An Efficient Caching Mechanism for Network-based URL Filtering by Multi-level Counting Bloom Filters

Yi-Hsuan Feng, Nen-Fu Huang, and Chia-Hsiang Chen
Department of Computer Science, National Tsing Hua University, Taiwan
{dr918302, nfhuang}@cs.nthu.edu.tw

Abstract—Network-based URL filtering (NUF) is one of the most widely used tools for detecting and stopping malicious and unwanted web traffic, like preventing children from sex. However, currently the conventional techniques still suffer from high bandwidth consumption due to millions of URL analysis requests to the network servers per day. In this paper, a model of NUF using a novel multi-level counting bloom filter (MLCBF) is proposed to address this issue. In the gateways of NUF, MLCBF is used to cache the analysis results from the network server to accelerate web traffic, alleviate the server load, and reduce bandwidth consumption of the entire NUF service. Analysis and trace-based experiments are employed to explore the properties of MLCBF and evaluate its performance in NUF. The results show that the proposed scheme typically eliminates at least 90% of memory requirements as compared to a general hashing table solution.

1. Introduction

In ISP, enterprise, and SOHO networks, URL filtering is widely used to prevent users to access unwanted and malicious web sites. Several service and device providers like Cisco, Websense, Surfcontrol, Blue Coat, and Gemtek provide network-based URL filtering (NUF) as a solution to classify, monitor, and control web traffic. Figure 1 illustrates the schematic procedure of NUF.

1. The end user browses a page on the web server, and the browser sends an HTTP request to the web server.
2. After the gateway receives HTTP request, it extracts the URL from the HTTP request. The URL is then sent to the network servers for analysis, while the HTTP request is forwarded to the web server simultaneously. For safety, each gateway needs to be authenticated before sending requests to the network server.
3. After the network server receives the analysis request, it checks its database to classify the web site represented by the URL. It then returns an integer representing the category of URL. Restated, the analysis result of a URL is just an integer (i.e., categorization ID) which represents a specific category. For example, “P2P” is represented by integer 2 and “online news” is 30. Notably, there are 50 to 90 categories usually, which are all represented by integers.
4. The HTTP response from the web server is queued for waiting for the decision by the gateway.
5. After getting the analysis result from the network server, the gateway sends or blocks the corresponding HTTP response according to the analysis result and management policy. For example, assume that although the P2P access is not allowed in an enterprise, the management policy allows for P2P access from an internal testing laboratory. Consider the classification result of a specific URL is a web site of P2P forum. If the source IP is in the range of the testing laboratory, the gateway sends the HTTP response to the end user. Otherwise, it sends a warning page to the end user.

NUF provides two important benefits over gateway-based URL filtering (GUF) which analyzes the URLs by simply comparing them with the local database in a gateway and updating the database continuously.

1. NUF can employ a cluster of powerful servers to quickly analyze extracted URLs using multiple complex techniques, including blacklist, web content inspection, and intelligent behavioral analysis. Moreover, the network servers are able to cooperate for detecting new malicious URLs more quickly because they collect and analyze URLs in a bird’s eye view. Such comprehensive capability can significantly increase the detection coverage and accuracy of URL classification.
2. Compared to GUF, the task of a gateway of NUF is only to extract URLs from HTTP requests, send them to the network server, and wait for the responses. This lowers the implementation complexity of a gateway engine of NUF, while the gateways no longer need to continually update local databases, thereby reducing administrative cost. Next, simplifying the engine allows the service to be used in resource-limited devices (e.g., mobile phones) that lack sufficient computing power but remain as a target of malicious web sites.

However, a service provider of NUF reported that they receive over 100 million requests for URL categorization per day. The bandwidth consumption amongst the gateways and network servers is therefore the key factor of the capacity and the maintenance cost of the service. Furthermore, waiting for the response from the network server indeed introduces processing delay to web traffic. In our preliminary tests, the average network latency from our laboratory to three network servers is between 100 to 500 ms. This motivates us to design an efficient model for NUF to reduce the bandwidth cost between the gateways and network.
servers, while accelerate the processing time in the gateways of NUF. The first idea is the local caching of URL analysis results in the gateways. The second idea is to use a hashing structure as the data representation of local caching.

In this paper, to minimize the resource requirements of NUF, a novel hashing data structure, called Multi-Level Counting Bloom Filter (MLCBF) is introduced to address this issue, and specifically we show that how to integrate MLCBF into the gateway of NUF as local caching. Based on the idea of using Counting Bloom filter (CBF) [1],[2] to store state machine [3], MLCBF is used to cache the URL classification results to minimize the memory requirements. Analysis and experiments are employed to explore the properties of the proposed structure and evaluate efficiency by several metrics. The results show that our method significantly reduces the memory requirements of a gateway of NUF as compared to a normal hashing table, and provides with small and constant latency time for the operations on MLCBF.

The remainder of this paper is organized as follows. We go over the related works in Sec. 2. In Sec. 3, we describe the architectural model of NUF of this work. In Sec. 4, we introduce the algorithm of MLCBF and its properties in details. Furthermore, we show that how to use MLCBF in NUF for local caching. The proposed method is evaluated by real URL access logs in Sec. 5. Finally, conclusions and future works are presented in Sec. 6.

2. Related Works

A wide range of techniques have been proposed for enhancing web applications, like web access security, URL forwarding and lookup engine [4], and web proxy caching [1]. Web content filtering is one of popular approaches to provide web access security. The key function of this method is the classification on web pages. In [5], it provides a hierarchical structure for classifying a large collection of web content. In the works of [6],[7],[8], different machine-learning-based methods are used to perform web content filtering. Although those methods provide accurate filtering results, it seems to take too much time to process each web page by multiple intelligent techniques. In contrast, NUF and GUF are more appropriate for ISP, enterprise, and SOHO networks.

URL blacklist is another common method to implement web filtering engine. Allowing HTTP access or not depends on comparing the URL of an HTTP request to the URLs in the blacklist. In [9], URL filtering is performed based on caching mechanism. In [10], a Wu-Manber-like matching algorithm with a support of CRC32 is used in a URL filtering system. In [11], two functions are proposed for hashing the signatures of URLs which can get efficient URL lookup performance. In sum, similar to GUF, the above works store blacklist in the local filtering engines, and they therefore have to update the databases periodically.

To the best of our knowledge, our work is the first literature to improve NUF by using hashing data structures as local caching. In this paper, three hashing structures are evaluated, including MLCBF, d-left CBF (DLCBF) [3],[12], and CBF. DLCBF is a simple and practical alternative to CBF [13]. Compared to CBF, DLCBF saves a factor of two at least on memory for the same false positive rate. Notably, MLCBF can be viewed as a modification of DLCBF. Motivated by multi-level hash table [14],[15],[16] as an improvement of d-left hashing table [17], in this work, we introduce skewness to the basic construction of DLCBF to improve the run-time false positive rate and storage utilization, and to retain its benefits of simple construction, and small filter size as compared to CBF.

3. Design of Network-based URL Filtering

In this paper, NUF uses precise message as communication protocol between a gateway and network server. Restated, the format of an analysis request from a gateway to the network server is <URL>, while an analysis result is <URL, integer>. The integer indicates the category of the corresponding URL. According to the trace-based evaluation presented in Sec. 5, the average size of URLs is about 55 to 59 bytes.

To extract URLs from HTTP requests (or other applications, e.g., FTP, mail, and P2P), a lightweight agent runs on gateways that identifies URLs and sends them to the network server for analysis.

3.1 Local Caching in NUF Gateway

Once a classification result has been generated for a URL, it can be stored in a local cache on the gateway. This indicates that once a URL has been analyzed, subsequent accesses to that URL can be determined locally. This strategy can accelerate web traffic, alleviate the load of network servers, and reduce bandwidth costs of the entire NUF service.

Moreover, once a single gateway has accessed an unclassified URL and sent it to the network server for analysis, any subsequent access of the same URL by other gateways can leverage the existing analysis result by sharing the result amongst the network servers. Cached analysis result stored in the network servers may also periodically be pushed to the gateways to update URL classification and be invalidated if necessary.

4. Multi-level Counting Bloom Filter (MLCBF)

The strategy of using hashing structure as data representation is widely used in various network applications to minimize resource requirements, like web caching proxy [1], P2P resource routing [18], distributed metadata management [19], and other applications [2]. In this section, a novel hashing structure, called as MLCBF, is introduced to be integrated into the gateway of NUF service.
Algorithm 1 Pseudo-code for insertion and search in MLCBF

Function MLCBF-INSERT(key)
1: if MLCBF-SEARCH(key) then
2: for i = 1 to D do
3: pos ← LV[i][h_i(key) mod BN_i]
4: for j = 1 to H do
5: if BE[pos, LB][j] = 0 then
6: BE[pos, LB][j] ← fingerprint + h_j(key)
7: CC ← BE[pos, LB][j].CC + 1
8: BE[pos, LB][j] ← 1
9: return 1
else return 0

Function MLCBF-SEARCH(key)
10: for i = 1 to D do
11: pos ← LV[i][h_i(key) mod BN_i]
12: if BE[pos, LB] = 0 then
13: for j = 1 to H do
14: if BE[pos, LB][j] = 1 then
15: if BE[pos, LB][j] . fingerprint ≠ h_j(key) then
16: j ← j + 1
17: else return 1
18: else j ← j + 1
19: return 0

4.1 Filter Structure and Insertion Algorithms

Suppose that we have a set \( S = \{x_1, x_2, ..., x_n\} \) (i.e., \( S \) contains \( n \) items or keys over universe \( U \)) that is changing by item insertion and deletion over time. With the same functionalities of CBF, Multi-Level Counting Bloom Filter (MLCBF) represents \( S \) by allowing for item insertion and deletion, while MLCBF provides membership and multiplicity queries on \( S \). A query of \( x \in S \) on MLCBF always produces a positive answer, while a query of \( y \notin S \) can yield a small false positive rate.

In Fig. 2, it schematically depicts the construction of MLCBF. The basic idea of MLCBF is to store most of items in the largest (i.e., first) level. All operations of MLCBF can then probably be completed in the 1st level.

To hold \( n \) items of \( S \), MLCBF is a hierarchy of \( D \) levels \( (LV_1, ..., LV_D; D \geq 2) \) with \( D \) independent and uniform hash functions \( h_1, ..., h_D \), where each level comprises different bucket numbers \( (BN_1, ..., BN_D) \). To insert, query, or delete an item, MLCBF contains \( D \) possible buckets. The level sizes are decreasing linearly by a fixed decreasing ratio \( R \) \((R < 1)\). Notably, the level \( LV_i \) holds \( BN_i = \lfloor n/(H \cdot R_0) \rfloor \) bucket elements \( (BEs) \), and \( LV_i \) holds \( BN_i = \lfloor BN_{i-1} \cdot R \rfloor \) buckets for the level number \( i \geq 2 \).

Each \( BE \) consists of \( H \) cells \((H \geq 1)\) and a load bitmap \((LB)\) of \( H \) bits to record the number and location of active or in-use cells. Each cell holds a cell counter (denoted as \( CC \), C-bit) and a fingerprint \((F\)-bit) from a hash function \( h_F(key) \). For a situation in which the corresponding \( LB \) bit is not set, a cell is identified as empty or non-active, and the cell access for a query and deletion can be avoided. Denoting the total bucket number as \( BN_n = n/H \) gives a memory size bounded by \( n \cdot (F+C)+BN-H \) bits.

To insert an item \( x \), as shown in Algorithm 1, if \( h_F(x) \) does not exist in MLCBF, MLCBF simply places the item \( x \) to an empty cell of \( BN_i \) indexed by \( (h_i(x) \mod BN_i) \) with the smallest \( i \). An item is only inserted into the level \( LV_{i+1} \) when the hashed bucket in \( LV_i \) is full. MLCBF thus allows bucket overflows in \( LV_i, 1 \leq i < D \). The insertion probing is stopped until an empty cell or an overflow in \( LV_D \). If \( h_F(x) \) already exists in any \( BE \) during an insertion, the corresponding cell counter \( CC \) is simply increased.

Fig. 2. An example of MLCBF construction. The settings of MLCBF are \((D,H) = (4,8), F=20-bit, R=0.5, and 8 cells per bucket.\)

Fig. 3. Average maximum achievable loads of 10k-trial simulation with different \((D,H)\) settings. Total cell number is 10k.

To answer \( \"y \in S\?\" \), one checks whether the fingerprint \( h_F(y) \) is found in \( D \) associated \( BEs \) by MLCBF-SEARCH(key) of Algorithm 1. If not, \( y \notin S \). All \( D \cdot H \) probes are thus required in the worst case scenario and lookup complexity is \( O(1) \). In MLCBF-SEARCH(key), because a search naturally accesses the buckets in the same order as insertion and the wanted item is probably in the level \( LV_1 \), it starts from \( LV_1 \) and continues until \( LV_D \).

In a deletion, when the inserted item is found through MLCBF-SEARCH(key), the cell counter \( CC \) is just decremented and the \( LB \) is set to 0 if the \( CC \) becomes 0.

4.2 Properties of MLCBF

Because DLCBF uses \( D \) “equal-sized” \((3,12)\) subtables and \( H \) cells in each bucket, both MLCBF and DLCBF can be extended by extending hash number \( D \) and changing \( H \). This feature is unified by using a parameter pair \((D,H)\) to compare MLCBF and DLCBF in terms of performance and tradeoffs. Hereinafter, MLCBF\((D,H)\) denotes the setting \((D \text{ levels, } H \text{ cells per bucket})\) of an MLCBF. Without a loss of generality, MLCBF and DLCBF are referred to hereinafter as multi-level fingerprint-based filters (MFFs) to highlight how their construction concept differs from that of Bloom filter-based filters like CBF.

Storage utilization or load of an MFF is defined as the ratio between the number of inserted items and total cell number. Load distribution of the level \( LV_i \) (called as \( LD_i \)) denotes the ratio between the inserted item number in \( LV_i \) and total inserted item number. Load factor \( \alpha \) of an MFF measures the expected item number per bucket, while \( \alpha_i \) denotes the load factor of \( LV_i \). Finally, given \( D, H, \) and a total cell number, consider new items are continuously inserted to an MFF from scratch until an overflow. Maximum achievable load, denoted as Load\(_{\text{max}}\), of an MFF is defined...
as the ratio between the number of total inserted items before an overflow and the total cell number.

A. Maximum Achievable Loads of MLCBF

The $Load^{\text{max}}$ of MFFs are investigated in simulation by a variety of choices ($2 \leq D \leq 8$, $2 \leq H \leq 8$). Fig. 3 summarizes those results (the simulation setup is described in Sec. 5). The next subsection discusses the expected $Load^{\text{max}}$ of MLCBF.

First, except for $D=2$, MLCBF has higher $Load^{\text{max}}$ than those of DLCBF by the same $(D,H)$. Second, Fig. 3 indicates an MFF needs $n/Load^{\text{max}}$ cells at least to store $n$ items. For example, to support $10^5$ items, MLCBF and DLCBF by $(4,4)$ need at least 115,207 and 117,994 cells, respectively. For MLCBF, $(4,8)$ is almost as space-efficient ($Load^{\text{max}}=93.65\%$) as $(8,4)$ ($Load^{\text{max}}=97.26\%$) but with less hash functions. This suggests a smaller latency in a platform without hash acceleration hardware.

B. Storage Utilization and Load Distribution

The proposed MLCBF scheme is analyzed next. The properties of DLCBF are investigated based on the experimental simulations. For simplicity, assume that the probability of fingerprint collisions is ignored; in addition, the analysis does not consider item deletion. The first task involves computing the expected storage utilization, $LD_i$, and the load factor $\alpha_i$ of the level $LV_i$ of MLCBF.

If $n$ items are inserted into a hash table with separate chaining by a uniform hash function, the fraction with load $k$ is $1/k! (e^{-\mu \mu})$ as $n$ goes to infinity and the average load is $\mu$. Most of the analysis on normal hashing is based on the above Poisson distribution.

In MLCBF, given the number of items $n$ to be inserted into a level with $m$ bucket elements $(BE)$, the corresponding expected number of items lying in all $BE$s which have exactly the load of $k$ mapped into them is then $km \binom{m}{k} (m-1)^{n-k} / m^n$.

To calculate the load distribution of MLCBF, denote $n_{LV_i}^{\text{overflow}}$ as the expected number of items left from the level $LV_i$ to be inserted to $LV_{i+1}$, $n_{LV_i}^{\text{success}}$ as the expected number of items inserted to $LV_i$ successfully. Then,

$$n_{LV_i}^{\text{overflow}} = n_{LV_i}^{\text{insert}} = \sum_{j=H+1}^{n} (j - H) m \binom{m}{j} (m-1)^{n-j} / m^n$$

Applying Eq. (1) recursively, starting from $LV_i$ with $n_{LV_i}^{\text{insert}} = n$ and $n_{LV_i}^{\text{success}} = n_{LV_i}^{\text{insert}} - n_{LV_i}^{\text{overflow}}$, $i = 1, ..., D$, allow us to estimate $\alpha_i$ as $n_{LV_i}^{\text{success}} / BN_i$, storage utilization of $LV_i$ as $n_{LV_i}^{\text{success}} / (BN_i \cdot H)$, and $LD_i$ as $n_{LV_i}^{\text{success}} / n$. For instance, according to Eq. (1), to insert 15k items into an MLCBF$(4,8)$ containing 20k cells, $n_{LV_i}^{\text{success}}$ of $LV_i$, to $LV_D$ are 10,336, 4,237, 427, and 0, which are very close to the 10k-trial simulation result: 10,328, 4,237, 432, and 0 on average. Finally, by a given total cell number, $Load^{\text{max}}$ of MLCBF$(D,H)$ can be estimated by increasing the load till $n_{LV_D}^{\text{overflow}} > 0$.

C. False Positive Rates

For an MFF, a false positive occurs if and only if for a query of $y \notin S$, $x \in S$ exists with $h_y(x) = h_x(y)$ in an associated bucket element. Restated, the fraction that this event occurs, called as false positive rate (denoted as $P_{FP}$), is calculated from the likelihood that one of all possible cells produces the same fingerprint for $y \notin S$. Thus, despite increasing $Load^{\text{max}}$, a higher $D$ or $H$ increases the probability of hash collisions and the resulting $P_{FP}$.

The rate $P_{FP}$ of an MFF$(D,H)$ can be upper bounded by $D \cdot H \cdot 2^{-F}$. Moreover, $P_{FP}$ of an MFF can be expressed as

$$P_{FP} = \sum_{l=1}^{D} \alpha_l \cdot 2^{-F}$$

Due to the skewness and insertion strategy of MLCBF, $\sum \alpha_l$ of MLCBF at a given load and $F$-bit is smaller than that of DLCBF; even their load factors $\alpha$ are identical. Thus, by Eq. (2) and simulation, Fig. 4 reveals that MLCBF has a lower $P_{FP}$ than DLCBF does. Next, MFFs use less memory than CBF does; normally saving a factor of two, at least for the same $P_{FP}$.

4.3 Using MLCBF as Local Cache of NUF Gateway

Based on the idea of using CBF to store state machine [3], MLCBF is used to cache the URL classification results. Restated, the cell counter $CC$ of MLCBF is used to store the classified integer of $h_y(\text{URL})$ directly, not used in the way of its original design; as a counter of a specific $h_y(\text{key})$.

Fig. 5 illustrates the proposed model of using MLCBFs in a client engine of NUF. Initially, when a URL is extracted by the client engine from HTTP request, it is searched in the local MLCBF by $h_y(\text{URL})$. The lookup complexity is $O(1)$ as described in Sec. 4.1. If the URL is not found, the client engine sends the URL as an analysis request to the network server for classification. The analysis response is $<\text{URL, integer}>$. After the client engine receives the result, it inserts $<h_y(\text{URL}),\text{integer}>$ into the local MLCBF. Next time when the same URL gets into the engine, the classification result...
can be found locally.

To store key-and-state information, MLCBF is used to support `insert(key,state)`, `modify(key,state)`, `lookup(key)`, and `delete(key)` operations. Notably, $2^{-I}$ is equal to the highest integer number. The notion simply involves storing the fingerprint $h_f(x)$ of a key $x$ and its classification integer into MLCBF. For instance, if the category for a URL is classified as 7 by the network server, the cell of local MLCBF in a gateway is obtained by $h_f(URL) \mod BN_i$ and $h_f(URL)$, while the cell counter CC is updated to 7.

In practice, using randomization to represent the URLs and their states may introduce some errors. A false positive (FP) refers to the lookup on a non-inserted flow returning a valid state. A false negative (FN) refers to no valid state for an active flow. Moreover, an inaccurate state (IS) refers to an incorrect returned state. In practice, these errors are not only introduced by hash functions (i.e., $P_{FP}$), but also likely by bucket/counter overflows, fingerprint collisions during dynamic operations, and early recycling of active URLs based on memory management.

### 4.4 Using MLCBF as URL Blacklist and White-List

Besides URL categorization, some gateway devices also support the functionalities of URL blacklist or white-list. If a URL is found in blacklist or white list, it will be blocked or sent to the web server directly without any more check, regardless of the categorization of URL.

Notably, MFFs and CBF are designed to answer a query of “$y \in S$?”. The blacklist and white list can be therefore implemented easily by MLCBF by its membership operations, which updates a state of $h_f(URL)$ when it changes from 0 to 1 (i.e., $y \in S$) or 1 to 0 (i.e., $y \notin S$). Here, `insert(key)`, `lookup(key)`, and `delete(key)` are supported using MLCBF in URL blacklist/white-list, in which FP rate (i.e., $P_{FP}$) is of priority concern. Due to the lack of paper space, we only present the evaluation results of NUF in the next section.

### 5. Evaluations

The experiments were performed on a machine (Intel Pentium-4 2.0 GHz, 512-kB L2 cache, and 1024 MBs RAM) as a NUF gateway. By the design of Fig. 5, the functions of URL filtering are implemented as a kernel module in Linux 2.6.30. Notably, the simulation presented in Sec. 4 to study the filter properties is performed by the prototype too. The random keys in all tests are read from Linux `/dev/urandom`. A general hash table by separate chaining is used to store precise keys (URLs), URL categorization ID, and metadata. In this section, we compare the performance of MLCBF, DLCBF, and CBF with the solution that uses a normal hash table as local caching to store the analysis results of URLs.

For the settings of MFFs, $(D,H)$ is set as $(4,8)$ (Load of MLCBF(4,8) = 93.65%, $R$ is 0.5 for MLCBF, and $F$ is 20-bit. For CBF, the number of hash functions is 4, and load factor of CBF (i.e., $m=n$, the ratio between the number of filter slots and the number of inserted items in maximum) is 10. The false positive rate is about 1.2% in theory. An SHA-1 implementation with modification is used for $D$ hash functions, fingerprint, and signatures of CBF.

![Fig. 5. The proposed model of network-based URL filtering. The basic ideas are to use caching for analysis results and hashing structure as data representation.](image)

The FN and IS rates are verified by comparing the precise key and integer stored in the hash table with the corresponding data in the filters.

Table I shows the evaluation results, while the memory reduction ratios achieved by MLCBF and CBF are compared to the solution by storing URLs and integers in a hash table. The memory reduction rates of MLCBF are as high as 90.9% at least. By $F=20$-bit and $(4,8)$, the range of FP rates of MFFs for six collections is from 0.0003% to 0.0012%. The IS rates are all smaller than 0.0014%. The FP and IS rates of CBF at a load factor of 10 are 0.637% and 14.96%. By a load factor of 40 for CBF, they are 0.0012% and 1.5% at a cost of 4 times memory requirements. The FN rates are zeros, because of no URL removal and no overflow. With an $F$ of 24 bits, the FP and IS rates of MLCBF and DLCBF are 0.0001% and 0.78% for many URLs. The filter size is lower than 21 MBs for rtp (2007/1/10) at 80% load of the filters.

The resource requirements and false rates observed in the tests are likely to be reduced in practice, because it is expected that a gateway would not contain so many URLs. For a local caching, it is probably that only a set of the most frequently accessed URLs would be stored.

Finally, by `rdtsc` instruction [21], Fig. 6 reports the measured CPU cycles of `insert()` and `lookup()` implementations of MFFs and CBF in the experimental tests. First, the cycles increase as the load increases for an MFF and the performance of CBF is proportional to the number of hash functions. For any given load, the insertion cycles of MLCBF are lower than those of DLCBF. Moreover, MLCBF provides a considerably better performance than other methods in
TABLE I: SIMULATION RESULTS OF NETWORK-BASED URL FILTERING BY REAL URL COLLECTIONS FROM NLANR [20].

<table>
<thead>
<tr>
<th>Access list name and time</th>
<th>Total access logs</th>
<th>Total unique URLs</th>
<th>Mean URL size (bytes)</th>
<th>URL string size (kBs)</th>
<th>DLCBF(4,8), $f=20$-bit</th>
<th>MLCBF(4,8), $f=20$-bit</th>
<th>CBF</th>
</tr>
</thead>
<tbody>
<tr>
<td>bo2(2007/01/09)</td>
<td>2,341,173</td>
<td>144,852</td>
<td>57</td>
<td>1,353</td>
<td>1,271</td>
<td>90.6%</td>
<td>1,375</td>
</tr>
<tr>
<td>bo2(2007/01/10)</td>
<td>2,073,704</td>
<td>133,420</td>
<td>56</td>
<td>1,138</td>
<td>1,071</td>
<td>90.4%</td>
<td>1,175</td>
</tr>
<tr>
<td>rtp(2007/01/09)</td>
<td>3,176,750</td>
<td>1,655,579</td>
<td>59</td>
<td>184,066</td>
<td>16,752</td>
<td>90.9%</td>
<td>17,933</td>
</tr>
<tr>
<td>rtp(2007/01/10)</td>
<td>2,986,122</td>
<td>1,501,494</td>
<td>58</td>
<td>169,990</td>
<td>15,747</td>
<td>90.7%</td>
<td>17,201</td>
</tr>
<tr>
<td>sd(2007/01/09)</td>
<td>1,426,885</td>
<td>879,114</td>
<td>55</td>
<td>77,341</td>
<td>7,534</td>
<td>90.3%</td>
<td>7,790</td>
</tr>
<tr>
<td>sd(2007/01/10)</td>
<td>1,497,891</td>
<td>933,756</td>
<td>55</td>
<td>81,420</td>
<td>7,899</td>
<td>90.3%</td>
<td>8,320</td>
</tr>
</tbody>
</table>

Fig. 6. Average insertion and lookup times (in CPU cycles) of CBF, DLCBF, and MLCBF at different loads.

lookups. For instance, MLCBF yields an improvement of 2 to 7 times for CBF on lookups. For all methods, their cycle times of modify() and delete() are similar to those of lookup(). Notably, the average times of the insertions and lookups on our implementation of hash table take 3,156 and 1,759 cycles at 80% load. By contrast, they are all smaller than 2,500 and 1,125 cycles for MLCBF, DLCBF, and CBF. Although heavily dependent on implementation, the above measurements on CPU provide a practical look on feasibility of using MLCBF as a local cache.

6. Conclusions and Future Works

The focus of this work is to reduce the resource requirements of network-based URL filtering (NUF) by using the hashing data representation for caching of URL analysis results. This work presents a new compact data structure, called as Multi-Level Counting Bloom Filter (MLCBF), to use the effect of skewness and insertion distribution over the levels of MLCBF for storing a large number of URLs. Specifically, we present that how MLCBF can be integrated into a NUF gateway.

The proposed methods have been implemented by Linux as a real platform. By real URL access logs, trace-based simulation reveals that MLCBF reduces memory requirements of a local cache in NUF gateway typically by 90.9%, as well as provides low operation latency.

As a future work, we are studying how to utilize web caching proxy squid [22] and SquidGuard [23] to implement the network servers of our real tested. Next, as a URL lookup method, the method of using MLCBF as URL cache should be compared to other methods like hash chain [4] and ufdbGuard [24]. Notably, in [4], it reports the memory requirement of hash chain is larger than CBF. Thus, we believe that MLCBF outperforms hash chain at least in the area of memory costs.

References