Scalable Techniques for Memory–efficient CDN Simulations*

Purushottam Kulkarni, **Prashant Shenoy** and **Weibo Gong**†

Computer Science Dept,
University of Massachusetts,
Amherst, MA 01003
{purukulk,shenoy}@cs.umass.edu

†ECE Department
University of Massachusetts,
Amherst MA 01003
gong@ecs.umass.edu

Abstract

Since CDN simulations are known to be highly memory-intensive, in this paper, we argue the need for reducing the memory requirements of such simulations. We propose a novel memory-efficient data structure that stores cache state for a small subset of popular objects accurately and uses approximations for storing the state for the remaining objects. Since popular objects receive a large fraction of the requests while less frequently accessed objects consume much of the memory space, this approach yields large memory savings and reduces errors. We use bloom filters to store approximate state and show that careful choice of parameters can substantially reduce the probability of errors due to approximations. We implement our techniques into a user library for constructing proxy caches in CDN simulators. Our experimental results show up to an order of magnitude reduction in memory requirements of CDN simulations, while incurring a 5-10% error.

1 Introduction

Content distribution networks are increasingly being used to disseminate data in today's Internet. A *content distribution network (CDN)* is a collection of proxies that act as intermediaries between the origin servers and the end clients. Proxies in a CDN cache frequently accessed data from origin servers and serve requests for these objects from the proxy closest to the end-user. Proxies can also prefetch, transform, and process content before delivering it to the end-user. By providing these services, a CDN has the potential to reduce the load on origin servers and the network and also improve client response times. Due to these benefits, the design of content distribution networks is an active area of research, and numerous techniques for architecting CDNs are being studied by researchers. Typically, these researchers have used two different methods to evaluate new research ideas—prototype implementation in a laboratory testbed and simulations. While prototyping and experimentation on laboratory testbeds has several advantages, these testbeds are typically small and consist of a few machines or tens of machines. In contrast, actual CDNs may consist of hundreds or thousands of proxies [13], and hence, it is not possible to study the scalability of new techniques using a small-scale testbed. Consequently, researchers have resorted to simulations to evaluate the behavior of new CDN mechanisms and policies in large-scale settings.

*This research was supported in part by NSF grants CCR-0098060, EIA-0080119 and CCR-0219520.

In general, simulation of a large CDN is highly memory- and compute-intensive. The memory-intensive nature arises from the need to simulate a disk cache at each proxy—the larger the number of objects in the cache and the larger the number of proxies in the CDN, greater the memory requirements for simulating the CDN. The compute-intensive nature arises from the need to simulate a large number of end-user requests and various inter-proxy and proxy-server interactions. In general, larger the number of end-user requests that are simulated and larger the size of the CDN, greater are the computational requirements. The following example illustrates these simulation requirements for a typical CDN.

Example 1 *Consider a content distribution network with 1000 proxies. Assume that each proxy processes a million requests per day from its end-users and that 500,000 of these requests are for unique objects (the remaining requests are assumed to access objects already in the proxy cache). In such a scenario, a day-long simulation will require the CDN to process a total of 1 billion requests and maintain cache state for 500 million unique objects (0.5 million objects per proxy cache). A simulated cache needs to store several pieces of information as part of the object-specific state; the state includes the object ID, the size of the object, its last modification time, the type of the object, etc. Thus, if we conservatively assume that 20 bytes are required to maintain the cached state of each object, then the total memory requirements for 500 million objects is 10GB. This is beyond the memory capacity of typical compute-servers (except for very high-end servers that are available today). Similarly, the computational requirements for processing a billion requests can overwhelm most servers, necessitating long running simulations.*

Thus, the memory and computational requirements of CDN simulations are a key hurdle for many researchers. In one recent work, due to the hardware constraints imposed by our machines (fast dual processors with 1GB memory), we were limited to simulating only a small proxy group within a larger CDN—we found that the memory bottleneck was reached when we simulated a 25 proxy group, each processing 1 million user requests [22]. Other researchers face similar hurdles in their work, and consequently, resort to simulating smaller CDNs or simulating the system for smaller durations. Neither approach allows the scalability of the techniques under consideration to be thoroughly studied. These arguments motivate the need to reduce the memory and computational requirements of CDN simulations. In this paper, we focus on techniques for scaling CDN simulations along the memory dimension; techniques for scaling simulations along the computation dimension are beyond the scope of this paper and are being pursued in a separate piece of work.

To address this problem, we propose a novel memory-efficient data structure that stores cache state for a small subset of popular objects accurately and uses approximations for storing the state of the remaining objects. Our design is based on the following observation: popular objects receive a large fraction of the requests, while less frequently accessed objects consume much of the memory space. Maintaining accurate state for a large fraction of the requests and approximate state for the objects consuming much of the space, yields large memory savings and reduces errors. We use bloom filters to store approximate state and show that careful choice of parameters can substantially reduce the probability of errors due to approximations. We implement our techniques into a user library for constructing proxy caches in CDN simulators. Our experimental results show up to an order of magnitude reduction in memory requirements of CDN simulations, while incurring a 5–10% error.

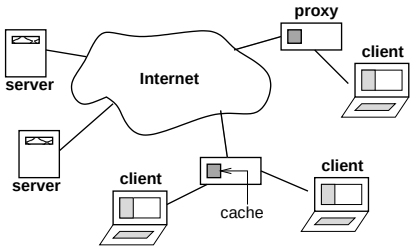


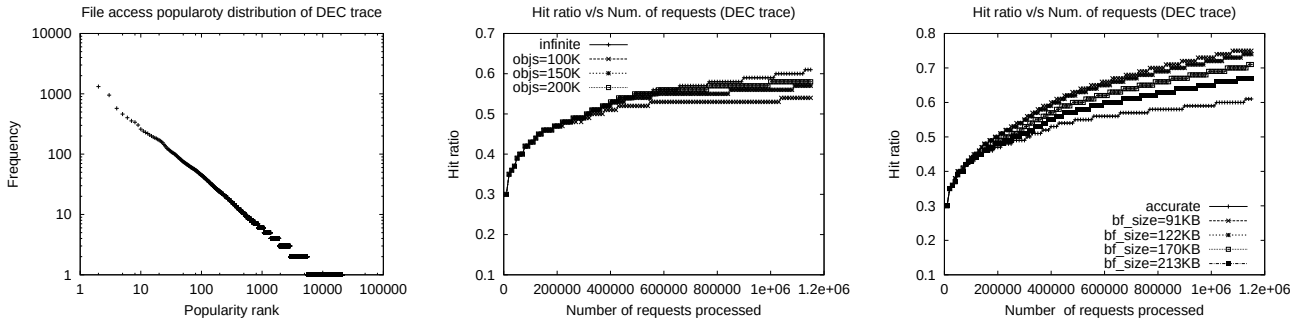
Figure 1: A typical content distribution network

The rest of this paper is structured as follows. Section 2 formulates the problem studied in this paper and provides the necessary background. Techniques for reducing memory requirements and the approach used in this paper are presented in Sections 3 and 4. Implementation issues are discussed in Section 5 and experimental results are presented in Section 6. Section 7 presents related work and Section 8 presents our conclusions.

2 Background and Problem Formulation

Consider a content distribution network with N proxies, each of which act as an intermediary between servers and the end-users (see Figure 1). We assume that each end-user sends requests for web content to a proxy in the CDN. Each proxy is assumed to maintain a cache of frequently accessed content; this cache is typically stored on disk. Upon receiving a request, the proxy services the request from the local cache (in the event of a cache hit) or by fetching the requested object from another proxy or the origin server (in the event of a cache miss). Objects fetched upon a cache miss are inserted into the cache (assuming they are cacheable) for servicing future requests. The specific details of (i) how to service a cache miss (i.e., the policy that determines whether to fetch the object from another proxy or the server) and (ii) the meta-data information required at the proxy to make such decisions are CDN-dependent. Similarly, issues such as organization of the CDN into a hierarchy or proxy groups, the degree of cooperation among proxies to service user requests, the policies used to determine a suitable proxy to serve a particular end-user are also CDN-specific. Since the focus of this paper is on the cache maintained at the proxy, the specific policies and mechanisms employed by the CDN are orthogonal to our work.

The basic goal of our work is to make CDN simulations tractable along the memory dimension. Typically, maintaining cache state is the dominant fraction of the memory requirements in a CDN simulation. This is because the simulator essentially simulates an on-disk proxy cache using in-memory data structures. A typical on-disk proxy cache can be several gigabytes in size and may contain several million objects; maintaining this state in memory can be expensive. To overcome this drawback, we focus on the design of a memory-efficient cache abstraction and library that will serve as the building block for implementing large-scale CDN simulators. Observe that, our cache abstraction will need to be sufficiently general to allow a variety of cache operations that may arise in various CDN simulations.



(a) File access popularity distribution

(b) Hit–ratio with fixed number of objects in cache

(c) Hit–ratio with approximate data structure

Figure 2: Basic techniques to reduce memory requirements during simulations

We assume the following abstraction for a proxy cache. Each proxy cache is assumed to be an ordered repository of data items. The ordering of data items is determined by the cache replacement policy. While our prototype cache library uses LRU to order objects in the cache, in general, any user-defined cache replacement policy may be used to order objects in the cache. Typical operation on the cache include inserting a new object, deletion or modification of existing objects and lookup operations. The simplest lookup operation is a boolean function that determines whether or not an object is in the cache; we use an object ID, such as the URL or a hash of the URL, to uniquely identify objects in the cache. More complex lookup operations are possible. For instance, the simulator may require a list of all objects with certain attributes or a list of all objects whose attributes belong to a range. Examples of such lookup operations include computing a list of all objects greater than 1MB in size or all objects that were modified in the last hour, or a list of the 100 most popular objects in the cache. A cache library should be sufficiently general to support complex cache manipulations and complex lookup operations, and must do so efficiently.

For the purpose of this paper, we assume that attributes of a cached object are either integers or floating point values. Most common attributes such as the object size, the last modified time, and access frequency fall into this category. Our current implementation does not support character string attributes. For instance, this does not allow the object URL to be stored as part of the object-specific state (instead, a MD5 hash of the URL, which yields an integer, may be used to uniquely identify the object).

Assuming the above model for proxy caches, we address the following problem in this paper: How can we reduce the memory footprint of a simulated proxy cache while supporting a flexible range of cache operations?

3 Techniques to Reduce Memory Requirements of CDN Simulations

Several simple techniques can be used to reduce the memory requirements of CDN simulations. We outline these techniques briefly with their advantages and disadvantages.

- *Compression:* A possible technique to reduce the memory footprint of the simulated caches is to use compression. In this approach, the state maintained for each cached object is compressed to

reduce memory usage. The unique identifier for each object (usually the Object ID) is stored in an uncompressed form to enable fast cache lookup operations. The rest of the object-specific state, such as the object size, the last modified time, the object popularity, etc., is compressed (and decompressed only when this state needs to examined or modified by the simulator). The approach is simple to implement, since libraries for compression algorithms such as *zlib* are widely available [1]. There are two potential limitation to this approach. First, the compression algorithm may not be very effective on small objects—the memory required to store object specific state is a few tens of bytes, and thus, small. Compression algorithms are less effective on small objects since they need to store meta-data information with each compressed object so enable decompression; this meta-data information may itself be several bytes in size, reducing the overall compression ratio for small objects. Our preliminary experiments with *zlib* have shown that compressing small objects (upto 90 bytes, each representing a cached element) actually results in an increase in size (since the meta–data overhead is larger than the reduction due to compression). One possible approach to address the problem is to compress a group of objects together; however, the entire group needs to be decompressed to access any element in the group. Another approach is to develop a specialized compression algorithm that is effective on small objects. A second limitation is, the use of compression and decompression makes cache operations compute-intensive. Since CDN simulations are already highly compute-intensive, additional computational overheads for each request can exacerbate the problem.

- *Distributed Simulations:* Another technique to scale CDN simulations is to distribute the simulation across multiple machines—each machine simulates a portion of the CDN and these machines interact with one another to simulate the entire system. Distributed simulations can undoubtedly address the memory and compute bottlenecks of CDN simulations. The limitation though is that inter-process communications can substantially slow down the simulation. In a single process simulation, various simulated entities communicate via shared data structures that are stored in memory. In contrast, distributed simulations require simulated entities to communicate via message passing over a network, which is several orders of magnitude slower than shared memory communication. The greater the amount of communication between simulated entities, the greater the slowdown. In general, distributed simulations are inevitable for simulating very large CDNs, since these are beyond the capabilities of single machines. Consequently, our techniques to make CDN simulations memory-efficient are complementary to this approach—our techniques will enable each machine to simulate a larger number of proxies, and thereby reduce the communication overhead of distributed simulations.

- *Store cache state only for popular objects:* Studies have shown that object popularities of web pages tend to be Zipf-ian [4]. This implies that a large fraction of the requests go to a small fraction of the popular objects. We illustrate this property for a publicly available Digital web proxy trace in Figure 2(a)—the object popularity is indeed heavy tailed with the most popular objects receiving a large fraction of the requests. Based on this observation, a CDN simulator could maintain state for only a certain fraction of the popular objects. Essentially a large on-disk cache is emulated by a smaller fixed size cache; such a fixed-size cache size can substantially reduce the memory requirements of

CDN simulations. Observe that, since the cache continues to store the most popular objects, accesses to these objects behave exactly as they would have with a larger on-disk cache. The only drawback is the handling of requests to unpopular objects. By maintaining state for only popular objects, accesses to less frequently accessed objects get reported as cache misses (whereas they might have been hits in the on-disk cache that stores state for a larger number of objects). These false negatives result in errors in the simulations. Thus, the approach will yield good results only if this error is small in practice. We illustrate this approach for the DEC proxy trace depicted in Figure 2(a). This single-proxy trace contains about 1.2 million requests spanning a 28 hour duration and 449,203 unique objects are requested over the duration of the trace. Figure 2(b) plots a time-series of the cache hit rates when the entire on-disk cache is simulated in memory (i.e., essentially an infinite size cache). The figure also plots the observed hit rates when state for only the 100,000, 150,000 and 200,000 most popular objects are maintained in the cache. We observe an error of 10% for a cache size of 100,000. The error falls to 5% when the cache size is increased to 200,000 objects. The larger the simulated cache size, the smaller the number of observed false negatives. For this trace and a simulated cache size of 200,000 objects, a factor of two reduction in memory requirements ($449203/200000$) is achieved with a false negative error rate of 5%.

- *Store approximate cache state:* An alternate approach for reducing the memory requirements of simulated proxy caches is to use approximations to store cache state. The use of approximations allows a tradeoff between memory space and accuracy — the greater the approximation, the smaller are the memory requirements and the larger the possible error. One possible approximation technique is to quantize object attributes such as object size and the last modified time by a quantization factor Q. The quantized value requires fewer bits of storage and original value of the attribute can be recovered by computing a product of the quantized value and the quantization factor. The limitation of the approach is that all values of an attribute in the range $i \cdot Q$ to $(i+1) \cdot Q$ map to same quantized value and the simulator loses its ability to distinguish between specific values within this range. To illustrate, if quantization is used to store the last modified times of each cached object, then successive modification times of a frequently changing object may get mapped onto the same quantized value, causing the simulator to assume that the object has not been modified. This may result in serving stale data to a user request—a scenario that will not occur if the simulator maintains actual modification times instead of their quantized values.

  Another possible technique to approximate the cache state is to use approximate data structures. For instance, a simulator could use a *bloom filter* to determine if an object is present in the cache. The bloom filter is a boolean lookup function that can determine if an object is present in a set with a high probability [3, 12]. The probabilistic lookup can result in false positives; the number of false positives depends on the exact parameters chosen for the bloom filter. Figure 2(c) illustrates the behavior for the Digital proxy trace. With a bloom filter of size 213KB (approximately $1,750,000$ bits) to hold the 449,203 objects in the trace, a false positive rate of 9.8% is incurred. With a 170KB (approximately $1,400,000$ bits) bloom filter, the error rate increases to 16.39%. False positives cause the simulator to

assume that the object is cached locally, causing it to overestimate the cache hit rate (and erroneously treat these requests as cache hits). An advantage of bloom filters is that their fixed size results in a large reduction in memory requirements and the probability of false positives can be carefully controlled by proper choice of bloom filter parameters. The limitation though is that they can only be used as lookup functions (to determine whether an object is present in the cache) and can not be directly used to store other object attributes such as the size and modification times.

4 Our Approach

The last two techniques mentioned in the previous section merit additional attention. Storing the state of only popular cached objects allows the simulator to handle a large fraction of requests correctly but can yield false negatives for less frequently accessed objects. Storing cache state approximately, on the other hand, yields errors for all cached objects due to the approximate nature of the information stored in the simulated cache. Whereas both techniques have advantages and disadvantages, using a combination of the two can overcome most limitations while retaining their advantages.

Consequently, the approach employed in this paper is to store cache state for a certain fraction of popular objects accurately and to store the state of the remaining (less frequently accessed) objects using approximations. Figure 3 illustrates our approach. Observe that, since popular objects receive most of the requests and the state for these objects is maintained accurately, these requests are handled exactly as they would be in an infinite cache simulation. Further, since state for the remaining objects is also maintained, albeit approximately, the potential for false negatives is eliminated. While errors due to the approximate nature of the state may still occur, these are limited to the small fraction of requests that access these (less popular) objects. Finally, since less popular objects constitute a large fraction of the total objects (and consequently, consume much of the space), maintaining their state using approximate data structure helps reduce the overall memory requirements.

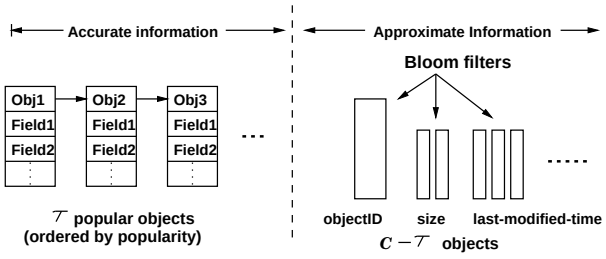


Figure 3: Data structures used to store cached objects of a proxy in the simulation

In the rest of this section, we present the details of our approach and examine various design issues.

4.1 Use of Bloom Filters to Store Approximate State

Consider a simulated proxy cache that is designed to hold up to C distinct objects. Let us assume that state for τ popular objects is maintained accurately and state information for the remaining $C - \tau$ objects is stored

approximately. We assume that the τ popular objects are ordered in the cache by the cache replacement policy (see Figure 3). Any replacement policy suffices for our purpose; for simplicity, we assume LRU in our prototype implementation (our prototype allows a simulator designer to implement other policies as well). The remaining $C - \tau$ objects are not ordered and their state is maintained approximately using *bloom filters* [3, 12].

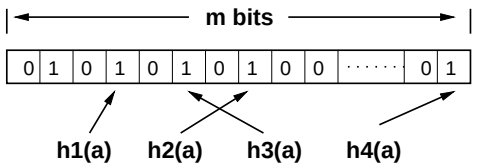


Figure 4: A Bloom filter with 4 hash functions and *m*–bit vector

A bloom filter is essentially a lookup function implemented using hash tables, and can be used to insert, delete and lookup elements. As shown in Figure 4, a bloom filter consists of a m-bit vector and a set of hash functions $H = \{h_1, h_2, h_3, \ldots, h_k\}$ ($k = 4$ in the figure). Initially, all bits of the vector are set to zero. To insert an element a, the bits corresponding to positions $h_1(a), h_2(a), \ldots, h_k(a)$ are set to 1. A deletion requires these bit positions to be reset to zero. A lookup involves examining these bit positions; an element is said to be present if these bit positions are all set. Observe that, a false positive can result if the bit positions for an object are set by other objects that hash to these positions. A false negative can result if two objects a and b have hash functions $h_i(a)$ and $h_j(b)$ that map onto the same bit position and one of these objects is deleted (causing that bit position to be reset). The probability of false positives (errors) assuming no deletions in the bit vector or with deletions permitted on a vector with integer counters, depends on the size of the vector m and the number of hash functions used k. If the number of elements stored in the vector are n, the error probability 'p_e' is given by,

$$p_e = \left(1 - \left(1 - \frac{1}{m}\right)^{kn}\right)^k \approx \left(1 - e^{\frac{-kn}{m}}\right)^k \tag{1}$$

Thus, given $n = C - \tau$, proper choice of the size of the bit vector m and the number of hash functions k can yield a sufficiently small error probability. For Example: choosing $k = 4$ and $\frac{m}{n} = 6$, results in $p_e = 5.61\%$. For our approach, the above error probability is an over–estimate of the overall error of the simulator, as our technique uses the accurate state and if required the approximate state to service requests.

We use a set of bloom filters to store various attributes of the $C - \tau$ less frequently objects; each attribute requires one or more bloom filters. The object ID for each cached object is stored in a single bloom filter (and this filter can be used to determine whether an object is present in the cache). The use of bloom filters to store other object attributes such as object size is discussed next.

4.2 Handling Range Attributes

A bloom filter is essentially a lookup function that uses a unique identifier to determine if the object is present in the set. Thus, bloom filters can not be directly employed to store other attributes such as object size or modification times.

We address this issue by using a combination of quantization and bloom filters. Consider an attribute that can take values between (t_{min}, t_{max}). We then define a set of ranges for the attribute: $(t_{min}, t_1]$, $(t_1, t_2]$, $(t_2, t_3]$, ..., $(t_k, t + max)$ and associate a bloom filter with each range. To insert an element into the cache, we map the attribute to one of the these ranges and insert the object ID into the bloom filter associated with the range. To illustrate, assume that the object size is partitioned into the ranges $(0, 10KB]$, $(10KB, 100KB]$, $(100KB, 500KB]$ and $(500KB, \infty]$. In such a scenario, an object with size 50KB gets mapped to the second range and its object ID is then inserted into the corresponding bloom filter. To lookup the object size at a later time, the simulator needs to determine which of the four bloom filters contains the object ID. The process of mapping the object attribute to a range is essentially quantization and results in an approximation—the simulator can only determine the range containing the attribute. Thus, in the above example, information about the exact size is lost and the simulator can only determine that the object has a size that lies between 10KB and 100KB. Observe that false positive can occur if multiple bloom filters report the presence of an object (for the same reason false positives occur in a single bloom filter).

Several design issues arise when storing range attributes in such a data structure. First, the simulator designer needs to determine the minimum and the maximum value of each attribute. In case of trace-driven simulations, these values could be determined by sampling a portion of the trace. Second, the number of ranges and the size of the range needs to be determined for each attribute. A larger number of ranges permits higher accuracy but also imposes greater lookup overhead and potential increase in the number of false positives. Further, the range size can be determined such that each range has a fixed size or each range has a fixed number of objects. Whereas fixed size ranges are easy to implement, they may result in a non-uniform distribution of objects into ranges (recall that the larger the number of objects in a bloom filter, the greater the probability of false positives). Determining range sizes such that objects are uniformly distributed across ranges is harder and requires a distribution of the object attribute values. The advantage though is that the technique can result in fewer false positives. We experimentally examine the tradeoff of these two choices in Section 6. For trace-driven simulators, our prototype cache library also provides a set of simple tools to analyze the traces and determine various parameters of each attribute (e.g., (t_{min}, t_{max}), the distribution of the values in this range).

4.3 Memory usage savings

To analyze the potential memory benefits of our approach, let us assume that each cached data item has j attributes and let S_i denote the memory requirements (in bytes) of the i^{th} attribute. Then, the memory required to store the state of a data item is $S = \sum_{i=1}^{j} S_i$. Assuming we need to store a total of C items at a proxy, the total memory requirements to simulate each proxy cache accurately is $M_{acc} = C \times S$. In our approach, the state for τ popular objects is stored accurately and the state of remaining objects is stored in a set of bloom filters. Let B denote the memory requirements to store each attribute in one or more bloom filters. Then, the memory requirements of our approach is $M_{approx} = (\tau \times C) + (B \times j)$. Hence, the memory savings per proxy cache is

$$M_{saved} = (C \times S) - [(\tau \times S) + (B \times j)] \quad (2)$$

From the above equation, we can see that if τ is small relative to the total number of cached elements C and if B is not very large we can get good memory savings. Hence, the number of popular objects τ with accurate state information and the size of each bloom filter B should be chosen carefully to realize these savings in practice.

4.4 Handling Reverse Lookups

One limitation of using bloom filters is that reverse lookups are not supported. That is, given an object ID, it is possible to determine if the object is present in the cache and determine its attributes. However, it is not possible to generate a list of objects that satisfy a certain criteria (e.g., a list of all cached objects that are larger than 4KB in size). Answering such queries involves a reverse lookup, since it requires a list of all object IDs matching the criteria to be computed. This limitation arises because the bloom filter stores a *hash* of the object ID; the object ID is itself not stored, and consequently, can not be reconstructed. Note that this drawback is limited only to the objects stored in the bloom filter; reverse lookups can still be supported for the τ popular objects, since the object ID is stored in the cache for these objects.

The approach outlined in the previous section needs to be modified to support cache operations with reverse lookups on *all* cached objects. The modification is simple but it requires additional memory. Like before, we assume that state for τ popular objects is stored accurately. To support reverse lookups on the remaining objects, we store their objects IDs in a binary search tree (instead of storing the hash of the object ID in a bloom filter). All other attributes are stored in bloom filters like before. Maintaining a list of object IDs allows reverse lookup operations, while maintaining them in a binary search tree enables fast lookups. The overall memory savings of this enhanced approach depends on the total number of attributes j that is maintained for each cached object—since memory savings can still be realized for the remaining attributes, the larger the value of j, the greater these savings.

5 A Cache Library for CDN Simulations

Based on the approach outlined in Section 4, we have implemented a library that can be used to simulate proxy caches in CDN simulations. The library provides an important building block for implementing memory-efficient CDN simulators.

The library can be configured to instantiate caches where all information is stored accurately as well as caches where only information for popular objects is maintained accurately (and state for remaining objects is stored using bloom filters). The library requires the simulator designer to specify various attributes that need to be stored with each object; an arbitrary number of attributes can be associated with each cached object. Currently only integer and floating point attributes are supported by the library. The library allows various bloom filter parameters to be configured at initialization time. These include the memory associated with each filter, the number and type of hash functions that are used, the number and size of ranges for range attributes. For trace-drive simulation, the library provides simple tools to examine trace workloads and determine some of the above parameters automatically.

Each proxy cache instantiated using the library supports a flexible set of cache operations. Be default, the LRU cache replacement policy is used to manage the cache; user-defined replacement policies are also supported. Typical operations on the cache include insertions, modifications and deletions of data items. Simple boolean lookups as well as lookups on arbitrary attributes are supported. The latter operations are supported using *iterators*–the iterator returns a list of all objects, one at a time, that match certain criteria.

5.1 Basic functionality of a cache

To understand the functionality a cache librabry should provide, we look at the functionality of a real proxy cache. A cache stores objects and information about objects (e.g: expiry time of objects) and the commonly supported operations are: *insertObjectInCache(), deleteObjectFromCache(), updateObjectInCache() and lookupObjectInCache()*. The objects stored in the cache can also be queried to determine the state of the cache, examples of which are: (i) Retrieve list of all objects in the cache, (ii) What is the number of objects with size greater than 4KBytes ?, (iii) Retrieve list of all objects with expiry times in next 10 minutes. Thus, the important categories on which the operations of a cache are: (i) functions to manipulate the state of the cache and (ii) querying the cache based on fields of objects.

5.2 Design of the user–library

The library should be flexible so that a user can configure it according to requirements of the proxy–based solution. The library should provide an interface to initialize it with the required cache parameters and also an interface to store and perform operations on the objects stored in the cache. We intend to support storing objects in the accurate cache and in approximate data structures(bloom filters). A user can set the size of a cache in terms of number of objects in the accurate cache with a call to the function int sizeOfAccurateCache(size). If a user desires to store all objects accurately, the user can use the same call with a special flag, e.g: sizeofAccurateCache(ALL_ACCURATE). The flag *ALL_ACCURATE* can be viewed as a pre–definied constant to denote infinite number of objects in the accurate cache. If all objects are not to be stored accurately, the number of objects to be stored accurately can be passed as a parameter to the function.

Each object stored in the cache has a number of fields associated with it and often a primary field (e.g: the object Id in most cases). To initialize the library with the fields, their names, their types, the range of values they can hold and the number of ranges they should be split into if they are to be sotred approximately, the following functions can be used:

```
void setNumberOfFields(int);
int getNumberOfFields();
int addPrimaryField(String fieldname, int fieldtype);
int addField(String fieldname, int fieldtype, maxvalue, minvalue, NumRanges);
int addField(String fieldname, int fieldtype);
int addField(String fieldname, int fieldtype, maxvalue, minvalue);
int addField(String fieldname, int fieldtype, NumRanges);
```

Fields that do not specify their range or the number of ranges they shoud be divided into during approximation will have default values. The library uses index values to store and refer to fields of an object. When

fields of an object are initialized, the library builds the mapping between a fieldname and its index. Index values can be incrementally generated as fields are added to the cache structure.

Once the fields for objects to be stored in the cache are initialized, the user can use the following functions to manipulate the state of the cache:

int insertInCache(Object)

int insertInCache(primaryfieldvalue)

int insertInCache(primaryfieldvalue, fieldname, value)

int insertInCache(primaryfieldvalue, fieldindex, value)

int deleteFromCache(primaryfieldvalue)

int deleteFromCacheField(primaryfieldvalue, fieldname)

int deleteFromCacheField(primaryfieldvalue, fieldindex)

int updateCache(primaryfield, fieldname, value)

int updateCache(primaryfield, fieldindex, value)

Note: The **Object** structure used above needs to be a special strucute that supports iterating through the fields for their values, so that can be used by the cache storage. An object can be looked up if it exists in the cache by using the function Object lookupCache(primaryfieldvalue). The library assumes that all objects are hashed or indexed using a single primary field that uniquely identifies an object.

The following functions can be used to set the size of bloom filters for the fields of objects in the cache, setBloomFilterSize(size), setBloomFilterSize(fieldname, size) . The library can use a default size for the bloom filters or use set of heuristics to determine the size of a bloomfilter for each field based. For the library to calculate the size of bloom filters automatically it needs extra information about the objects that will be stored in the cache. In particular, the approximate number of unique objects in the trace, available memory for simulation. The library can provide the following interface for the user to initialize these values: setApproxUniqueObjects(value), setAvailableMemory(memsize). The library can use either one or both values as hints to set the size of the bloom filter for each field.

Other cache related helper functions are setReplacementPolicy(policy), int getReplacementPolicy(), …. The library can pre–define a set of replacement policies that are implemented and the user specifies the one he is wishes to use for simulation.

Functions that can be used to query the cache are as follows:

int createIterator(fieldname, value)

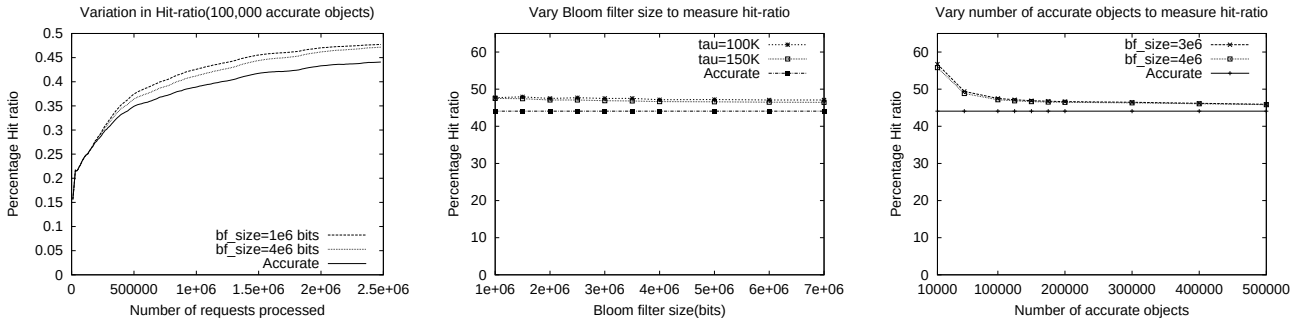
int createIterator(fieldindex, value)

Object getNextObject()

The *createiterator* sets–up the library to start querying the cache on a particular field and the *getNextObject* retrieves the next object in the cache with value of as specified.

The functions ObjectList getObjects(fieldname, value) and ObjectList getObjects(primaryfield) can be used to retreive list of all the objects with a particular value of a field. Using the the primary field as an argument will retreive the entire list of objects stored in the cache. This can be helpful in answering more involved query, e.g: how many objects have size greater than 4Kb but less than 10KB.

Implementing a user–library that uses this design and implements the described interface is part of on–going work.



(a) Time series of hit rates (b) Impact of bloom filter size (c) Impact of the number of objects in the accurate cache

Figure 5: Effect of varying bloom filter parameters

Name of trace	Duration	Number of requests	Number of unique requests
DEC	1 day 4 hrs	1,141,412	449,203
NLANR	3 days	3,588,996	1,146,417
BOEING	1 day	4,421,526	1,498,327
UCBerkeley	4 days 16 hrs	2,472,954	1,382,813

Table 1: Workload characteristics for simulations

6 Experimental evaluation

In this section, we study the effectiveness of our approach using two simulators that we construct using our cache library. The first simulator simulates a cache where the state of all objects is maintained accurately. The second simulator only maintains state for a small subset of objects accurately and stores the state of remaining objects in bloom filters. Using these simulators, we evaluate the impact of approximations on the accuracy of the results as well as quantify the memory savings and scalability of our approach.

To ensure a fair comparison, both simulators are identical in every other respect other than the cache implementation. Each simulator simulates a number of proxies that service web requests. Given our focus on caching overheads, for simplicity, we assume that proxies don't cooperate with one another while servicing requests (thus, all cache misses are serviced by fetching the object from the server). We assume infinite size caches for the accurate simulator and assume that each cache is managed using the LRU cache replacement policy (due to the infinite cache size, no cache replacement occur. The only impact of the LRU policy is to impose an ordering of objects in the cache—an aspect that determines which objects are stored using bloom filters and which are not). The replacement policy is also responsible for moving objects from bloom filters to the accurate cache data structure and vice versa.

We use a set of real-world traces to generate the workloads in our study. The characteristics of these traces are listed in Table 1. Each trace has over a million requests and span at least a day. We assume that each proxy in the CDN processes this trace independently of other proxies.

Assuming the above setup, we now describe our experimental results.

6.1 Effect of bloom filter parameters

As explained in section 4, we use bloom filters to store approximate state. Several parameters must be chosen to properly configure bloom filters: (i) the number of bits for each bloom filter, (ii) the number of hash functions used by a bloom filter, (iii) the number of ranges a field is divided into and (iv) the number of objects stored in the accurate cache. We study the effect of these parameters on the cache hit-ratio seen in the approximate simulation and compare it with the accurate simulation. The experiments presented are using the UCBerkeley trace, for results using other traces refer to [19].

We assume that each cached object has 10 attributes. We assume that each bloom filter has 4 has functions and 4 ranges for each field. In our first experiment, we systematically vary the size of the bloom filter and compare the cache hit rate to that in the accurate simulation. Figure 5(a) shows the hit rate observed at various instants in the simulation for the two scenarios. The hit rate for the 1Mb bloom filters is 47.72%, while that for 4Mb filters is 47.16%. The accurate simulation shows a hit rate of 44.08%, yielding errors of 8.25% and 6.98%, respectively. Figure 5(b) shows the final cache hit rates when each simulation terminates. The figure depicts the observed hit rates for different bloom filter sizes and compares it to the hit rate from the accurate simulation. Although larger bloom filters yield lower errors, the reduction in error rate is very small, indicating that beyond a certain threshold, increasing the bloom filter size yields diminishing returns. The figure also shows that the choice of the τ—the number of objects for which state is maintained accurately—has a greater impact on accuracy that the size of the bloom filter.

To further understand this effect, we fix the size of the bloom filters and vary τ—the number of objects stored in the accurate cache. As seen in Figure 5(c), initially, the error in the hit rate decreases sharply with increasing τ values. Beyond a certain threshold, an increase in τ yields only marginal reduction in the error.

Figure 6 shows the impact of varying the number of hash functions used in the bloom filter. As shown, the number of false positives decreases with the increase in the number of hash function, consistent with Equation 1. Increasing the number of hash function also increases the probability of false negatives. We see that beyond a certain threshold, the approximate simulator under-estimates the hit rate slightly, indicating the presence of false negatives. The number of hash functions should be chosen properly to balance these tradeoffs.

6.2 Effect of number of ranges

As described in section 4, we propose to split the values a field can hold into multiple ranges. Each range is associated with a bloom filter and objects with values in a range are stored in the corresponding bloom filter. When a value for an object is to be retrieved, a lookup in each range has to be performed to check if the object exists in the particular range. The use of ranges instead of using accurate values can result in two side-effects: (i) multiple bloom filters can return *true* for a queried object resulting in an undecidable range for an attribute and (ii) a modified attribute may map to the same range and be incorrectly treated as unchanged. To study these two side-effects we used the *last–modified–time* field of objects from the UCBerkeley trace. Figure 7(a) shows the results of measuring the cases when last–modified–times were wrongly reported as unchanged when they actually did change. From the figure, we see that as the number of ranges for an

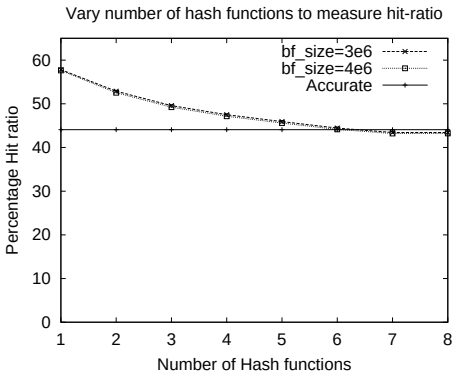
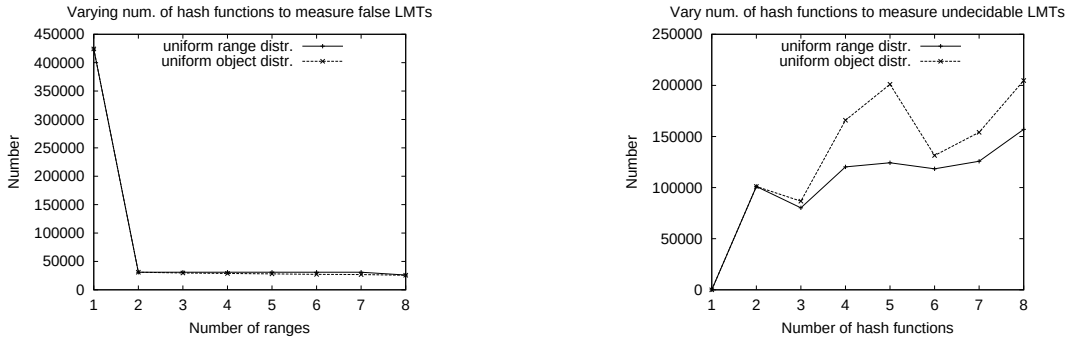


Figure 6: Effect of the number of hash functions



(a) Cases when LMT is wrongly reported as unchanged (b) Cases when range of LMT is undecidable

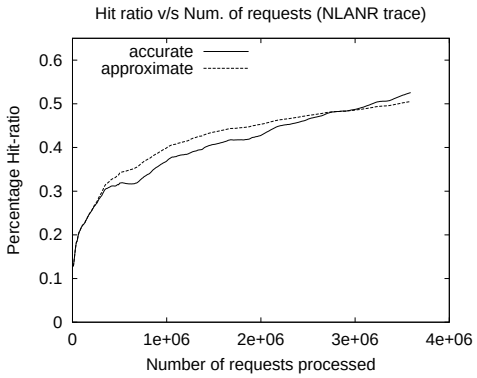
Figure 7: Effect of varying number of ranges for the *last–modified–time*(LMT) field

attribute increases, the approximation error decreases, yielding fewer undetected values. Figure 7(b) shows the results for the number of cases when multiple ranges reported the queried object being present. The number of undecidable cases increases with increase in number of ranges. This is because larger the number of ranges, smaller is each individual bloom filter and greater the chances of false positives. The number of ranges can simultaneously affect the accuracy of the stored value and the number of false positives. Thus, this parameter must be chosen carefully to balance the tradeoff.

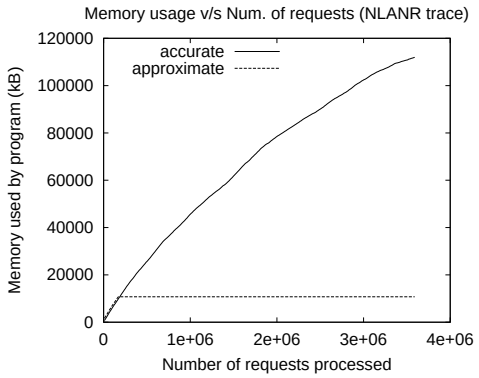
6.3 Memory savings and Scalability

In this section, we present experiments demonstrating the advantage of using our approach due to potential memory savings.

First, we demonstrate the reduced memory usage of the approximate simulator. For this purpose we use the NLANR proxy trace. The experiment simulates a singe proxy and measures the amount of memory used by the simulator to implement the cache. Figure 8 shows results for the NLANR trace with the approximate simulator storing $100,000$ objects in the accurate cache and bloom filters of size 1Mb for 10 fields. As seen in Figure 8(b) the amount of memory used by the accurate simulator increases almost linearly with increase



(a) Variation of hit–ratio with increase in number of requests processed



(b) Memory consumption in a single proxy simulation with increase in number of requests processed

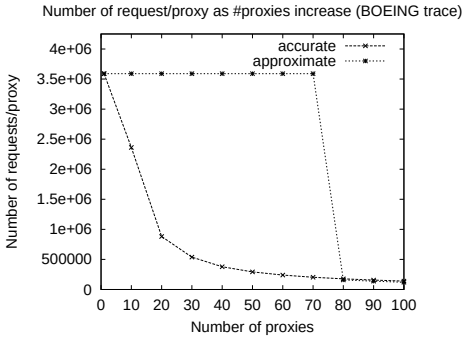
Figure 8: Scalability of our approach in terms of memory consumed to simulate a single proxy

in the number of requests processed. The approximate simulation on the other hand does not allocate memory incrementally with increase in number of requests processed. The amount of memory required in an approximate solution, is the memory to store all bloom filters and the memory required to store the $100,000$ objects in the accurate cache. As a result, we find that in the approximate solution the memory used increments for a short duration (till $100,000$ objects in accurate cache are not allocated) and then remains constant. In this case the memory used by the accurate simulator is 109.34MB and that by the approximate simulator is 10.49MB, an 10.42–fold savings in memory usage (see Figure 8(b)). The reported hit–ratios are 52.57% and 50.54% for the accurate and approximate simulation respectively, resulting in a prediction error of 3.86% (see Figure 8(a)). Referring to equation 2 in section 4.3, we can predict the potential memory savings due to the approximate approach. For example, for this experiment we have size of accurate cache structure $S = 80$ bytes, number of unique objects $n = 1,146,417$, size of accurate cache $k = 100,000$, number of fields $j = 10$ and bloom filter size for each field $B = 1,000,000$ bits. Thus,

$$\begin{aligned} M_{accurate} &= (80 \times 1,146,417) \\ &= 87.46MB \\ M_{approximate} &= [(80 \times 100,000) + (12500 \times 10)] \\ &= 7.93MB \end{aligned}$$

As a result, with the chosen parameters we can expect approximately 11–fold savings in memory using our approach, which is fairly close to one measured in the experiment (see Figure 8(a)).

In Figure 9(a) we measure the number of requests each proxy processed with increase in the number of concurrent proxies in a simulation. The simulation used the NLANR trace as input to each proxy and an upper limit on the memory usage for the simulated cache was fixed at 800MB. The approximate approach stored $100,000$ objects accurately, simulated 8 fields per object with 1Mb bloom filters for each field and



(a) Scalability in terms of number of proxies that can process the complete trace when simulated concurrently

Number of Requests	Unique objects	Accurate simulation		Approximate simulation		
		# proxies	Normalized runtime(secs)	# proxies	% error	Normalized runtime(secs)
500,000	223,748	27	14	200	4.77	33
1,000,000	393,140	15	34	118	3.12	66
1,500,000	558,177	11	54	86	3.18	100
2,000,000	725,662	8	76	62	2.38	122
2,500,000	883,373	7	100	50	1.98	165
3,000,000	1,040,737	5	124	43	1.44	200
3,500,000	1,199,508	5	152	36	1.3	231
4,000,000	1,357,768	4	182	32	1.32	265

(b) Number of proxies that can be concurrently simulated given the number of request per trace

Figure 9: Number of proxies that can be simulated concurrently with different number of requests

another $1000,000$–bit bloom filter for the object identifiers. With a single proxy, both the accurate and approximate simulation are able to process all the $3,588,996$ requests in the trace. In an accurate simulation with 10 proxies, each proxy is able to process on an average of $2,362,009$ requests and cannot process the entire trace. The approximate simulation, due to lower memory requirements for each proxy, can simulate all requests in the trace upto 70 concurrent proxies in a simulation. Whereas in the accurate simulation, with as many as 10 concurrent proxies all requests cannot be processed by each proxy and the number decreases further with increase in number of proxies. In the approximate simulation, with more than 70 proxies, the average number of requests processed be each proxy decreases sharply. This is due to the fact that, the approximate approach does not consume memory incrementally but allocates all the bloom filters at the start of the simulation and allocates memory incrementally only for the accurate state. As a result, we find that with more than 70 concurrent proxies in a simulation, the memory required by the bloom filters of all these proxies and the accurately stored objects is more than 800MB and hence the simulator cannot process all requests at each proxy. This is an interesting result, as we find that even though the approximate approach uses less memory over the entire run of the simulation, our bloom filter approach can hit a memory bound as the number of proxies increase. The result verifies that, due to reduced memory requirement of the approximate approach the simulator can scale to more proxies. In this case we see a minimum of 7–fold increase in the number of proxies that can simulate the entire trace of requests.

The next experiment demonstrates the scalability of our approach based on the number of requests processed by each proxy. For this purpose, we used the BOEING trace and varied the number of requests to be processed by each proxy from $500,000$ to $4000,000$. Additionally, the memory used by all the proxies to simulate the cache was fixed to 500MB. The experiment was performed on a DELL PowerEdge server with a Pentium III processor running at 966MHz and 1GB RAM. Figure 9 reports our results. As we increase the number of requests to be processed by each proxy, the number of proxies that can be concurrently simulated decreases. For the accurate simulation, with $500,000$, requests 27 proxies can be concurrently simulated

and with $4,000,000$ requests 4 proxies. The number of unique objects stored as part of the cache state increases with the increase in the number of requests processed. As a result with more number of requests processed, the available memory is consumer at a faster rate. Whereas, for the same cases in the approximate simulation, 200 proxies and 32 proxies respectively can be concurrently simulated. The average ratio of the number of proxies that can be simulated using the approximate approach and at accurate approach is 7.72. The experiment also reports the normalized runtime per proxy required to complete the simulation. As the bloom filter uses more than one hash function to perform cache operations, it potentially takes more time to execute. On an average the approximate approach takes 1.75 times the runtime for accurate simulation. The time of execution for the approximate approach depends on the number of hash functions used to service each request. In our example, we used 5 hash functions on a single field and with an increase in number of hash functions the time to complete the simulation also increases. Such a condition does not arise in the accurate simulation as a single lookup provides information of all fields. By storing popular objects accurately, our approach tries to mitigate the problem. The percentage error of the approximate solution with the chosen parameters is also reported to be always less than 5%. Given a fixed set of requests to be processed, our approach can always simulate more number of proxies concurrently and in our experiment an average of 7 times more at very good accuracy. Note that the bloom filter parameters can be varied to affect accuracy, runtime of the simulation and the number of concurrent proxies that can simulated.

7 Related Work

The design of policies and mechanisms for content distribution networks is an active area of research. Recent and current research efforts have investigated web caching algorithms [18, 5, 27], cache consistency mechanisms [6, 11, 16, 30, 32, 31, 33], cooperative proxies [7, 9, 22, 24, 25, 28], and the characterization of web traffic [2, 8, 10]. Our work seeks to provide scalable simulation techniques to investigate the behavior of such mechanisms and policies in large scale settings.

The notion of bloom filters was first proposed in [3] to reduce amount of memory required to store hash–coded information; the approach traded space for quality. More recently, bloom filters have been used to compactly store meta-data information in cooperating proxy caches [12]. Inspired by this technique, we propose to use bloom filters to store cached state approximately and thereby reduce the memory requirements of CDN simulations. Compressed bloom filters have been studied in [21], and improve performance of bloom filters by using compression techniques that reduce its size during message passing.

Techniques for scalable simulations have been investigated in the simulation research community. Techniques for distributed and parallel discrete event simulations have been studied in [23, 26]. In the context of networks, traditional packet-level discrete-event simulations do not scale to larger networks, and several simulation techniques based on approximations have been studied. The approaches can be broadly classified into two categories: (i) techniques that abstract the level of traffic [14, 17, 20] (i.e: treating a set of packets as a single unit or abstracting a flow of packets using a fluid model) and (ii) techniques that reduce the number of packet arrival events either by simulating the system at fixed time intervals or by using arrival distributions [15, 29]. While these techniques are not directly applicable to reduce the memory requirements

of CDN simulations, they may help reduce the computational requirements (e.g., by treating request arrivals at a proxy as a flow or by using arrival distributions). The design of techniques to reduce the computational requirements of CDN simulations is the subject of ongoing work.

8 Conclusions and Future work

Since CDN simulations are known to be highly memory-intensive, in this paper, we argued the need for reducing the memory requirements of such simulations. We proposed a novel memory-efficient data structure that stores cache state for a small subset of popular objects accurately and uses approximations for storing the state for the remaining objects. Our design was based on the following observation: popular objects receive a large fraction of the requests, while less frequently accessed objects consume much of the memory space. Maintaining accurate state for objects accessed by a large fraction of the requests and approximate state for the objects consuming much of the space yields large memory savings and reduces errors. We used bloom filters to store approximate state and showed that careful choice of parameters can substantially reduce the probability of errors due to approximations. We implemented our techniques into a user library for constructing proxy caches in CDN simulators. Our experimental results showed up to an order of magnitude reduction in memory requirements of CDN simulations, while incurring a 5-10% error.

Future work: For our approach, the probability of false–positives in bloom filter depends on the number of hash functions used, the size of the bloom filter, and the number of ranges used for fields of objects. We can formulate an optimization problem to minimize the false positives and also the required time for completion, by choosing optimal values of the above parameters. As part of on going work, we are exploring issues related to solving such an optimization problem to predict values of parameters for desired quality of simulator results. We are also interested in studying techniques to reduce the computation requirements of CDN simulations.

Acknowledgments

We would like to thank Yujing Wu, from the ECE department at University of Massachusetts Amherst, for the numerous useful discussions and her feedback on this work.

References

[1] The zlib compression library. http://www.gzip.org/zlib/.

[2] M. Arlitt and C. Williamson. Web Server Workload Characterization: The Search for Invariants. In *Proceedings of the SIGMETRICS Conference on Measurement and Modeling of Computer Systems*, May 1996. http://www.cs.usask.ca/projects/discus/discus_reports.html.

[3] B. Bloom. Space/time trade-offs in hash coding with allowable errors. *Communications of ACM*, 13(7):422–426, July 1970.

[4] L. Breslau, P. Cao, L. Fan, G. Phillips, and S. Shenker. Web Caching and Zipf-like Distributions: Evidence and Implications. In *Proceedings of Infocom'99, New York, NY*, March 1999.

[5] P. Cao and S. Irani. Cost-Aware WWW Proxy Caching Algorithms. In *Proceedings of the USENIX Symposium on Internet Technologies and Systems, Montrey, CA*, December 1997.

[6] P. Cao and C. Liu. Maintaining Strong Cache Consistency in the World-Wide Web. In *Proceedings of the Seventeenth International Conference on Distributed Computing Systems*, May 1997.

[7] A. Chankhunthod, P. B. Danzig, C. Neerdaels, M. F. Schwartz, and K. J. Worrell. A Hierarchical Internet Object Cache. In *Proceedings of 1996 Usenix Technical Conference*, January 1996.

[8] M R. Crovella and A. Bestavros. Self-Similarity in World Wide Web Traffic: Evidence and Possible Causes. *IEEE/ACM Transactions on Networking*, 5(6):835–846, December 1997.

[9] M. Dahlin, R. Wang, T. Anderson, and D. Patterson. Cooperative Caching: Using Remote Client Memory to Improve File System Performance. In *Proceedings of the First Symposium on Operating Systems Design and Implementation*, pages 267–280, November 1994.

[10] B. Duska, D. Marwood, and M. Feeley. The Measured Access Characteristics of World-Wide-Web Client Proxy Caches. In *Proceedings of the USENIX Symposium on Internet Technologies and Systems*, December 1997.

[11] V. Duvvuri, P. Shenoy, and R. Tewari. Adaptive Leases: A Strong Consistency Mechanism for the World Wide Web. In *Proceedings of the IEEE Infocom'00, Tel Aviv, Israel*, March 2000.

[12] L. Fan, P. Cao, J. Almeida, and A Z.Broder. Summary Cache: A Scalable Wide-area Web Cache Sharing Protocol. In *Proceedings ACM SIGCOMM'98, Vancouver, BC*, pages 254 – 265, September 1998.

[13] FreeFlow product details. Akamai, Inc., http://www.akamai.com/service/freeflow.html, 1999.

[14] S. Gadde, J. Chase, and A. Vahdat. Coarse–grained network simulation for wide-area distributed systems. In the Communication Networks and Distributed Systems Modeling and Simulation Conference (CNDS 2002).

[15] Y. Guo, W. Gong, and D. Towsley. Time-Stepped Hybrid Simulation (TSHS) for Large Scale Networks. In *The Proceedings of INFOCOM 2000*, Tel Aviv, Israel, March 2000.

[16] J. Gwertzman and M. Seltzer. World-Wide Web Cache Consistency. In *Proceedings of the 1996 USENIX Technical Conference*, January 1996.

[17] P. Huang, D. Estrin, and J. S. Heidemann. Enabling Large-Scale Simulations: Selective Abstraction Approach to the Study of Multicast Protocols. In *MASCOTS*, pages 241–248, 1998.

[18] B. Krishnamurthy and C. Wills. Proxy Cache Coherency and Replacement—Towards a More Complete Picture. In *Proceedings of the 19th International Conference on Distributed Computing Systems (ICDCS)*, June 1999.

[19] P. Kulkarni, P.Shenoy, and W. Gong. Scalable Techniques for Memory-efficient CDN Simulations. Technical report, Department of Computer Science, University of Massachusetts, Amherst, 2002.

[20] B. Liu, D. R. Figueiredo, Y. Guo, J. F. Kurose, and D. F. Towsley. A Study of Networks Simulation Efficiency: Fluid Simulation vs. Packet-level Simulation. In *The Proceedings of INFOCOM 2001*, pages 1244–1253, 2001.

[21] M. Mitzenmacher. Compressed bloom filters. In *The 20th Annual ACM Symposium on Principles of Distributed Computing*, pages 144–150, 2001.

[22] A. Ninan, P. Kulkarni, P. Shenoy, K. Ramamritham, and R. Tewari. Cooperative Leases: Scalable Consistency Maintenance in Content Distribution Networks. In *Proceedings of the Eleventh World Wide Web Conference*, Honolulu, Hawaii, 2002.

[23] A. L. Poplawski and D. M. Nicol. An Investigation of Out-of-Core Parallel Discrete-Event Simulation. In *Proceedings of the Winter Simulation Conference*, pages 524–530. IEEE Computer Society Press, 1999.

[24] P. Sarkar and J. Hartman. Efficient Cooperative Caching using Hints. In *Proceedings of Operating Systems Design and Implementation Conference*, October 1996.

[25] R. Tewari, M. Dahlin, H M. Vin, and J. Kay. Beyond Hierarchies: Design Considerations for Distributed Caching on the Internet. In *Proceedings of the 19th International Conference on Distributed Computing Systems (ICDCS)*, June 1999.

[26] V. Vee and W. Hsu. Parallel discrete event simulation: A Survey. Technical report, Centre for Advanced Information Systems, Nanyang Technological University, Singapore, August 1999.

[27] S. Williams, M. Abrams, C. R. Standridge, G. Abdulla, and E. A. Fox. Removal Policies in Network Caches for World-Wide Web Documents. In *Procedings of the ACM SIGCOMM '96 Conference*, Stanford University, CA, 1996.

[28] A Wolman, G. Voelker, N. Sharma, N. Cardwell, A. Karlin, and H. Levy. On the Scale and Performance of Cooperative Web Proxy Caching. In *Proceedings of the 17th SOSP, Kiawah Island Resort, SC*, pages 16–31, December 1999.

[29] A. Yan and W. Gong. Time-driven fluid simulation for high-speed networks with flow based routing. In *The Symposium on Applied Telecoms*, Boston, MA, April 1998.

[30] J. Yin, L. Alvisi, M. Dahlin, and C. Lin. Using Leases to Support Server-Driven Consistency in Large-Scale Systems. In *Proceedings of the Eightteenth International Conference on Distributed Computing Systems*, May 1998.

[31] J. Yin, L. Alvisi, M. Dahlin, and C. Lin. Hierarchical Cache Consistency in a WAN. In *Proceedings of the Usenix Symposium on Internet Technologies (USEITS'99), Boulder, CO*, October 1999.

[32] J. Yin, L. Alvisi, M. Dahlin, and C. Lin. Volume Leases for Consistency in Large-Scale Systems. *IEEE Transactions on Knowledge and Data Engineering*, January 1999.

[33] H. Yu, L. Breslau, and S. Shenker. A Scalable Web Cache Consistency Architecture. In *Proceedings of the ACM SIGCOMM'99, Boston, MA*, September 1999.