ from the FD of ρ_1 as follows.

$$\lambda = \lambda_1 \tau; P^r(s, t/\tau) = P_1^r(s, t); P^a(s, t/\tau) = P_1^a(s, t). \quad (8)$$

It is worth noting that scaling does not change the hit rate curve of $P^r(s, t/\tau)$ since the marginal distribution of $P^r(s) = P_1^r(s)$.

Time Complexity. Let S and T be the maximum value buckets for s and t respectively. The computations in Equation 8 can be performed in $O(ST)$ time, faster than addition or subtraction.

Note. It should be noted that the footprint descriptor calculus described in this section predicts the byte hit rate of a request sequence, which is the metric considered in this paper. The calculus works just the same to predict the object hit rate, with the slight modification that λ for a request sequence ρ is the arrival rate in requests per second rather than in bits per second. Also, note that the calculus allows us to compute the FD of complex traffic mixing operations by composing the three supported operators. For example, to add half the volume of class τ_1 to τ_2 and add a third of the resultant to τ_3, the FD of the final traffic mix $\tau = ((((\tau_1 \otimes 1/2) \oplus \tau_2) \otimes 1/3) \oplus \tau_3)$ can be computed efficiently using the calculus and FFT from the FD's of τ_1, τ_2 and τ_3.

4.4 A Simpler Footprint Descriptor (SFD)

In this section, we outline a simplification of footprint descriptors that makes implementations faster, at the cost of some theoretical rigor. Empirically, we observed that on production traces the descriptor functions $P^a(s, t)$ and $P^r(s, t)$ were statistically similar, i.e., the reuse sequences that start and end in a request for the same object, and arbitrary sequences that do not have the reuse property were statistically similar. The reason is that request sequences have requests for millions of different objects, and conditioning on starting and ending on a request for the same object does not alter the statistical behavior of the rest of the sequence very much. Therefore, a simpler footprint descriptor (SFD) is a tuple $\langle \lambda, P^r(s, t) \rangle$, i.e., the any-sequence descriptor $P^a(s, t)$ is dropped since it is similar to reuse-sequence descriptor $P^r(s, t)$. Having just one descriptor function makes computing SFD much simpler and faster for the addition and subtraction operations. For instance, if $P^a(s, t)$ is assumed identical to $P^r(s, t)$, Equation 4 simplifies to the following equation that requires just one convolution instead of two.

$$P^r(s \mid t) = P_1^r(s \mid t) * P_2^r(s \mid t). \quad (9)$$

Note that SFDs can be used to derive the cache hit rate curve HRC as described in Section 4.2, since it depends only on $P^r(s, t)$. For these reasons, we often use SFDs in practice, in lieu of FDs.

4.5 Validation of Footprint Calculus

In this section, we validate the addition operation described in Section 4.3 by computing the hit rate curves using the footprint calculus on SFDs described in Section 4.4. We then compare the calculus predictions with the hit rates obtained via cache simulations using the production traces, a simple baseline algorithm, as well as hit rates obtained directly from the production server. Further validation of addition and subtraction also appears as part of the case studies in Sections 5.1.1 and 5.1.3 respectively. We do not validate the scaling operator since the hit rate curve after scaling remains unchanged.

Additional traces for validating scalability of addition. Most production CDN servers serve at most two major traffic classes, i.e., the top two traffic classes account for most of the traffic from the server. Our initial set of traces described in Table 1 were from such servers that each served two classes. However, a few servers serve three or more traffic classes. To validate the addition of more traffic classes, we chose two additional production servers one that served four traffic classes across the four content types of web, image, video and download and another server that served nine traffic classes across three content types, namely web, video and download. These new traces let us evaluate the accuracy of the calculus when a larger number of traffic classes are mixed. The details of the additional traces are described in Tables 2 and 3 respectively. In Table 3 we show nine different traffic classes that have web, video, and download content from different content providers.

Traffic class	Web	Download	Image	Video
Length of trace (days)	1	1	1	1
Arrival rate (req/s)	223.53	51.04	216.42	180.01
Traffic volume (Mbps)	411.95	101.40	41.23	181.11
Object count (millions)	2.89	0.23	8.78	2.54
Average object size (MB)	0.3	0.4	0.02	0.24

Table 2: Characteristics of the 2^{nd} set of traces.

Baseline cache provisioning algorithm. We describe a baseline cache provisioning algorithm commonly used in operations that predicts the cache hit rate of a traffic mix using only the hit

Traffic class	Web-1	Web-2	Web-3	Web-4	Video-1	Video-2	Video-3	Video-4	Download
Length of trace (days)	8	8	8	8	8	8	8	8	8
Arrival rate (req/s)	168	16	6	3	53	21	4	3	22
Traffic volume (Mbps)	1105.4	114.7	1.8	0.002	292.6	112.4	21.6	14.9	234.5
Object count (millions)	15	1.6	0.05	0.07	11.7	2.5	1.9	0.6	1.9
Average object size (MB)	1.6	1.9	0.05	1.7	0.7	0.8	0.7	0.9	2.0

Table 3: Characteristics of the 3^{rd} set of traces.

rate curves of all traffic classes. For every value of the cache hit rate, the baseline algorithm determines the cache capacities for each traffic class from their respective hit rate curves and adds them up. The hit rate curve thus produced is the predicted curve for the traffic mix. For example, consider two traffic classes with hit rate curves $HRC_1(s)$ and $HRC_2(s)$ respectively. Then, the cache capacity required by the traffic mix to achieve hit rate h is predicted as $HRC_1^{-1}(h) + HRC_2^{-1}(h)$. This is repeated for all values of h to produce the hit rate curve of the mix.

The baseline algorithm described above is extremely simple and fast with time complexity $O(S)$, for hit rate curves having S cache size buckets. While simple, the baseline scheme does not account for the inter-arrival time distributions of the request sequences, and hence is an unreliable predictor of cache hit rates. We discuss the shortcomings in the following section.

FD calculus is superior to the baseline algorithm. We show via simulations using production traces that the calculus is more accurate at predicting hit rates of traffic mixes and is necessary for cache provisioning. For our simulation-based validation, we combine production traces corresponding to each traffic mix and perform a cache simulation on these merged traces for different cache sizes to obtain a hit rate curve. We call this the "simulated" hit rate curve. We also compute the hit rate curve of the traffic mix using the calculus. We call this the "calculated" hit rate curve. Finally, we compute the hit rate curve using the baseline algorithm and we call this the "baseline" hit rate curve.

Traffic mixes	Average error of baseline, %	Average error of calculus, %
web+download (Table 1)	0.24	0.13
video+image (Table 1)	0.63	0.10
Traffic classes in Table 2	10.2	0.28
Traffic classes in Table 3	11.8	0.34

Table 4: Average prediction error of the baseline algorithm vs. the FD calculus.

In Table 4, we present the average error of the baseline and the calculated hit rates with respect to the simulated values. We present the error for performing the addition operation for the web and download classes, and video and image classes in Table 1 and the addition of all classes in Tables 2 and 3. While the baseline algorithm has a small error for web+download and video+image from Table 1, the large difference and variability in error between the baseline algorithm and the calculus in general, and the consistently small average error of the calculus, highlight the need for the more accurate calculus in predicting the effects of traffic mixing.

We now discuss another scenario where the calculus is superior to the baseline algorithm. Very often, CDN operators need to predict the effects of traffic scaling on traffic mixing, to better provision caches under varying traffic conditions. For instance, CDN operators would like to know the hit rate of a traffic mix when the traffic volumes of one or more traffic classes are varied. We show that under such circumstances, the calculus (using the scaling operation in conjunction with addition) provides more reliable outputs than the baseline algorithm which responds erratically to traffic scaling.

To illustrate this scenario, we consider the traffic mix of the traffic classes in Table 2. We consider two scenarios, 1) the traffic classes are mixed at their current traffic volumes (unscaled versions) and 2) we scale the traffic volume of the download traffic class up by 20 times, to 2028 Mbps, and predict the cache hit rate under traffic mixing in this new scenario (scaled versions). In Figure 4, we plot the "simulated", "calculated" and "baseline" hit rate curves without scaling. We also plot the hit rates curves of the traffic mix (after scaling the download traffic class) predicted by the calculus ("calculated-scale") and the baseline algorithm ("baseline-scale"). We refer to traffic mix web+image+video+download from Table 2 as "wivd". We also zoom in on the x-axis for clarity.

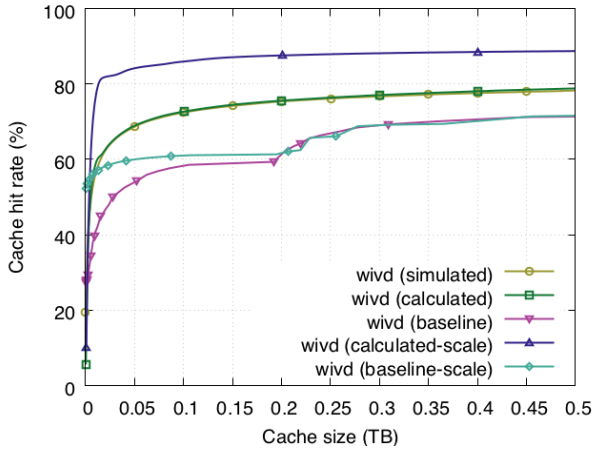

Figure 4: Hit rate curves of the traffic mix in Table 2 before and after scaling the download traffic class by a factor of 20.

From Figure 4, we see that the hit rate curve predicted by the calculus, wivd(calculated), closely matches the simulated hit rate

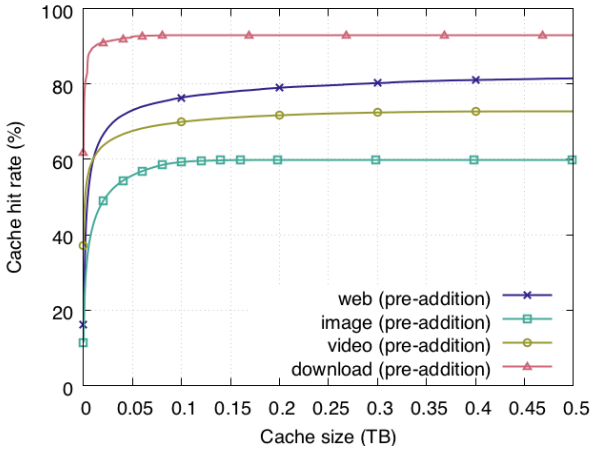

Figure 5: Hit rate curves of traffic classes in Table 2.

curve, wivd(simulated), with an average error of 0.28%. However, the error between the simulated curve, wivd(simulated), and the baseline scheme, wivd(baseline), without scaling is much higher at 10.2% on average, as previously discussed. After scaling the download traffic class up by a factor of 20, we see that the hit rate of the traffic mix, wivd(calculated-scale), increases as expected because the download traffic class dominates the mix (see the hit rate curve of the download traffic class labelled pre-addition in Figure 5). On the contrary, there is little change in the estimate of the baseline algorithm after scaling, wivd(baseline-scale), and the estimated hit rate is less than that of the calculus by 17.8% on average. This is because the baseline algorithm is unaware of the changes in the inter-arrival times of requests after scaling, which the calculus takes that into account.

We do similar comparisons for the other sets of traces. We scale up 20 times the image traffic class in the video+image mix in Table 1. We also scale up 20 times the download traffic class in both the web+download mix in Table 1 and the mix of traffic classes in Table 3. We find that "baseline-scale" is 8.6% less than "calculated-scale" on average in the case of video+image in Table 1, 4.2% less than "calculated-scale" on average in the case of web+download in Table 1 and 23.1% less than "calculated-scale" on average for the traffic mix in Table 3. This evaluation further emphasizes the need for the more accurate calculus that accounts for both the spatial and temporal interactions between traffic classes during traffic mixing and predicts hit rates accurately.

FD calculus matches well with production setting. We perform a production validation by comparing the hit rates produced by our calculus with the hit rates measured directly from the production server serving the required mix of traffic over the same time period. Measuring it directly from the production server measures the actual production caching software implemented on the deployed hardware. This validation method validates only one point on the HRC, the point that corresponds to the actual cache size of the production server. We measured the average hit rate reported by the production servers corresponding to the traffic mixes in Tables 1, 2 and 3. The results are in Table 5.

From Table 5, we see that that calculus predicts the cache hit rate of the traffic mixes considered with a prediction error of at most 2.5% in all cases. The difference in hit rates is in part due to the fact that the production servers were intermittently used to serve small amounts of other traffic classes by the mapping system. Moreover, the footprint calculus models a pure LRU algorithm while the production system has extra optimizations that we do not currently model. The low prediction errors from our production validation further strengthens the case for using footprint descriptor calculus to provision caches in CDNs.

Traffic mix	Cache size (TB)	Hit rate from calculus, %	Hit rate from server, %
web+download (Table 1)	3.0	86.6	84.1
video+image (Table 1)	3.7	37.7	39.3
Traffic classes in Table 2	2.0	79.9	77.4
Traffic classes in Table 3	3.7	68.6	67.5

Table 5: Production validation.

5 APPLYING FOOTPRINT DESCRIPTORS IN A PRODUCTION CDN

In this section, we provide case studies to show how footprint descriptors plays a key role in cache provisioning in CDNs.

5.1 Traffic mix evaluation service

Several what-if questions arise in the process of determining the optimal traffic mix of traffic classes that is served by a cache server. A traffic mix evaluation service can provide detailed information about cache occupancy and hit rates when various traffic classes are combined together. The output of the service prevents poor mixing choices from going into effect in any cache server. We used footprint descriptors to implement the service that is currently in use by the operations staff at Akamai in a limited beta setting. The service computes footprint descriptors for each traffic class hosted on the CDN using the techniques described in Section 4. The service keeps a database of all the cache servers and their properties, including the cache space available. We show how FDs are used to answer key questions that arise in the context of traffic mixing.

5.1.1 Estimating space requirement of a traffic mix. Given a set of traffic classes with their respective traffic volumes, a CDN operator might be interested in the cache capacity required by the traffic mix to provision servers to achieve a target hit rate. The traffic mix evaluation service computes the output using the following steps.

(1) The footprint descriptor of all the traffic classes of interest are computed efficiently using the techniques outlined later in Section 5.3. Each traffic class is then scaled to the required traffic volume using the $\otimes$ described in Section 4.3.3, and added together using the $\oplus$ described in Section 4.3.1, which gives the footprint descriptor of the traffic mix.

(2) Using Theorem 4.1, the HRC of the traffic mix is computed from its footprint descriptor.

(3) Given the HRC and the target hit rate h, the required cache size s is determined such that the hit rate $HRC(s) \geq h$.

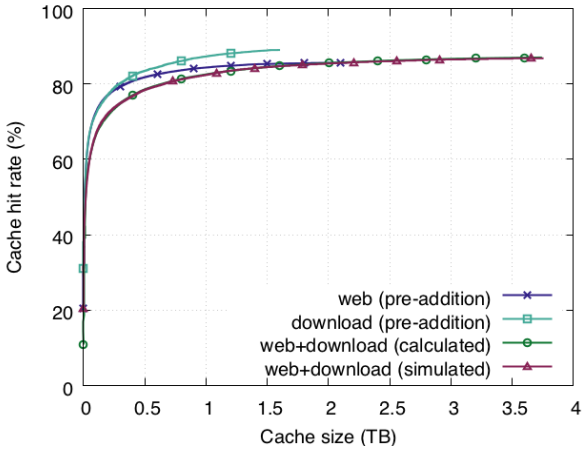

Figure 6: Addition operation on web and download traffic classes in Table 1.

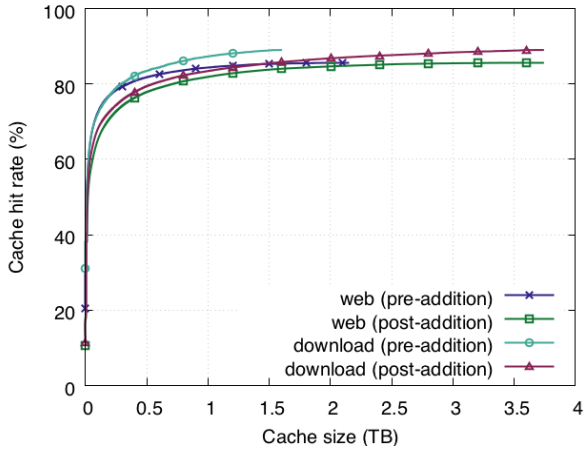

Figure 7: Pre-addition and post-addition HRCs of web and download traffic classes in Table 1.

Validation of functionality. We consider one example, the traffic mix of the web and download traffic classes in Table 1 to illustrate the use of footprint descriptor calculus. Figure 6 shows the HRCs of the web and download traffic classes labeled "pre-addition", as well as the HRC of the mix as computed by the calculus labeled "calculated". To validate the correctness of the HRC of the mix, we merge the request traces for both the classes by interleaving the requests in the ascending order of their time stamp. The HRC derived from the merged trace using cache simulations is shown in Figure 6, labeled "simulated". As can be seen, the calculated and simulated HRCs match closely with an average error of 0.13%, thus validating that the addition algorithm works on production traces.

5.1.2 Predicting the outcome of traffic mixing in a given server. For this use case, the operator provides the cache size of the server and the traffic classes with their respective traffic volumes to be mixed in that server. With this input, the service does the following:

(1) The footprint descriptors of all the traffic classes of interest are computed, scaled to the required traffic volume using the $\otimes$ described in Section 4.3.3, and added together using the $\oplus$ described in Section 4.3.1, to give the footprint descriptor of the traffic mix.

(2) Using Theorem 4.1, the HRC of the traffic mix is computed. Using Equation 5, we also compute the (post-addition) HRCs of each traffic class in the mix.

(3) Given the total cache size and the HRC of the mix, we obtain the hit rate of the mix. Using the total cache size and the (post-addition) HRC of the individual traffic classes, we obtain the hit rates of each traffic class in the mix.

(4) Using the post-addition hit rates of each traffic class and their (pre-addition) HRCs, we can obtain the cache space occupied by each traffic class after the addition.

Thus, our service predicts the overall cache hit rate for a given traffic mix, individual cache hit rates for each traffic class after mixing, and the cache space occupied by each traffic class in the cache after mixing. This information is valuable to differentiate between good and poor mixes.

Validation of functionality. We continue with the above example of mixing web and download traffic classes described in Table 1. We assume here that the cache server has a total cache size of 1 TB. As shown in Figure 6, the HRC labeled "web+ download(calculated)" says that the cache will get 82% overall hit rate. The HRC labeled "web+download(simulated)" confirms the correctness of the value. Figure 7 shows the pre-addition and post-addition HRCs for both the web and download traffic classes. Plugging in 1 TB as the cache space, we see that within the mix, the web traffic class gets a hit rate of 80.5%, while the download traffic class gets a hit rate of 84%. Now, performing a reverse lookup for these hit rates in the original HRCs for these two traffic classes in Figure 7, we see that the web traffic class occupies 300 GB in cache, while the download traffic class occupies the remaining 700 GB.

5.1.3 Splitting domains in a traffic class. A traffic class consists of user requests for content hosted on a specific set of domains. Often, domains need to be removed from traffic classes to achieve better load balancing. In such situations, it is important to predict how the caching characteristics of the traffic class would change if some domains are removed. This prediction can be performed using the subtraction operator $\ominus$ described in Section 4.3.2. We first collect the traces for the domains to be removed from the original traffic class, and we compute the footprint descriptor for the resulting traffic class using the subtraction operation. The resultant footprint descriptor can be used to compute the hit rates after the split.

Validation of functionality. We plot the results of this operation in Figure 8. Note that the x-axis has been truncated for clarity of presentation. In this instance, we want to remove a certain set of domains from the video traffic class in Table 1. In Figure 8, video_complete indicates the hit rate curve for the entire traffic class, video_set2(calculated) is the hit rate curve of the remaining domains when video_set1 is removed. video_set2(simulated) is the hit rate curve of video_set2 computed via cache simulations. We see that video_set2(calculated) compares very well with video_set2(simulated), with an average error of 0.05%, confirming that $\ominus$ works well on production traces as well.

5.2 Hit rate targets with cache partitioning

In many situations, it is necessary to guarantee a certain hit rate performance for a subset of traffic classes in a traffic mix, while

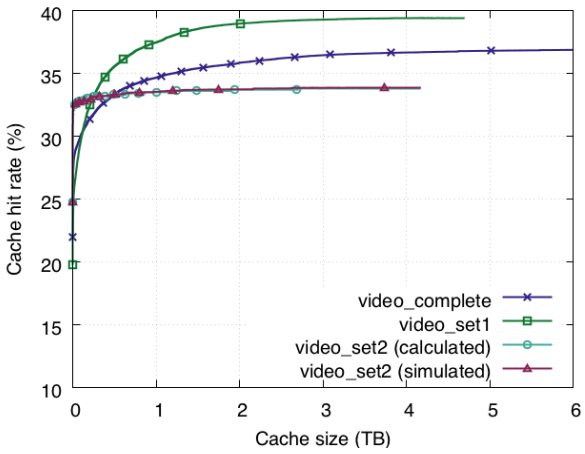

Figure 8: Subtracting a subset of domains from the video traffic class in Table 1.

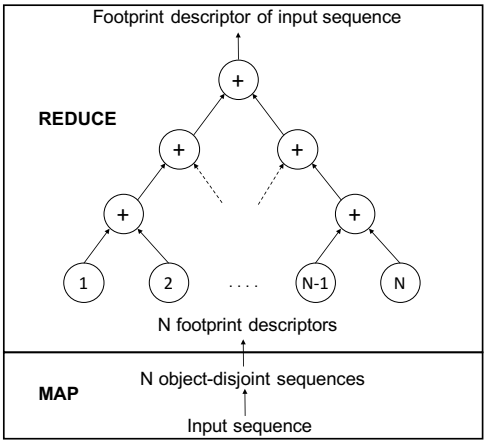

Figure 9: Map-reduce framework to parallelize footprint descriptor computation.

ensuring that the traffic mix does well overall. The traffic mix evaluation service uses cache partitioning to ensure that each traffic class meets its target hit rate while making the best use of the remaining cache space to maximize the overall cache hit rate.

The aims of cache partitioning are twofold: *(1)* ensure that each traffic class gets at least its requisite cache space, and *(2)* any leftover space is assigned appropriately to the traffic classes, so that the hit rate of the cache as a whole is maximized.

The first aim is achieved using our calculus as follows: *(1)* Using Theorem 4.1, HRCs are computed from the footprint descriptor for all traffic classes; *(2)* A reverse lookup is performed on the HRCs, to get the cache size needed for the given target hit rate. This is the requisite partition size for each traffic class.

Towards the second aim, the leftover cache space is divided into a number of smaller blocks. Each block is added incrementally to the traffic classes, using the following method: *(1)* Compute the traffic-weighted first derivative of the HRC of each traffic class at the point where size of the cache is equal to its current partition size; *(2)* Identify the traffic class with the highest value of this first derivative. This class can provide the most benefit in cache hit rates if it is given the block being considered; *(3)* Assign the block to the class with the highest derivative, and increase its partition size.

All the remaining cache space is assigned to traffic classes in this way. The resulting partition of cache space maximizes the server's cache hit rate, while meeting the hit rate targets of the individual traffic classes. Once the partition sizes are determined, the partitions are implemented as separate virtual LRU caches within the given server. The method of allocating leftover cache space is similar to the utility maximization approach described in [20] where the first derivative of the HRC is the utility function.

Validation of functionality. We continue with our example of mixing web and download traffic classes from Table 1. We assume here that the cache server has a total cache size of 1 TB. First we consider targets of 85% hit rates to both the traffic classes. The (post-addition) HRCs for the individual traffic classes suggest that neither class can achieve this hit rate, since the cache space available is 1 TB. Thus, partitioning to achieve 85% hit rate is infeasible. Next, we consider a hit rate target of 83% for the web class and 75% to the download class. These targets are chosen to illustrate how footprint descriptors can be used for cache partitioning. A reverse lookup of hit rate in Figure 6 shows that the requisite cache spaces for the web and download classes are 625 GB and 125 GB, respectively. This leaves 250 GB of available cache space unassigned to either class. The first derivative method [20] is used to determine the assignment of the leftover 250 GB, and it assigns all of it to the download class. Thus, the 1 TB cache is partitioned into 625 GB and 375 GB partitions. The hit rate of the web class meets its target of 83%, while the download class achieves a better than target hit rate of 82.5%. As observed in [20], cache partitioning can be used to improve the overall cache hit rate when multiple traffic classes share the cache space. Indeed the cache hit rate after mixing the web and download traffic classes increases from 82% without explicit partitioning to 83% with partitioning.

5.3 Parallelizing the computation of FDs

Before footprint descriptors can be used in operational decision making as illustrated in previous sections, they need to be computed from request sequences. Typically, request sequences over observation periods of a couple of days to a week are processed to compute footprint descriptors. These sequences may contain over a billion distinct URLs. Such industrial-strength computation is a heavyweight proposition both in terms of memory and CPU cycles. We develop a novel map-reduce-based framework that uses the $\oplus$ operation of our calculus to parallelize the computation of footprint descriptors for large request sequences. The procedure is below.

Map phase. Split the input request sequence into N smaller request sequences that share no objects between them. We accomplish this by hashing the URL of each request into N buckets, each bucket representing a smaller request sequence.

Reduce phase. Compute the footprint descriptors of the N smaller sequences in parallel. Using $\oplus$, add the footprint descriptors for the smaller sequences in parallel, until we are left with one footprint descriptor. This is the footprint descriptor of the input stream.

The complete framework is shown in Figure 9 where, $\oplus$ is the addition algorithm described in Section 4.3.1. The reduce phase begins by computing the footprint descriptors for the N object-disjoint

smaller sequences (the N leaves) and adding them in parallel, bottom up, until we obtain the footprint descriptor of the original input sequence which is the root of the tree. This framework parallelizes a seemingly serial process and can speed up computation that increases nearly linearly in the number of compute nodes.

Validation. We implement this parallel footprint descriptor computation algorithm in Amazon EMR [1]. We run map-reduce jobs in a cluster with up to 32 nodes. We use m3.xlarge machines in all experiments. The reported time is the elapsed time of the map-reduce job recorded by the cluster.

To evaluate the speedup due to parallelization, we use a much larger set of production traces corresponding to a video traffic class collected from 29 servers over a period of 8 days. The request rate is 1,234 req/s and the traffic volume is 6.76 Gbps. The trace contains 227.3 million objects with an average object size of 0.7 MB.

We observe that it takes 420 minutes to compute the footprint descriptor of the video traffic class without parallelization and 28 minutes with a 16-way parallelization and 16 minutes with a 32 way parallelization, that is a speed up of 15 and 26.2 respectively with no impact on the accuracy of the output.

6 RELATED WORK

Caching has been active area of research for the past decades. We only review the work on caching that is most closely related to our work. Much of the closely-related prior work fall into the realm of cache modeling and cache composition.

Cache modeling. We subdivide the relevant work on cache modeling into empirical modeling based on stack distance, which is the number of unique bytes in a request subsequence, and theoretical modeling that assumes certain statistical properties of the request sequences. Stack distance-based caching models were first proposed in [16]. Stack distance is useful to compute hit rate curves that plot the cache hit rate as a function of cache size. The simple algorithm proposed in [16] has high space and time overheads and is infeasible for large input sequences such as those in the CDN context. Subsequently, several time and space-efficient algorithms have been proposed in the literature, such as those in [3, 18, 24, 26]. *However, none of the stack distance algorithms in the literature provide a calculus that allows operations such as addition, subtraction, and scaling, a key necessary ingredient that the footprint descriptor calculus provides for CDN cache provisioning.* Several theoretical models have been proposed to predict cache hit rates as early as the 1970's [11, 14]. Of particular interest is the work in [5, 9] that relates the cache size with the cache hit rate and cache eviction age for IRM traffic. More recent work [7, 10] extend such ideas to caching policies beyond LRU. *In contrast, our work is focused on modeling and predicting properties of arbitrary production workloads of a CDN that are hard to capture with IRM-like models.*

Cache composition. Cache composition has been studied in the context of CPU caches, where applications running in multi-processor machines share the CPU cache. More recently, cache composition has been studied in the context of memory caches and storage systems. Analytical models have been developed to predict cache hit rates of time-shared systems, such as those in [2, 22, 23]. These models compute the hit rate of a shared cache in the presence of context switching. The authors in [4] propose three cache composition models with varying degrees of accuracy that predict the impact on cache hit rates when two non-overlapping applications run together in the shared L2 cache. The model presented in [8] goes one step further to characterize overlapping data in multi-threaded programs by predicting the overlapping footprint based on how threads interleave when running concurrently. Some other related work [25, 27, 28] develop models that more accurately characterize memory footprint of processes using novel sampling techniques and derive hit rates from those footprints. A more recent work [12] develops a kinetic model of LRU cache, based on the average eviction time (AET). *In contrast to prior work in cache composition, we support a wider range of composition operations on traffic classes, including addition, subtraction and scaling. Unlike prior work, our work is based on a theoretical sound foundation of footprint descriptor calculus. Further, our empirical approach is focused to the specific challenges in CDN cache provisioning.*

7 CONCLUSIONS AND FUTURE WORK

Cache provisioning in CDNs is challenging because of the diverse requirements imposed by diverse traffic classes. It is also challenging due to the immense scale of the operations, both in terms of traffic volumes and the large network of cache servers. Footprint descriptors provide a simple and elegant way of capturing the caching properties of a traffic class. The theory of footprint descriptor calculus allows us to add, subtract, and scale traffic classes to answer important "what-if" questions that arise in CDN operations. The connections to Fourier analysis that allow footprint descriptors to be manipulated in the "frequency domain" is also of interest. Footprint descriptors are well-suited for use in production network operations, since as we show the prediction error is under 2.5% for key use cases on the servers considered.

Our work also leaves several interesting open problems for future work. We briefly highlight two such directions. Given the footprint descriptor of a request sequence entering a cache, can we derive the footprint descriptor of the requests that result in a cache miss? Such an extension will allow us to analyze a hierarchy of caches using footprint descriptors. Another challenging problem of immense interest is how to load balance traffic classes across servers or groups of servers such that the overall miss traffic of the traffic mix is minimized, without overloading the servers or the network? In this paper, we discuss how the traffic mix evaluation service could be used to predict the effects of a traffic mix. But this requires manual intervention by an operator. We are working on developing optimization models based on footprint descriptor calculus to automate the evaluation of traffic mixes such that the overall miss rate is minimized while the servers are load balanced.

ACKNOWLEDGMENTS

We thank our anonymous reviewers for their comments that helped improve the quality of the work. We are also grateful to our shepherd Sergey Gorinsky for his great feedback on the paper. This research was supported in part by NSF grant CNS-1413998. Opinions expressed in this paper are solely that of the authors and not necessarily that of Akamai or UMass or any other organization.

REFERENCES

[1] Amazon EMR, https://aws.amazon.com/emr/.

[2] A. Agarwal, J. Hennessy, and M. Horowitz. An analytical cache model. *ACM Trans. Comput. Syst.*, 7(2):184–215, May 1989.

[3] G. Almási, C. Caşcaval, and D. A. Padua. Calculating stack distances efficiently. In *ACM SIGPLAN Notices*, volume 38, pages 37–43. ACM, 2002.

[4] D. Chandra, F. Guo, S. Kim, and Y. Solihin. Predicting inter-thread cache contention on a chip multi-processor architecture. In *Proc. 11th Int. Symp. High-Performance Computer Architecture*, pages 340–351, Feb. 2005.

[5] 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.

[6] F. Chen, R. K. Sitaraman, and M. Torres. End-user mapping: Next generation request routing for content delivery. In *ACM SIGCOMM Computer Communication Review*, volume 45, pages 167–181. ACM, 2015.

[7] M. Dehghan, L. Massoulie, D. Towsley, D. Menasche, and Y. C. Tay. A utility optimization approach to network cache design. In *Proc. IEEE INFOCOM 2016 - The 35th Annual IEEE Int. Conf. Computer Communications*, pages 1–9, Apr. 2016.

[8] C. Ding and T. Chilimbi. A composable model for analyzing locality of multi-threaded programs. Technical report, Technical Report MSR-TR-2009-107, Microsoft Research, 2009.

[9] R. Fagin. Asymptotic miss ratios over independent references. *Journal of Computer and System Sciences*, 14(2):222–250, 1977.

[10] M. Garetto, E. Leonardi, and V. Martina. A unified approach to the performance analysis of caching systems. *ACM Transactions on Modeling and Performance Evaluation of Computing Systems*, 1(3):12, 2016.

[11] E. Gelenbe. A unified approach to the evaluation of a class of replacement algorithms. *IEEE Transactions on Computers*, C-22(6):611–618, June 1973.

[12] X. Hu, X. Wang, L. Zhou, Y. Luo, C. Ding, and Z. Wang. Kinetic modeling of data eviction in cache. In *2016 USENIX Annual Technical Conference (USENIX ATC 16)*, pages 351–364. USENIX Association, 2016.

[13] P.-H. Kamp. Varnish LRU architecture, June 2007. available https://www.varnish-cache.org/trac/wiki/ArchitectureLRU accessed 09/12/16.

[14] W. F. King. Analysis of paging algorithms. In *IFIP Congress, Ljublanjana, Yugoslavia, Aug 1971*.

[15] B. M. Maggs and R. K. Sitaraman. Algorithmic nuggets in content delivery. *SIGCOMM Comput. Commun. Rev.*, 45(3):52–66, July 2015.

[16] 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.

[17] F. Memon. A guide to caching with nginx and nginx plus, July 2015. https://www.nginx.com/blog/nginx-caching-guide/ accessed 06/18/17.

[18] Q. Niu, J. Dinan, Q. Lu, and P. Sadayappan. Parda: A fast parallel reuse distance analysis algorithm. In *Proc. IEEE 26th Int. Parallel and Distributed Processing Symp*, pages 1284–1294, May 2012.

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

[20] M. K. Qureshi and Y. N. Patt. Utility-based cache partitioning: A low-overhead, high-performance, runtime mechanism to partition shared caches. In *Proc. 39th Annual IEEE/ACM Int. Symp. Microarchitecture (MICRO'06)*, pages 423–432, Dec. 2006.

[21] R. K. Sitaraman, M. Kasbekar, W. Lichtenstein, and M. Jain. Overlay networks: An akamai perspective. *Advanced Content Delivery, Streaming, and Cloud Services*, pages 305–328, 2014.

[22] G. E. Suh, S. Devadas, and L. Rudolph. Analytical cache models with applications to cache partitioning. In *Proceedings of the 15th International Conference on Supercomputing*, ICS '01, pages 1–12, New York, NY, USA, 2001. ACM.

[23] D. Thiebaut and H. S. Stone. Footprints in the cache. *ACM Trans. Comput. Syst.*, 5(4):305–329, Oct. 1987.

[24] C. A. Waldspurger, N. Park, A. T. Garthwaite, and I. Ahmad. Efficient MRC construction with SHARDS. In *FAST*, pages 95–110, 2015.

[25] X. Wang, Y. Li, Y. Luo, X. Hu, J. Brock, C. Ding, and Z. Wang. Optimal footprint symbiosis in shared cache. In *Cluster, Cloud and Grid Computing (CCGrid), 2015 15th IEEE/ACM International Symposium on*, pages 412–422. IEEE, 2015.

[26] J. Wires, S. Ingram, Z. Drudi, N. J. Harvey, A. Warfield, and C. Data. Characterizing storage workloads with counter stacks. In *OSDI*, pages 335–349, 2014.

[27] X. Xiang, B. Bao, T. Bai, C. Ding, and T. Chilimbi. All-window profiling and composable models of cache sharing. In *ACM SIGPLAN Notices*, volume 46, pages 91–102. ACM, 2011.

[28] X. Xiang, C. Ding, H. Luo, and B. Bao. HOTL: a higher order theory of locality. In *ACM SIGARCH Computer Architecture News*, volume 41, pages 343–356. ACM, 2013.