# Towards Line-Speed and Accurate On-line Popularity Monitoring on NDN Routers

Huichen Dai, Yi Wang, Hao Wu, Jianyuan Lu, Bin Liu
Tsinghua National Laboratory for Information Science and Technology
Dept. of Computer Science and Technology, Tsinghua University

***Abstract*—NDN enables routers to cache received contents for future requests to reduce upstream traffic. To this end, various caching policies are proposed, typically based on some notion of content popularity, e.g., LFU. But these policies simply assume the availability of content popularity information without elaborating how that information is obtained and maintained in routers.**

**Towards line-speed and accurate on-line popularity monitoring on NDN routers, we propose a Bloom filter-based method to continuously capture content popularity with efficient usage of memory. In this method, multiple Bloom filters are employed and each one is responsible for a particular range of popularity. Content objects whose popularities fall into a Bloom filter's range will be inserted into that Bloom filter. Meanwhile, a sliding window monitoring scheme is proposed to implement more frequent and real-time update of the popularities. Moreover, we put forward three optimization schemes to further speed up the monitoring operations. Using a real trace stored in off-chip memory as input and setting the monitoring time window to 30 min, this method achieves a monitoring speed of 20.92 million objects per second (M/s) with multiple threads. This speed is equivalent to 16.74 Gbps throughput assuming the content length is 100 Bytes in average, but only consumes around 32 MB memory. By simulating the environment on the line card using a real-time generated synthetic trace, this method even reaches a speed of 251.07 M/s (equivalent to 200.86 Gbps) because the trace is fetched from high speed on-chip memory, rather than the off-chip DRAMs. Furthermore, both theoretical and experimental analyses elucidate very low relative error of this method. At last, a real trace-driven comparison shows that LFU policy achieves higher hit rate than LRU with much less unnecessary cache replacements.**

***Index Terms*—NDN, Line-Speed, Popularity Monitoring, Bloom Filter.**

## I. Introduction

Named Data Networking (NDN) [1], [2] enables routers with the capability of caching content objects passing by, which is considered as a property different from current Internet. These distributed in-router caches make up the *in-network* cache, which can improve network performance remarkably by increasing cache hit rate and decreasing upstream traffic. Today's routers are equipped with packet buffers too, they are mainly used for packet scheduling and queuing on line cards. In-network caches, however, aim to store content objects for a long period of time, which is useful for satisfying retransmitted requests due to packet loss, repeated duplicate requests and consumer mobility.

While the traffic of content objects traversing through a router are in huge-scale and high-speed, caching all of them is unrealistic. Therefore, which content objects to cache is crucial for cache storage efficiency. Intuitively, routers should cache *popular* contents since they are very likely to be requested again. Therefore, popularity-based policies are widely studied [3]–[12] on the overall cache performance in NDN.

Popularity-based caching policies require popularity information as a prerequisite, however, most of these works [3]–[12] simply assume the availability of content popularity information without elaborating how that information is obtained and maintained in routers. This is actually a daunting task. A straightforward method is simply using a hash table to record the content objects passing by and count their access times, which is impractical because of the large volume of content traffic and the high speed links. One highly-related existing work is capturing popularities of *popular* Web objects on a Web proxy [13]. But this work depends largely on the prior knowledge of the traces, and the speed requirement is not as high as on router line cards. Moreover, routers should measure the popularities of the content objects belonging to all the applications, rather than a specific one. We formally define *popularity* as the number of accesses over a time period. Capturing popularity statistics on a router is a non-trivial task, and we describe this problem as: *line-speed on-line popularity measuring by routers with high accuracy and low memory cost.* We are faced with the following challenges when trying to solve it:

1) *High speed.* Current routers are equipped with high bandwidth (e.g., OC-192, OC-768), and we have to examine each packet passing by to collect popularity statistics. *On-line* popularity monitoring at line-speed is difficult and challenging;
2) *High accuracy.* Other than fast speed, the monitoring should be simultaneously accurate since popularity information is the basis for many caching policies and applications. Large relative error will fundamentally degrades their performance.
3) *Large memory consumption.* NDN uniquely identifies each content object by assigning it a name, which is hierarchical and could be very long. Storing the name of each content object and an associated counter storing its popularity is impractical because of the large amount of content objects. Allowing for the limited memory on router line cards, a memory-efficient and easy-to-implement method is under urgent demand;
4) *Real-time monitoring.* Object popularity varies over time, timely updating the popularity information of the past time window can help popularity-based caching policies to make in time and effective cache decisions.

In this paper, we propose an on-line Bloom filter-based method to capture popularity statistics on routers. Specifically, we utilize $k$ Bloom filters ($BF_1 \sim BF_k$) to keep track of the content names, and each Bloom filter is responsible for a popularity range. Content objects whose popularities fall into a Bloom filter's

This work is supported by 863 project (2013AA013502), NSFC (61373143), Tsinghua University Initiative Scientific Research Program (20121080068). Corresponding Author: Bin Liu (lmyujie@gmail.com).

range will be inserted into that Bloom filter[1]. For example, $BF_1$ represent popularity range $[1, N]$, so it records the content objects whose popularities are in the range $[1, N]$, $BF_2$ records the content objects with popularities in range $[N+1, 2N]$, and $BF_i$ corresponds to popularity range $[(i-1)*N+1, i*N]$, so on and so forth. This design reveals that, within the monitoring time interval $[t_0, t_1]$, the most popular contents are recorded in $BF_k$, and we will also record them in a small hash table for accurate popularity counting. The detailed design of this idea is presented in Section III.

While Bloom filters are inherently memory-efficient when recording the content names, three optimization schemes are also put forward to improve the speed of popularity monitoring, including merging the Bloom filters to reduce the number of memory accesses, balancing the access burden among all the Bloom filters, and using multi-thread programming to hide memory access latencies, etc.

Especially, in this paper, we make the following contributions:

1) We propose a memory-efficient Bloom filter-based method to capture popularity statistics at high speed, without the need of any dedicated infrastructure, or any prior knowledge about the traces. Both theoretical and experimental analyses elucidate very low relative error of our method. Three effective optimization schemes are also proposed to further boost the popularity monitoring speed;
2) Beyond the *fixed window* monitoring scheme, we propose the *sliding window* scheme to obtain *timely update* of the popularities in the past time window. This scheme also provides support for extremely fast monitoring speed.
3) Using real trace as input and setting the monitoring time window to 30 min, our method can achieve an accurate monitoring speed of 20.92 M/s, while only consuming around 32 MB memory. This speed is equivalent to 16.74 Gbps assuming the content length is 100 Bytes in average. With a synthetic trace to simulate the real environment on router line cards, our method can even achieve 251.07 M/s (equivalent to 200.86 Gbps) because of reduced memory access latency.
4) We provide a real trace-driven comparison between Least Frequently Used (LFU) and Least Recently Used (LRU) caching policies, evaluation results show that LFU achieves higher hit rate than LRU with much less unnecessary cache replacements.

The rest of the paper is organized as follows: Section II introduces the background knowledge on NDN related to this paper. Section III presents the detailed design of the Bloom filter-based online popularity method. Section IV analyzes the relative error from the theoretical viewpoint, and then put forward optimization methods. Section V evaluates our method in terms of popularity monitoring speed, accuracy and memory consumption. A real trace-driven comparison between our method and static sampling is conducted. Section VI surveys related work and Section VII concludes the paper.

## II. NDN Background

NDN is a novel network architecture proposed by [1]. Different from current Internet practice, it makes content ("what") as its central role, rather than "where" content is located. NDN has a vast background that we cannot fully cover, in this section we briefly introduce NDN with some basic background and a focus on its in-networking caching property.

### A. Naming

A critical distinction from IP is that, every piece of content in NDN network has an assigned name to uniquely identify it, and packets are routed/forwarded by names, rather than IP addresses. NDN names are application-dependent and opaque to the network, but they all share the common characteristics – hierarchically structured and composed of explicitly delimited *components*. A typical example of an NDN name is a reversed domain name followed by a directory-style path, e.g., *org/ieee-conference/2014/cfp.html*, where *org/ieee-conference/* is the reversed domain name of *ieee-conference.org*, and *2014/cfp.html/* is the content's directory path on the website server. '/' is the component boundary delimiter; *org, ieee-conference, 2014* and *cfp.html* are the 4 components of this name.

### B. NDN Communication

NDN adopts a requester-driven, data-centric communication mechanism. All communications in NDN are performed using two distinct types of packets: *Interest* packet and *Data* packet. Both types of packets carry a content name in the packet header, which uniquely identifies a piece of content. In essence, Interest packet is the request and Data packet is the response. To retrieve data, a consumer sends out an Interest packet with the name of the desired content in its header. Routers use this name to route and forward the Interest packet towards data sources, and a Data packet that carries both the name and the desired content is returned to the consumer following the reverse path of the Interest packet.

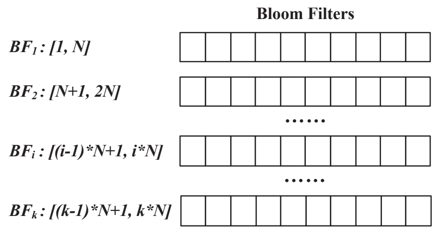
### C. Caching

Each NDN router is equipped with a Content Store (CS) that can strategically store Data packets passing by this router. Upon receiving an Interest packet, an NDN router first checks the Content Store. If there is data whose name falls under the Interest's name, the data will be immediately sent back as a response, which means a *cache hit*. The Content Store, in its basic form, is just the buffer memory on line cards of today's routers. Both IP routers and NDN routers buffer data packets. The difference is that IP routers cache packets mainly for packet scheduling and queuing, and cannot reuse the data after forwarding them, while NDN routers are able to reuse the data since they are identified by persistent names. For static contents, NDN is believed to achieve almost optimal data delivery. Even dynamic content can benefit from caching in the case of multicast (e.g., teleconferencing) or packet retransmission in case of packet loss. Cache management and replacement is very crucial for efficient content delivery and cache storage utilization. Prior work on caching policy suggested that the LRU or LFU policy would be desirable, and more simpler and efficient policies are under active research.

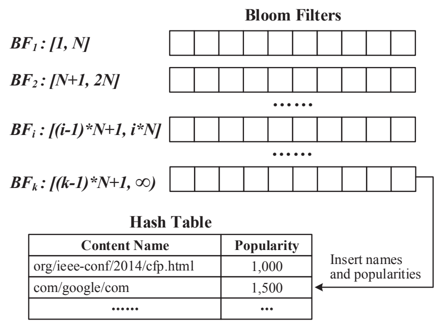
## III. Bloom Filter Based Popularity Monitoring

NDN routers have a Content Store to cache data, which is a novel property introduced by NDN. It should be provisioned to be able to store a large amount of *popular* contents for certain period of time (much larger than round-trip time as in packet buffer), and its purpose is to maximize cache hit rate so that upstream traffic is minimized.

[1]By "inserting a content object into a Bloom filter", we actually mean inserting the name of that content object into the Bloom filter.



(a) $k$ Bloom filters for popularity monitoring.



(b) $k$ Bloom filters and a small hash table for popularity monitoring.

Fig. 1: Bloom filter-based popularity monitoring.

Intuitively, the most straightforward solution towards popularity counting is to keep track of the full reference history of every content object requested through a router, and then count how many times each content object is requested. Recording the popularities can be accomplished by a hash table, with the content name as the key and the popularity as the value. This *off-line* method is obviously unscalable due to the huge scale of the content traffic, storing so many names into such a hash table will consume extremely large memory space. Moreover, the hash table cannot catch up with the high-speed of router line cards, either. Faced with this problem, we will propose a Bloom filter-based on-line method to capture the popularity statistics at high speed and high accuracy.

## A. Bloom Filter-based Popularity Monitoring

Bloom filter is a space-efficient data structure used to test whether an element is a member of a set. It is widely used in network applications.

We utilize $k$ Bloom filters to construct an on-line popularity monitoring system, with each Bloom filter being endowed to represent a popularity range, and those content objects whose popularities belong to this range are stored in this Bloom filter. As Fig. 1(a) shows, $BF_1$ stands for popularity range $[1, N]$, and it keeps track of the content objects with popularities falling into this range. Similarly, $BF_2$ corresponds to popularity range $[N+1, 2N]$, and $BF_i$ corresponds to popularity range $[(i-1)*N+1, i*N]$. Allowing for the fact that the top popularities could be very high, a more flexible and practical design would be letting the Bloom filter with the highest subscript ($BF_k$) to stand for popularity range $[(k-1)*N+1, \infty)$. And because there may be large differences among the popularities in $BF_k$, a small hash table is used to recode the content objects in $BF_k$ and their *accurate* popularities, as illustrated in Fig. 1(b). In the remainder of this paper, we use the term "popular content objects" to refer to the set of objects stored in $BF_k$.

The logic for popularity monitoring is as follows: each time a specific content object arrives, its popularity is increased by 1. Obviously, when a content object arrives for the first time, its popularity is increased from 0 to 1. If the current popularity is $p$, then it will be increased to $p+1$. Based on this logic, we will examine how the Bloom filters work to capture the popularity statistics.

**Algorithm 1** Bloom filter-based Popularity Monitoring

```
procedure BF_Popularity_Monitor(ContentName name)
    found_in_all_BF ← TRUE;
    for i : 1 → $k$ do
        if Query(BF_i, name) = FALSE then
            found_in_all_BF ← FALSE;
            if i = 1 then                ▷ name comes for the first time.
                Insert(BF_1, name);
                break;
            else if i = k then
                Insert(BF_k, name);
                Hash_Table.insert(name, (k-1)*N+1);
            else
                r ← rand()%N                ▷ r∈[0,N-1]
                if r = 0 then            ▷ Insert with probability 1/N
                    Insert(BF_i, name);
                    break;
    if found_in_all_BF = TRUE then        ▷ found in BF_k
        Hash_Table[name]++;
```

Initially, all the BFs are empty. When a content object $O$ arrives, it is queried in all the $k$ Bloom filters. If $O$ comes for the first time, it will not be found in any Bloom filter (assume no false positive happens, refer to Section III-B for false positive rate analysis). Therefore, $O$'s popularity is increased from 0 to 1. Then, $O$ will be inserted into $BF_1$, whose popularity range $[1, N]$ includes $O$'s current popularity.

Alternatively, if $O$ have arrived for more than 1 but no more than $N$ times, ideally, $O$ will only be found in $BF_1$. This query result indicates that $O$'s popularity $p$ is $1 \leq p \leq N$, but it has no clue about how much $O$'s popularity exactly is. Easily we have the insight that $p$ has equal probability to be any number between 1 and $N$ (inclusive), i.e., $p = p'$ ($p' \in [1, N]$) with probability $1/N$, which means $Pr\{p = N\} = 1/N$. If $p = N$, $O$'s new popularity should increase to $N+1$, which falls into range $[N+1, 2N]$. As a result, if $O$ is currently only in BF$_1$, it will be inserted into $BF_2$ (and also stays in $BF_1$) with probability $1/N$. Similarly, if $O$'s current popularity lies in $[N+1, 2N]$, it will be found in both $BF_1$ and $BF_2$. So $O$ will be inserted into $BF_3$ with probability $1/N$. Therefore, a content object may not be only stored in one Bloom filter, but instead in $m$ continuous Bloom filters, starting from $BF_1$ (i.e., $BF_1 \sim BF_m$, $1 \leq m \leq k$).

Generally, for each content object $O$ that goes through a router, its name will be looked up against all the $k$ Bloom filters. The lookup result indicates its popularity range. Ideally, two kinds of results will be returned when no false positive happens: 1) no Bloom filter contains $O$, this indicates that $O$ comes for the first time and its popularity now is 1, then $O$ will be inserted into $BF_1$; 2) $m$ Bloom filters with continuous subscripts ($BF_1 \sim BF_m$) contain $O$, this indicates that $O$'s popularity falls into the range that $BF_m$ stands for, and $O$ will be inserted into $BF_{m+1}$ with probability $1/N$. This popularity monitoring process is illustrated in Algorithm 1. Once a content object $O$ is inserted into $BF_k$, a new entry will be created in $BF_k$'s associated hash table, and $O$'s popularity is set to $(k-1)*N+1$. The entry's key is the content name and the value is the popularity. For the following times $O$ is found in $BF_k$, the corresponding entry will increase its popularity by 1.

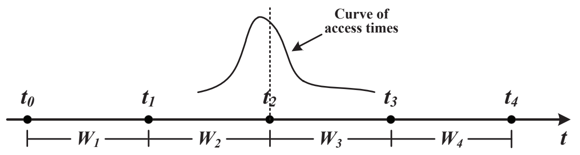


Fig. 2: Fixed window popularity monitoring scheme.

Apparently, except the content objects stored in the hash table, our proposed method cannot accurately tell how much an object's popularity exactly is, but can only tell a range. So we adopt an estimation function $f()$ to estimate the real popularity $\tilde{n}$.

$$\tilde{n} = f(m) = (m-1) * N + \frac{1+N}{2} \quad (1)$$

where $m$ is the number of continuous Bloom filters (starting from $BF_1$) that contain the object. Formula 1 is essentially the average value of the popularity range that $BF_m$ stands for.

## *B. Impacts of False Positive*

Bloom filters will introduce false positives, which would lead to errors when querying the $k$ Bloom filters for a content name. Assume that a name $y$ is stored in $BF_1$ to $BF_m$, querying $y$ will return an incorrect popularity under two cases: 1) a false positive happens on $BF_{m+1}$, since false positives on other Bloom filters can be detected because of the un-continuous subscripts. Due to the independence of each Bloom filter, the probability that a false positive happens on $BF_{m+1}$ is just its false positive rate $f$, which could be very low (our experimental setting make it as low as $10^{-8}$). 2) The other case is name $y$ is not recorded in any Bloom filter, but $m$ continuous Bloom filters (starting from $BF_1$) have false positives simultaneously. This would have a probability of $\frac{1}{\binom{k}{m}} * f^m$. If $k = 6$ and $m = 3$, $\frac{1}{\binom{k}{m}} * f^m = 5 * 10^{-26}$. Both cases have negligible false positives and will not affect the monitoring accuracy of our method. So we omit the influences brought by the false positives when analyzing the relative error in Section IV.

## *C. Fixed Window Popularity Monitoring*

The popularity monitoring method proposed above can be used within a given time interval. We divide the time axis into continuous fixed-length intervals, and each interval is called a *window*, as depicted in Fig. 2. Within a time window $[t_0, t_1]$, our method can be used to capture the popularities.

When it arrives at time $t_1$ – the end of the window $[t_0, t_1]$, the monitoring results will be saved, and the BFs and hash table will be cleared. Then another round of popularity monitoring round begins for time window $[t_1, t_2]$.

Since the popularities are obtained within a fixed-length time window, we call this scheme *Fixed Window Popularity Monitoring*, and the popularity can be viewed as the access frequency in this window. It's simple to understand and easy to implement, but it has an important deficiency. If a content object is frequently requested just at the junction of two continuous time windows, as depicted by the curve in Fig. 2, its popularity will be divided into two parts, one part counted in the first window, and the other part counted in the second window. As a result, the popularity of this object may not be accurately reflected.

Meanwhile, The time window could be short and long, from minutes to hours, with pros and cons. For a long time window, it can reflect the popularity over a longer past period, but it 1) has coarse time granularity, hence the responses to popularity changes are not timely enough (since cache decisions are made at the end of each window, a coarse time granularity makes the caching policy not sensitive enough to popularity changes); 2) requires more memory space, which would degrade the monitoring speed. For a short time window, though it consumes less memory, yet it also has shortcomings. A popular object during a short time window will not be popular in next window. So this object will be inserted into and immediately then evicted from cache, imposing negative influences on the cache performance. Moreover, long-period popular content objects may not be viewed as popular by some short time windows (and be evicted from cache). Therefore, letting cache policies make decisions based on these short-period granularity is not wise. Choosing a suitable monitoring time window is an important but non-trivial task.

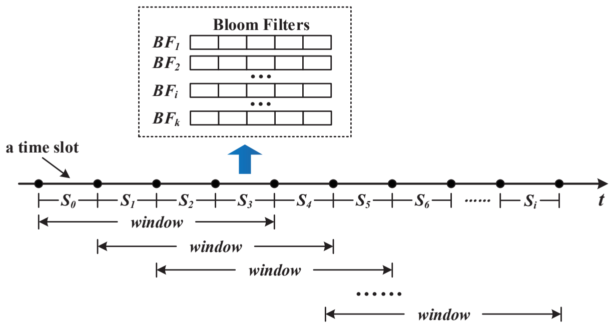


Fig. 3: Sliding window popularity monitoring scheme.

## *D. Sliding Window Popularity Monitoring*

Witnessing the deficiencies that the fixed window monitoring scheme encounters, we propose a *sliding window* scheme to overcome those shortcomings. The first step is to split the time axis into continuous time slots, which are fixed time spans shorter than a time window, as illustrated by Fig. 3.

In this scheme, a time window consists of multiple continuous time slots. Within each time slot in that window, the Bloom filter-based popularity monitoring method is used to capture popularities. (Cache decisions are made at the end of each time slot.) Each time slot maintains a small hash table $h$ to store the names of top popular content objects in this slot and their associative popularities. The popularity for each content during a window is the accumulation of the popularities within all the time slots in that window. Therefore, a window maintains a larger hash table $H$ that merges the small hash tables of all the time slots enclosed in the window. The entries with the same key (content name) are merged into one entry, with their popularities summing up as the value of that entry. As time elapses, a window moves forward to absorb one time slot ahead, and evict the time slot at the tail. Accordingly, the hash table entries in the evicted time slot are deleted from $H$, and hash table entries in the absorbed time slot are inserted into $H$. (Particularly, for the entries with same key, popularity is subtracted by the popularity in the evicted hash table and added by that in the absorbed hash table.) For unpopular objects in the Bloom filters, invoke $f()$ on each Bloom filter group for each slot in the window and sum up the results. In this way, we can implement the *Sliding Window Popularity Monitoring*, and the sliding step is the length of a time slot.

The most important property that the sliding window scheme brings is the more timely update of content popularity, which means more frequent update at the end of each slot, rather than only one update per entire window. This benefit is at the cost of more management expenses, such as more frequent hash table merges and evictions. But these operations can be handled in parallel or in background with the next iteration of popularity monitoring, only requiring that they finish before the next iteration ends. Alternatively, these monitoring results could even be exported to a remote analysis host for processing [14], [15]. Note that in this scheme, the popular contents can still be directly found in hash table $H$, while the unpopular ones need to query for themselves in the Bloom filters of the time window and invoke the estimation function $f()$.

It's worth pointing out that we do not intend to save memory consumption by this sliding window scheme, because the Bloom filters for all the slots in the window has equivalent memory consumption as those for the entire window in the fixed window scheme. (Because both schemes record the same number of objects in a window, and the false positive rates are nearly the same.) However, we do have got an opportunity by the sliding window scheme to accelerate the popularity monitoring speed, because only one slot's Bloom filter group is active at any time. And a single time slot's Bloom filter group is much smaller than that of a time window's, thus we have the possibility to put the whole Bloom filter group, or a subset of the Bloom filters into smaller but faster SRAM chips (refer to Section V-E for experimental results).

## IV. Analysis and Optimization

In this section, we first analyze the theoretical relative error of our popularity monitoring method. Afterwards, a mathematical model is established to find the relation between the number of Bloom filters and popularity monitoring accuracy. Afterwards, several optimizing schemes are proposed to boost Bloom filter querying speed.

### A. Analysis

Assume a content object $O$'s actual popularity is $n$. As aforementioned, we cannot tell this accurate value, but can only develop an estimation function $f(m)$ to calculate an estimation popularity $\tilde{n}$, i.e.,

$$\tilde{n} = f(m) = (m-1)*N + \frac{1+N}{2} \tag{2}$$

where $m$ is the number of continuous Bloom filters (starting from $BF_1$) that contain $O$. In fact, the actual popularity $n$ could be expressed by:

$$n = \lfloor \frac{n}{N} \rfloor * N + a \tag{3}$$

where $a = n\%N$.

By comparing 2 and 3 we can find that, obviously, $(m-1)*N = \lfloor \frac{n}{N} \rfloor * N$ (because $m$ can be derived by $\lfloor \frac{n}{N} \rfloor + 1$), while $(1+N)/2$ is not necessarily equal to $a$. Therefore, $E[\tilde{n}] = E[f(m)]$ is not necessarily equal to $n$ ($E[f(m)] \neq n$), which means $f(m)$ is not an unbiased estimation of $n$.

Next we estimate the relative error of $f(m)$.

$$\begin{aligned}
Var[\tilde{n}] &= E[\tilde{n}^2] - E^2[\tilde{n}] = E[f^2(m)] - E^2[f(m)] \\
&= E[\{(m-1)*N + \frac{1+N}{2}\}^2] - E^2[(m-1)*N + \frac{1+N}{2}] \\
&= E[\{\left\lfloor \frac{n}{N} \right\rfloor * N + \frac{1+N}{2}\}^2] - \{E[\left\lfloor \frac{n}{N} \right\rfloor * N + \frac{1+N}{2}]\}^2 \\
&= N^2\{E[\left\lfloor \frac{n}{N} \right\rfloor^2] - E^2[\left\lfloor \frac{n}{N} \right\rfloor]\}
\end{aligned} \tag{4}$$

Let $r = \lfloor \frac{n}{N} \rfloor$, then

$$Var[\tilde{n}] = N^2 Var[r] \tag{5}$$

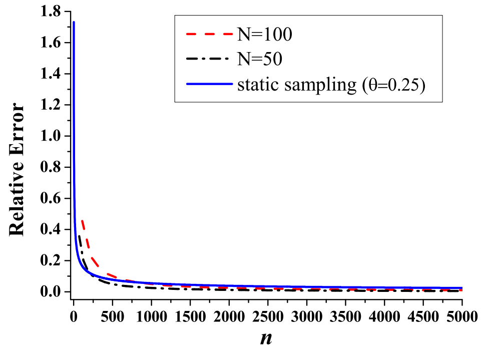


Fig. 4: Theoretical results of relative error.

The relative error (measured by Coefficient of Variation) is:

$$\frac{\sqrt{Var[\tilde{n}]}}{E[n]} = \frac{N\sqrt{Var[r]}}{E[n]} \tag{6}$$

Obviously, Formula 6 reveals that with the same variance, the larger $n$, the smaller the relative error. This means the relative error diminishes as the actual popularity increases. Theoretical results of the relative error derived from Formula 6 are illustrated in Fig. 4, which also compares our method with the static sampling approach (sampling rate $\theta = 0.25$). We can learn that Bloom filter-based method has lower relative error as the popularity grows larger, and the relative error of $f()$ is very low.

### B. Optimization Schemes

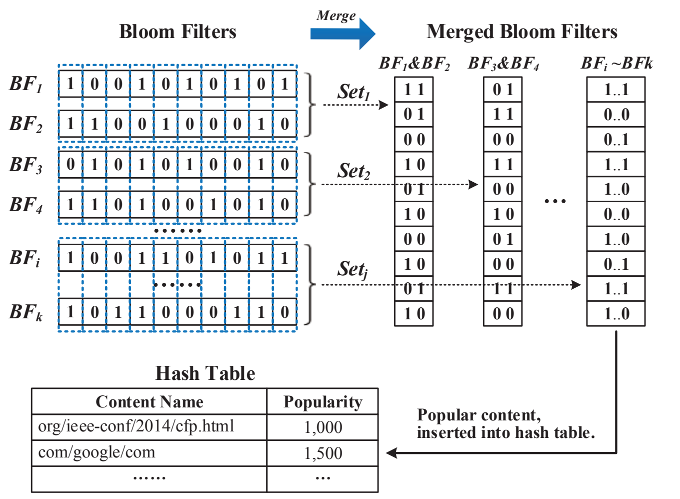
*1) Merging Bloom Filters:* Querying all the Bloom filters for each arrival content object is very time-consuming since it leads to a large number of memory accesses. Inspired by the usage of Bloom filters in [16], [17], we try to merge the Bloom filters by placing the bits at the same position in different Bloom filters into continuous bits in the memory. In this way, querying all the Bloom filters is like only querying one Bloom filter. However, this optimization scheme requires the Bloom filters to have the same length.

The false positive rate of a Bloom filter can be calculated by Formula 7, where $X$ is the number of elements inserted and $M$ is the number of bits in the array. From this formula, we can derive the formula to calculate $M$ given a specific false positive rate $f$, as shown by Formula 8 [18]. In our method, since each Bloom filter stores different number of content objects and they all maintain the same false positive ($10^{-8}$), the length of the Bloom filters are different according to Formula 8.

$$f = (1 - e^{-nX/M})^n \tag{7}$$

$$M = -\frac{X \ln f}{(\ln 2)^2} \tag{8}$$

Instead of allocating the same number of bits to all the Bloom filters, we propose to divide them into sets, and each set contains Bloom filters with continuous subscripts. Bloom filters in the same set are allocated the same number of bits, as illustrated in Fig. 5. For example, $BF_1$ and $BF_2$ stores the most and second-most names. Normally, they are built up with different lengths, and $BF_1$ will be longer than $BF_2$. In order to merge them together, we allocate the same number of bits to both $BF_1$ and $BF_2$, equal to the length of $BF_1$ (to keep low false positive rate of $BF_2$). In this way, $BF_1$ and $BF_2$ can be merged, the bits at the same position in $BF_1$ and $BF_2$ are placed together in continuous bits in the memory. Querying $BF_1$ and $BF_2$ now requires $n$ memory accesses, rather than $2n$ times, so the



Hash Table

| Content Name | Popularity |
|---|---|
| org/ieee-conf/2014/cfp.html | 1,000 |
| com/google/com | 1,500 |
| …… | … |

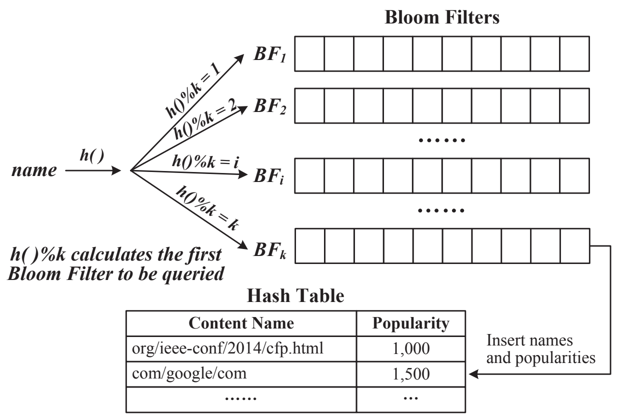
Fig. 5: Merging Bloom filters. Note that for simplicity, the Bloom filters in this figure are of equal size, but actually they are not. As an example of merging Bloom filters, $BF_1$ and $BF_2$ have the same number of bits $M_1$, $BF_3$ and $BF_4$ have the same number of bits $M_2$, and $B_i{\sim}B_k$ have the same number of bits $M_3$. $M_1 > M_2 > M_3$.

memory access latencies are shortened. While $BF_2$ may waste a little memory space, this extra cost is negligible compared with reduced memory access latency. As an example, Fig. 5 also shows the merging of $BF_3$ and $BF_4$, as well as $BF_i$ to $BF_k$. Merging Bloom filters can effectively improve popularity monitoring speed (refer to Section V).

*2) Balancing the Accessing Burden on Bloom Filters:* Algorithm 1 shows that in the popularity monitoring process, the querying process always begins with the first Bloom filter, which incurs the largest querying and inserting burden on $BF_1$. Merging the Bloom filters can reduce memory access times, but cannot relieve this access burden on $BF_1$, which could be the bottleneck of the monitoring speed. The reason why the querying process starts from $BF_1$ is that, the popularity of a object always starts from 1, and it is stored in a sequence of continuous Bloom filters, say $BF_1, BF_2, BF_3 \cdots BF_m$. In fact, we can break the rule of always querying from BF$_1$.

Now we offer an optimization scheme to skip the first $z$ Bloom filters when querying or inserting into the Bloom filters, aiming to *balance* the accessing burden on all the Bloom filters. As Fig. 6 illustrates, when a content object comes, we check its name in the Bloom filters, but not necessarily starting from $BF_1$. Instead, the first Bloom filter to query is determined by the name's hash value of a hash function $h()$, i.e., $h(name)\%k$. In this way, the Bloom filters that this object needs to query, or insert are at most $BF_{h(name)\%k} \cdots BF_k$. The querying process would stop at an intermediate Bloom filter $BF_i$ if this content object is not present in $BF_i, i \in [h(name)\%k, k]$. Accordingly, the probability that this object is inserted into $BF_i$ is $\frac{1}{(i-1)*N}$. As a result, $z = h(name)\%k - 1$ Bloom filters are skipped and the access burden on $BF_1$ is relieved. Even though $h()$ is not perfectly unbiased, the monitoring speed will be significantly advanced. By this method, $BF_1$ is no longer the bottleneck, and other methods such as pipeline and multi-thread will be more functional.

Note that if an object comes for the first time (has popularity 1), and it is queried from, say, $BF_3$, it may not be inserted into $BF_3$ due to the mismatch of its popularity and $BF_3$'s popularity range. This means that, some content objects with very low


Bloom Filters
name
h()
h()%k = 1
h()%k = 2
h()%k = i
h()%k = k
$BF_1$
$BF_2$
$BF_i$
$BF_k$
h( )%k calculates the first Bloom Filter to be queried
Insert names and popularities

Hash Table

| Content Name | Popularity |
|---|---|
| org/ieee-conf/2014/cfp.html | 1,000 |
| com/google/com | 1,500 |
| …… | … |

Fig. 6: Balance the access burden on all the Bloom filters.

popularity may not be counted by the Bloom filters, but a really popular content object will always be counted, not matter which Bloom filter it starts to query since it will come for enough times.

*3) Multi-thread Acceleration:* Counting one object needs operations of hash calculation, random number generation (the insertion probability) and memory accesses in serial. While we are trying to reduce memory accesses (since they are most time-consuming), we are also aware that we can calculate the hash values of other objects while the CPU is waiting for the memory accesses to finish. This parallelism approach will further boost the monitoring speed significantly, and it's naturally to implement this parallelism by multi-thread programming. If we assign identical objects to the same thread, we can even look forward several cells in the incoming queue for some identical objects, and need only to conduct one round of counting operations for more than one identical objects. Evaluation results in Section V show that multi-threads remarkably improves the popularity monitoring speed.

## V. Evaluation

This section evaluates the Bloom filter-based popularity monitoring method mainly from 3 aspects: 1) monitoring accuracy, 2) monitoring speed and 3) memory consumption. Besides, a brief comparison between our Bloom filter-based method and the static sampling method is also provided. At last, we conduct a comparative study between LFU and LRU based on real traces.

### A. Experimental Setup

*1) Trace:* Our method takes content objects and their names as input, which could be extracted from an NDN trace. However, since NDN has not been deployed, there is no real NDN trace. For this reason, we capture a 1-hour trace from an international gateway link (10 Gbps) of a major ISP[2], and want to translate it into an NDN trace by mapping a request/response pair in the IP trace to an Interest/Data packet pair in NDN. However, it's difficult to parse out all the requests in the trace, which involves complicated application identification and rule matching, and it is not the focus of this paper. For the purpose of quickly testing our proposed method, we use HTTP requests as the input since they are easy to parse out. Eventually, we parsed out 21.55 Million HTTP requests, containing 10.83 Million unique ones (containing unique URLs). The ground truth popularity distribution of this trace is presented in Fig. 7, which elucidates that the popularity can be as high as close to $10^6$, and the popular objects account for only a small portion among all the objects.

[2]Public traces available on line cannot satisfy our requirements since we need payloads of the packets. Public traces only contain packet headers.

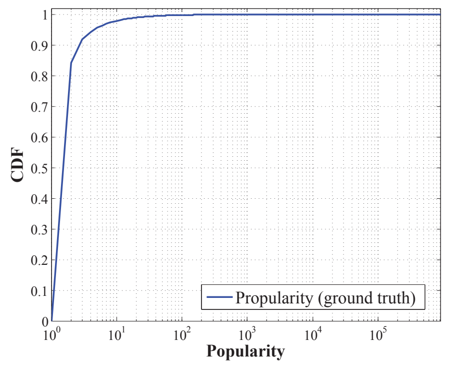


Fig. 7: Ground truth popularity distribution.

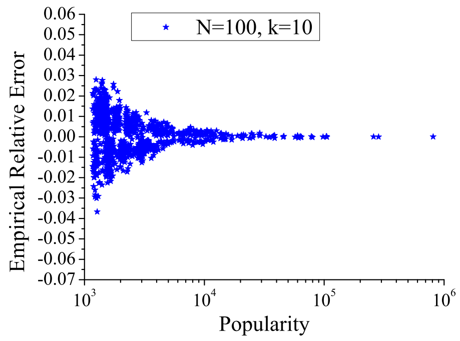


Fig. 8: Empirical relative error (N=100, k=10).

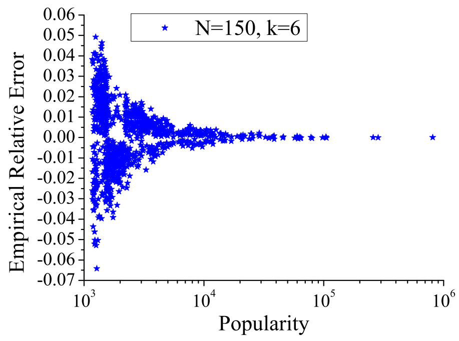


Fig. 9: Empirical relative error (N=150, k=6).

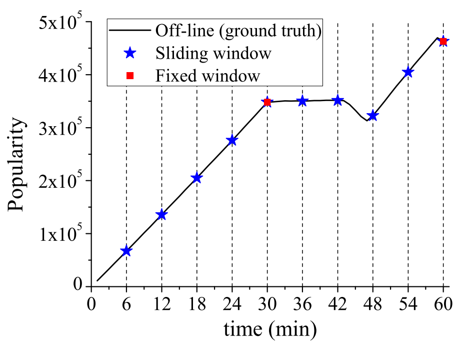


Fig. 10: Compare popularities monitored by the fixed window, sliding window scheme and the ground truth.

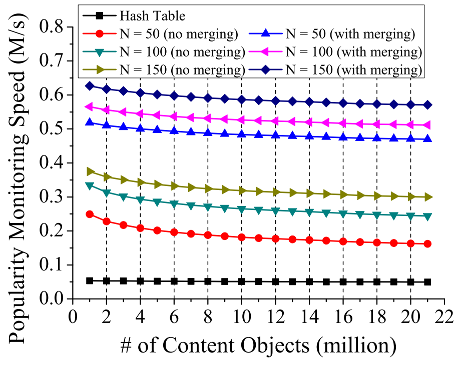


Fig. 11: Single-thread popularity monitoring speed on real URL trace. Hash Table has the lowest speed.

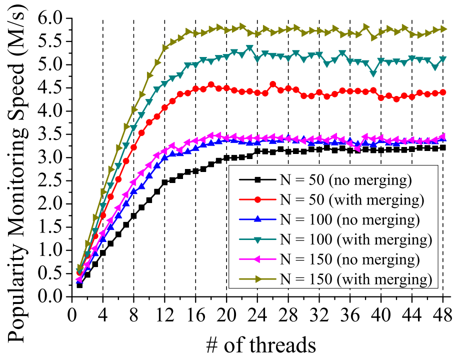


Fig. 12: Popularity monitoring speed with multi-thread and merging optimization methods on real URL trace.

*2) Platform:* The experiments are conducted on a server with Intel Xeon E5520 CPU (2.27 GHz), and 15.9 GB main memory. The CPU integrates a small cache (SRAM) between the main memory (DRAM) and the cores.

## B. Monitoring Accuracy

We first count the content popularity off-line, and obtain the real popularity distribution of the whole trace, as shown in Fig. 7. (This result also implies the Zipf-like distribution of the requests in the trace.) Then we run our popularity monitoring method to compare with the off-line results, with popularity range length for each Bloom filter set to different numerical numbers, $N = 50,\ 100$ and 150. The corresponding $k$ is 20, 10 and 6, respectively. Here *sliding-Window Monitoring* scheme is used, and time window $W = 30\ min$, time slot $s = 6\ min$, meaning that each time window consisting of 5 continuous slots. At the end of the time window, the hash table for popular content objects records 1,160 objects when $N = 50,\ k = 20$; 1,038 objects when $N = 100,\ k = 10$; and 1,278 objects when $N = 150,\ k = 6$. The popularities can be calculated from the Bloom filters and the hash table using $f()$. To evaluate the accuracy of popularities obtained by our method compared with the original distribution, next we calculate the *relative error*.

We select the most popular 1,000 content requests from the parsed HTTP requests and study how much their popularities obtained by the sliding window scheme are deviated from the real popularities. Here we define the relative error as $d = (real\ popularity - monitored\ popularity)/real\ popularity$. The relative error results are shown in Fig. 8 and 9, which reveal that the relative errors are very low. The higher the popularity, the more accurate the monitoring results, and this is consistent with the results in Fig. 4.

## C. Comparing Sliding Window and Fixed Window

Now we compare the performance of the sliding window and fixed window schemes. The time window is still 30 min for the fixed window scheme.

The sliding window is proposed to obtain more timely update of popularities in the past time window, which is just the property that the fixed window scheme lacks. To verify this property, we study the popularity of a randomly picked content object monitored by both the fixed window and sliding window schemes, and compare them with the ground truth. Fig. 10 presents the experimental results. The solid line in the figure shows the popularity of the 30-min time window calculated by the off-line method, which is the ground truth. The scatter stars represent the results obtained by the sliding window scheme, which are just like the frequent snapshots of the ground truth. The scatter squares represent the results of the fixed window scheme, only two squares in the figure indicate that the fixed window scheme is too lazy to update the popularity. Therefore, it's obvious that the sliding window scheme updates more frequently than the fixed window scheme.

It is worth pointing out that because the results of the two schemes achieve very close approximations of the real popularity, the line and the scatters overlap in Fig. 10.

## D. Monitoring Speed with Real URL trace

High-speed on-line popularity monitoring is very crucial to the performance of our method, we measure the monitoring speed in terms of how many content objects can be counted within a time unit, under different settings and optimization methods. In the following experiments, we will apply the three optimizations one by one.

We first conduct experiments by merging the Bloom filters, with range length $N = 50,\ 100,\ 150$. For $N = 50,\ 100$ and 150,

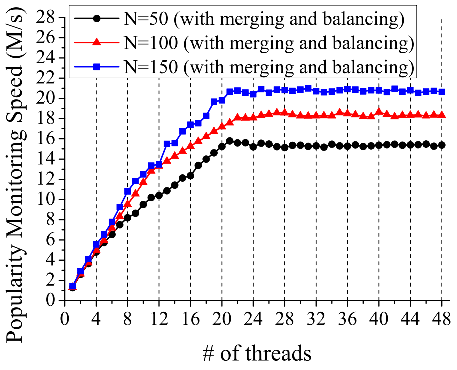


Fig. 13: Popularity monitoring speed with multi-thread, merging and balancing optimization methods on real URL trace.

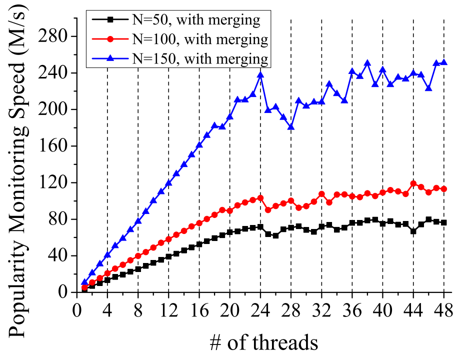


Fig. 14: Monitoring speed with multi-thread and merging optimization methods using synthetic URL trace.

TABLE I: Compare BF-based Monitoring with Static Sampling

| | Monitoring Speed (M/s) | Memory Cost (MB) | Performance Cost Ratio $\alpha$ |
|---|---|---|---|
| BF-based(N=50) | 1.27 | 32.64 | 0.038 |
| BF-based (N=100) | 1.31 | 32.52 | 0.040 |
| BF-based (N=150) | 1.43 | 32.47 | 0.044 |
| SS ($\theta$=0.10) | 0.47 | 149.68 | $3.14 * 10^{-3}$ |
| SS ($\theta$=0.25) | 0.29 | 420.92 | $6.89 * 10^{-4}$ |
| SS ($\theta$=1.00) | 0.056 | 1,019.79 | $5.49 * 10^{-5}$ |

$k = 20, k = 10$ and 6, respectively. After merging, $k = 11$, 6 and 4 for $N = 50$, 100 and 150, respectively. The input trace is the URL trace we extracted from our captured IP trace. Then we obtain the monitoring speeds under the merging optimization method, and compare these results with those monitoring speeds obtained without merging the Bloom filters, as well as compare with the hash table method. Experimental results are presented in Fig. 11, which shows that, by merging the Bloom filters and when $N = 150$, the monitoring speed reaches the highest 0.63 million content objects per-second (M/s) with single thread; and hash table has the lowest monitoring speed (around 0.05 M/s). Knowing the potential performance improvement of the multi-thread optimization method (Section IV-B3), we re-conduct the same experiments with multiple threads, the results are presented in Fig. 12, which reveals that the highest speed reaches 5.82 M/s ($N = 150$, by merging the Bloom filters) with more than 24 threads.

This speed, however, is still too low compared with the link line-speed. Based on the merging optimization scheme, we continue to adopt the access-balancing optimization method introduced in Section IV-B. Experimental results are present in Fig. 13, which shows that the monitoring speed are markedly advanced, since $BF_1$ is no longer the bottleneck. The highest monitoring achieves 20.92 M/s. Assume that the average length of a content object is 100 Bytes, this speed is equivalent to 16.74 Gbps, which is higher than the OC-192 line-speed.

We can further compare these results with the performance of static sampling (SS). We define the performance cost ratio $\alpha$ as $Monitoring\ Speed/Memory\ Consumption$, and Table 1 lists the comparison results under 1 thread. As we can see, the $\alpha$ of the BF-based method is much higher than those of the SS method, which means the BF-based method is more efficient.

## E. Monitoring Speed with Synthetic URL trace

In the previous experiments, the input URL trace is stored in the DRAM chip due to its large size, together with the Bloom filters. Each URL needs to be read from the DRAM chip and then processed by CPU. Reading all the URLs from DRAM during the experiments will cost a long time and influence the CPU's cache replacements (Bloom filters may be evicted out from CPU's cache), which harms the performance and is inconsistent with the actual environment on line cards. On a line card, traffic from the interface will enter an incoming FIFO waiting for processing, the FIFO is generally implemented by on-chip memory, such as MRAM in FPGA or SRAM in ASIC chips. Obviously, fetching traffic from a FIFO will be much more faster than reading out from DRAM memories.

Observing this problem, we re-conduct the above experiments using synthetic URL trace with the sliding window scheme. We also merge the Bloom filters to reduce memory accesses, but do not adopt the load balancing optimization method at this time[3]. The URL trace is generated by CPU at high speed, with an average URL length of 50 bytes. The time generating each URL takes several CPU cycles, which is comparable to reading out a URL from a FIFO. With the help of the small cache between the CPU cores and DRAM memory, the monitoring speed is significantly advanced, which is illustrated in Fig. 14. This figure shows that the monitoring speed grows dramatically as the number of threads increases, and the highest speed achieves 79.78, 119.14 and 251.07 million content objects per second (M/s) for $N = 50, 100$ and 150, respectively. The reason for this significant speed advance is twofold: 1) we no longer need to read URLs from DRAM; 2) besides this, the sliding window scheme only has a small group of active Bloom filters (compared with fixed window scheme) at any time, so the cache can now store a subset or all of the Bloom filters. Typically, a single off-chip memory reference would take 55 ns, and an on-chip memory access would only cost 3 ns. Therefore, the memory accesses are significantly accelerated, resulting in the speed improvement.

## F. Memory Consumption

Memory consumptions of the Bloom filters and the static sampling method are also listed in Table 1. Based on experiments, for $N = 50$, 100 and 150, the memory consumption of the Bloom filters are 32.64 MB, 32.52 MB and 32.47 MB ($W = 30\ min$) with false positive rate of $1.0009 \times 10^{-8}$. However, the hash table will take 1,019.79 MB, about 32 times higher. Note that the sliding window scheme is not supposed to reduce memory consumption, because using either scheme, the number of objects to record within a time window does not alter. Summing up the memory consumption for each slot in the sliding window scheme will approximate the memory consumption of the fixed window design.

## G. LFU and LRU Performance

*1) Trace locality:* The LRU cache policy [19] and its variants (such as SLRU [20]) have been used in flow cache, and exhibit

[3]The reason is that we want to examine the performance improvements brought only by the reduced memory access latency, excluding the effect of load balancing.

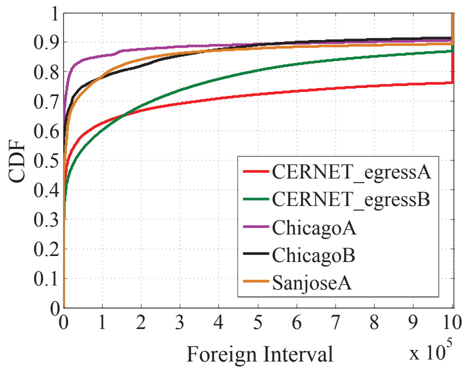


Fig. 15: Packet-level flow locality.

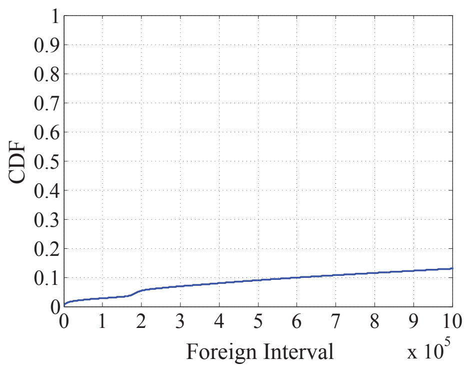


Fig. 16: Locality of content objects.

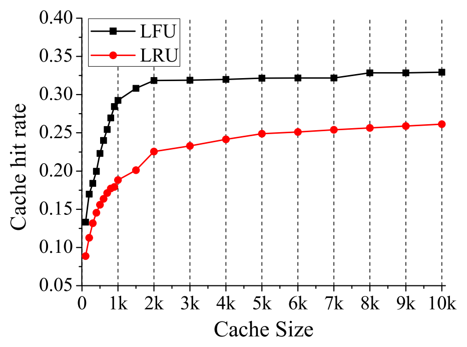


Fig. 17: Cache hit rate of LFU and LRU under different cache sizes.

good performance. This is mainly because IP flows have relatively good packet-level locality. To measure locality, we define *foreign packet* as the packet between the closest two packets that belong to the same flow, and the *foreign interval* is the number of such foreign packets between two closest packets in the same flow. The locality of 5 traces are shown in Fig. 15. Among the traces, *ChicagoA*, *ChicagoB* and *SanjoseA* are from CAIDA [21] and *Gateway_egressA* and *Gateway_egressB* are captured from the same gateway link of the ISP above. We can figure out that the shorter the foreign interval, the closer the packets within the same flow, hence the better the packet-level locality. Let $I$ denote the foreign interval, Fig. 15 shows that for *ChicagoA*, *ChicagoB* and *SanjoseA*, $Pr\{I \leq 5\} = 89.5\%$, and for *Gateway_egressA* and *Gateway_egressB*, $Pr\{I \leq 5\} = 70\%$ and 80%. These proportion numbers reveal that IP flows have very appealing packet-level locality.

Next we measure the locality of the URL trace. Similarly, *foreign request* is defined as the request between two same requests (that solicit the same content object), and *foreign interval* is the number of such foreign requests. Similar with foreign packets, the less foreign requests, the better locality of content objects. Results for the HTTP request trace are presented in Fig. 16. To compare with the results in Fig. 15, we let Fig. 16 have the same X-axis range as Fig. 15 does. We can learn that the foreign intervals less than $10^6$ only accounts for 14%, much smaller than that in Fig. 15. Actually, the number of foreign requests can reach $2.5 * 10^7$ (not shown in Fig. 16), and the range of $10^6 \sim 2.5 * 10^7$ accounts for the rest 86%. These results indicate that the same content requests can be separated by countless other requests, which means very poor content object-level locality.

*2) Cache hit rate:* According to the observations above,we infer that LFU may have better performance than LRU for the content objects. With our proposed method providing frequency statistics for LFU, Fig. 17 shows the cache hit rate of LFU and LRU under different cache sizes, and LFU always outperforms LRU by at most 6.80%.

*3) Unnecessary cache replacements:* Except for low cache hit rate, LRU can also lead to many *unnecessary cache replacements*. An unnecessary cache replacement means that an item in the cache is evicted out before it is requested again, which means it contributes nothing to the cache hit rate, but wastes a lot of resources. Intuitively, all the content requests would traverse the cache in LRU, leading to a high cost. In our experiments, the LRU incurs around 15,344,353 unnecessary cache replacements in a time window, while LFU only incurs as few as 6,763 ones.

## VI. Related Work

Though popularity-based caching policies have been widely studied, rare work has ever touched the problem of on-line popularity monitoring. Most of the previous works [3]–[13] just assume that the popularity statistics are readily available and make use of them directly. Jin *et al.* proposed present an on-line algorithm that captures popularity profile of Web objects on a Web proxy [13]. The popularity information will be used by a caching scheme. Since this popularity-capturing algorithm is running on a Web proxy and only deals with the requests and responses of the HTTP application, it does not have the speed requirement as high as routers do, which indiscriminately handles the content objects of all the applications. More importantly, [13] tries to only keep track of popularities of "popular content", in order to bound the memory space used to maintain such information. However, how to pick up the "popular content" now becomes a problem. The authors adopt a method to distinguish the popular contents based on the Zipf-like nature of popularity distribution, in which the parameters are, however, dependent on the prior knowledge of the trace. At last, the access frequency is calculated by a decay function, whose parameters are also trace-dependent.

A highly related field is the network measurement, such as flow statistics. Since our work is a Bloom filter-based popularity monitoring method, in the following, we will survey related works in two aspects: 1) network measurement; 2) researches that use Bloom filter in networking applications, which reveal the effectiveness of the Bloom filter.

### A. Network Measurement

Network Measure devotes to counting the flow sizes or volumes in routers, which resembles counting content popularities very much. In order to achieve high speed and small memory consumption, generally two categories of methods have been proposed: sampling and counter compressing. Static sampling [22]–[24] – sampling packets with the static rate – was firstly put forward. It exhibits good accuracy on elephant flows while brings huge relative error on mice flows. For this reason, many adaptive sampling methods have been proposed [14], [25], [26]. For example, [14] adopts adaptive sampling rate based on the value of current flow counter, assigning larger sampling rates to mice flows and smaller rates to elephant flows. Other researchers want to compress the flow counters so as to put them in a on-chip SRAM [15], [27]–[29]. [15] records a smaller value $c$ of the actual flow volume $n$ and maps $c$ to $n$ by a function. For a $q$-bit counter, SAC [27] divides it into two parts, an estimation part $A$ and an exponent part $mode$. The estimator of SAC is $\tilde{n} = A \cdot 2^{r \cdot mode}$, where $r$ is a parameter. In this way, SAC represents larger values using only $q$ bits. There is a rich literature that we cannot fully cover, but all these works are orthotropic to our work and can be combined together.

### B. Bloom Filters in Networking Applications

Bloom filters have been widely used in networking applications. An early example is its application in caching in distributed environments – the Summary Cache [30] system, which uses Bloom filters to distribute Web cache information. Summary Cache consists of cooperative proxies that store and exchange summary cache data structures, i.e., the Bloom filters. In order to synchronize the digest of caches at different proxies, proxies in the Summary Cache system periodically transfer the Bloom filters that represent the cache contents (URLs), since Bloom filters compactly summarize the cached contents. Summary Cache uses counting Bloom filters to maintain its locally cached dynamic contents when the dynamic contents change.

Bloom filters are also used in IP Traceback. Alex C. Snoeren *et al.* proposed a *Source Path Isolation Engine (SPIE)* to enable IP traceback, which means the ability to identify the source of a particular IP packet given a copy of that packet. SPIE adopts auditing techniques to support the traceback of individual packets, which is accomplished by computing and storing 32-bit packet digests rather than storing the packets themselves. Bloom filters are used to record the digests, in this way, SPIE reduces the storage requirements by several orders of magnitude over conventional log-based techniques.

Sarang Dharmapurikar *et al.* proposed an algorithm that uses Bloom filters for Longest Prefix Matching (LPM) [31]. This algorithm performs parallel queries on Bloom filters to determine address prefix membership in sets of prefixes sorted by prefix length. This work reveals that Bloom filter-based forwarding engines can achieve favorable performance compared to TCAMs. The idea of this algorithm is to have different regular Bloom filters for different lengths of address prefixes. These Bloom filters are implemented in hardware and updated by a route computation process. Routing updates are handled by Counting Bloom filters.

## VII. Conclusion

This paper proposes a high-speed and accurate on-line Bloom filter-based method to effectively capture the popularities of content objects passing by an NDN router. Each Bloom filter is endowed to represent a popularity range, and an content object is inserted into a Bloom filter if its popularity falls into that Bloom filter's range. In this way, our proposed method is efficient in memory consumption. Meanwhile, three optimization schemes are put forward to reduce memory accesses and therefore advance the monitoring speed. Both the fixed window and sliding window monitoring schemes are provided, the former is simple in design and implementation, while the latter updates popularity monitoring results more timely. Experimental results show that our method achieves a speed equivalent to 16.74 Gbps and only consumes around 32 MB memory. Using a synthetic trace and the sliding window scheme, this method even reaches a speed of 251.07 M/s (equivalent to 200.86 Gbps) due to reduced memory access latency. Moreover, a real trace-driven evaluation also shows that LFU policy achieves higher hit rate than LRU with much less unnecessary cache replacements.

## References

[1] L. Zhang, D. Estrin, V. Jacobson, and B. Zhang, "Named Data Networking (NDN) Project," in *Technical Report, NDN-0001*, 2010.

[2] V. Jacobson, D. K. Smetters, J. D. Thornton, M. Plass, N. Briggs, and R. Braynard, "Networking named content," in *Proc. of CoNEXT*, 2009.

[3] J. Li, H. Wu, B. Liu, J. Lu, Y. Wang, X. Wang, Y. Zhang, and L. Dong, "Popularity-driven coordinated caching in named data networking," in *Proceedings of the ACM/IEEE ANCS'12*, 2012, pp. 15–26.

[4] Y. Kim and I. Yeom, "Performance analysis of in-network caching for content-centric networking," *Computer Networks*, 2013.

[5] K. Cho, M. Lee, K. Park, T. T. Kwon, Y. Choi, and S. Pack, "Wave: Popularity-based and collaborative in-network caching for content-oriented networks," in *Proceeding of IEEE INFOCOM'13 NOMEN workshop*, 2012, pp. 316–321.

[6] H. Wu, J. Li, Y. Wang, and B. Liu, "Emc: The effective multi-path caching scheme for named data networking," in *Proceedings of IEEE ICCCN'13*, 2013.

[7] S. Wang, J. Bi, and J. Wu, "On performance of cache policy in information-centric networking," in *Proceeding of IEEE ICCCN'12*, 2012, pp. 1–7.

[8] S. Guo, H. Xie, and G. Shi, "Collaborative forwarding and caching in content centric networks," in *IFIP NETWORKING 2012, Part I, LNCS 7289*, 2012, pp. 41–55.

[9] D. Rossi and G. Rossini, "Caching performance of content centric networks under multi-path routing (and more)," *Relatório técnico, Telecom ParisTech*, 2011.

[10] S. Wang, J. Bi, Z. Li, X. Yang, and J. Wu, "Caching in information-centric networking," in *AsiaFI Future Internet Technology Workshop*, 2011.

[11] S.-J. Kang, S.-W. Lee, and Y.-B. Ko, "A recent popularity based dynamic cache management for content centric networking," in *Proceedings of Ubiquitous and Future Networks (ICUFN)*, 2012, pp. 219–224.

[12] G. Rossini and D. Rossi, "A dive into the caching performance of content centric networking," in *IEEE International Workshop on Computer Aided Modeling and Design of Communication Links and Networks (CAMAD)*, 2012, pp. 105–109.

[13] S. Jin and A. Bestavros, "Popularity-aware greedy dual-size web proxy caching algorithms," in *Proceedings of IEEE ICDCS'00*, 2000, pp. 254–261.

[14] C. Hu, S. Wang, J. Tian, B. Liu, Y. Cheng, and Y. Chen, "Accurate and efficient traffic monitoring using adaptive non-linear sampling method," in *In proceedings of IEEE INFOCOM'08*, 2008, pp. 26–30.

[15] C. Hu, B. Liu, H. Zhao, K. Chen, Y. Chen, C. Wu, and Y. Cheng, "Disco: Memory efficient and accurate flow statistics for network measurement," in *In proceedings of ICDCS'10*, pp. 665–674.

[16] M. Yu, A. Fabrikant, and J. Rexford, "Buffalo: bloom filter forwarding architecture for large organizations," in *Proceedings of ACM CoNEXT'09*, 2009, pp. 313–324.

[17] Y. Wang, T. PAN, Z. MI, H. DAI, X. GUO, T. ZHANG, B. LIU, and Q. DONG, "Namefilter: Achieving fast name lookup with low memory cost via applying two-stage bloom filters," in *Proceedings of IEEE INFOCOM'13 mini-conference*, 2013.

[18] S. Tarkoma, C. E. Rothenberg, and E. Lagerspetz, "Theory and practice of bloom filters for distributed systems," *IEEE Communications Surveys & Tutorials*, vol. 14, no. 1, pp. 131–155, 2012.

[19] D. V. Schuehler, J. Moscola, and J. W. Lockwood, "Architecture for a hardware-based, tcp/ip content-processing system," *IEEE Micro*, vol. 24, no. 1, pp. 62–69, 2004.

[20] R. Karedla, J. S. Love, and B. G. Wherry, "Caching strategies to improve disk system performance," *Computer*, vol. 27, no. 3, pp. 38–46, 1994.

[21] CAIDA Data. [Online]. Available: http://www.caida.org/data/overview/

[22] Cisco ios netflow data sheet. [Online]. Available: http://www.cisco.com/en/US/products/ps6601/products_ios_protocol_group_home.html

[23] K. C. Claffy, G. C. Polyzos, and H.-W. Braun, "Application of sampling methodologies to network traffic characterization," in *In proceedings of SIGCOMM'93*, 1993, pp. 194–203.

[24] K. Suh, Y. Guoy, J. Kurose, and D. Towsley, "Locating network monitors: Complexity, heuristics, and coverage," in *In proceedings of INFOCOM'05*, 2005, pp. 351–361.

[25] C. Estan, K. Keys, D. Moore, and G. Varghese, "Building a better netflow," in *ACM SIGCOMM Computer Communication Review*, vol. 34, no. 4, 2004, pp. 245–256.

[26] N. Duffield, C. Lund, and M. Thorup, "Learn more, sample less: control of volume and variance in network measurement," *Information Theory, IEEE Transactions on*, vol. 51, no. 5, pp. 1756–1775, 2005.

[27] R. Stanojevic, "Small active counters," in *in proceedings of IEEE INFOCOM'07*. IEEE, 2007, pp. 2153–2161.

[28] N. Hua, J. Xu, B. Lin, and H. Zhao, "Brick: A novel exact active statistics counter architecture," *IEEE/ACM Transactions on Networking*, vol. 19, no. 3, pp. 670–682, 2011.

[29] A. Kumar, J. Xu, and J. Wang, "Space-code bloom filter for efficient per-flow traffic measurement," *IEEE Journal on Selected Areas in Communications*, vol. 24, no. 12, pp. 2327–2339, 2006.

[30] L. Fan, P. Cao, J. Almeida, and A. Z. Broder, "Summary cache: a scalable wide-area web cache sharing protocol," *IEEE/ACM Transactions on Networking (TON)*, vol. 8, no. 3, pp. 281–293, 2000.

[31] S. Dharmapurikar, P. Krishnamurthy, and D. E. Taylor, "Longest prefix matching using bloom filters," in *Proceedings of the ACM SIGCOMM '03*, 2003, pp. 201–212.