Implementing Scalable URL Matching with Small Memory Footprint

By

Daniel Krauthgamer

Final project, submitted in partial fulfillment of the requirements for the M.Sc. degree, School of Computer Science
The Interdisciplinary Center, Herzliya

December 2015
This work was carried out under the supervision of Prof. Anat Bremler from the Efi Arazi School of Computer Science, The Interdisciplinary Center, Herzliya
1 Acknowledgments

I would like to thank my advisor Prof. Anat Bremler-Bar for her guidance throughout my final project. Her professional help and guidance were vital for the success of this project. I would also like to thank Mr. Yotam Harcol for his technical help. His wise advices and experience helped me facing difficult implementation conflicts with efficiency and productivity.

With the work on this project I was introduced to new fields in computer science. I learned and expanded my knowledge about compression techniques. I got introduced with the fields of pattern matching algorithms and heavy hitters algorithms.

In addition, working on this project had sharpened my knowledge about memory efficiency, optimizing and measuring memory management.
2 Abstract

URL matching lies at the core of many networking applications and Information Centric Networking architectures. For example, URL matching is extensively used by Layer 7 switches, ICN/NDN routers, load balancers and security devices (1), (2), (3), (4). Modern URL matching is done by maintaining a rich database that consists often of millions of URLs and consuming a large amount of memory.

Reducing the URL matching algorithm’s memory footprint enables these systems to handle a larger sets of URLs. The paper (5) introduces a generic framework for accurate URL matching that aims to reduce the overall memory footprint, while still having low matching latency.

The framework’s input is set of URLs, and the output: a DFA like data structure that encodes any URL to a compressed form. The encoded form of the URL then can be used as a key to a database such as hashtable. Therefore the DFA like data structure is a dictionary-based compression method that compresses the database by 60%, while having only a slight overhead in execution time.

The framework is very flexible and it allows hot-updates, cloud-based deployments. Moreover it can deal with strings that are not URLs.
תקציר

השוואת URL נמצאת בסיסן של רבות מאפליקציות רשת ומארכיטקטורות רשת. לדוגמה, switches, ICN/NDN routers, load balancers והשוואת URLيته של URL במערכות מודרניות מבוצע ע"י שמירתם בחונים שונים וניהולם המכלי של مليים מילים...

השוואת URL והחיפוש של URLים במערכות מודרניות מבוצע ע"י שמירתם בתוך מסדי נתונים גדולים המכילים מיליוני כתובות URL ודורשים זיכרון רב כדי שמירה.

הקטנת חתימת הזיכרון של אלגוריתמי החיפוש וההצהמה של URL ממאשר להגדיל את כמות הקונטוט המאוחסנת בחונים נוספים, וחיטוב ההמפות המאradorות URL במערכת להקטין את כמות הزهرות הדורשה כדי לבעז השיוואות ב-E URL שוויםuly והשוואת URL.

הקלט של התשתית הוא רשימת URLים, והפלט היא מבנה נתונים הדומה לאוטומט סופי דטרמיניסטי (DFA) המאפרת קידוד של כל URL לזרוע דומה. הקוד הדחוס הוא ייחודי ומאפרת השיוואה电视ית (DFA) ביו מoomla שלップ שומני בלתי апрיקטישÉtat הקונטקט המ諮詢, הקוד הדחוס יוני שלップ תמקחת מבנה הנתונים המ삮 (htable) התשתית היא למשמע שנל שיש דחוס מבוסס מילו המאפרפר התשקול של גודל החלל של המפתות בכ-60% עם עלות השתייה המוגדרת ומheticת המהנהנו / הקונטקט המ Elder מבוסס הנתונים.

התשתית מאפרפרתเกษตร גומיש ביוור כון: דחוס של URL משלו ויי בקטל המקרה וקריסת URL מבותר. בןף הינו גמ מאפרפרת דחוס של כל מהנהנו מום וגוו את.
4 Goals

Implementing and analyzing the framework as described in the paper (5) in C++ to be memory efficient and fast while reusing previous work of Anat’s team: Compressed AC implementation (6), and Double Heavy Hitters (7).

The implementation will be written with strong emphasis for memory efficiency both theoretical and practical (as described later) while keeping the code with real-time properties such as minimal function calls and no dynamic memory allocations past the setup stage.

The code will be published as open-source allowing other research groups and industries to challenge our own work with our results and hence is designed to be easily used both as a black box algorithm or to be modified in order to be improved.

This works goal is to implement to the best practice the framework in the article "Scalable URL Matching with Small Memory Footprint" by A. Bremler, D. Hay, D. Krauthgamer and S.T. David (5). This paper submission describes the work and results produced during this project. Therefore, with coordination with Advisor, I used the original paper to describe the algorithm and its background.
## Contents

1. **Acknowledgments** ................................................................. 3
2. **Abstract** .................................................................................. 4
3. **goals** ....................................................................................... 5
4. **Background** ............................................................................... 9
   6.1 **Aho–Corasick pattern matching algorithm** ........................................ 9
   6.2 **Double Heavy Hitters algorithm** ..................................................... 10
   6.3 **Slabs based memory management** .................................................. 11
5. **Related Work** ........................................................................... 12
6. **Algorithm Overview** ................................................................. 13
   8.1 **The Online Phase** .................................................................... 13
   8.2 **The Offline Phase** ................................................................... 15
   8.2.1 **Heavy Hitters Algorithm** ......................................................... 15
   8.2.2 **Anchor Selection** .............................................................. 16
   8.2.3 **Deterministic URL database compression and Huffman code creation** 18
   8.3 **Hot-updates Support** ............................................................ 18
   8.4 **Longest Prefix Match** ............................................................ 19
7. **Implementation Overview** ....................................................... 20
   9.1 **Considerations** .................................................................... 20
   9.2 **General** .............................................................................. 20
   9.3 **Longest Prefix Match** ............................................................ 20
   9.4 **Software design** ................................................................... 20
   9.5 **Memory efficiency** ............................................................... 22
   9.5.1 **Using dynamic memory allocation efficiently** ............................ 22
   9.5.2 **Using efficient alignment (and avoid padding)** ......................... 22
   9.6 **Runtime Optimizations** ......................................................... 23
   9.6.1 **Reduced function calls in hot-path** .......................................... 23
   9.6.2 **Fast Huffman encoding** ...................................................... 23
   9.7 **The Tester** .......................................................................... 24
   9.7.1 **Tester CMDs** ................................................................. 24
   9.7.2 **Tester flags** ................................................................. 24
Evaluation and Results ................................................................. 25
  10.1 The effect of the number of anchors on performance .................. 25
  10.2 The effect of number of anchors on memory footprint.................. 27
  10.3 Real hashtable memory footprint .............................................. 29
  10.4 Evaluate realtime behavior ..................................................... 30
  10.5 The effect of Double Heavy Hitters parameters on performance ....... 31
Conclusions .................................................................................... 33
References ...................................................................................... 34
6 Background

6.1 Aho – Corasick pattern matching algorithm

Aho-Corasick is a string matching algorithm (8) based on a dictionary where the matching algorithm locates elements of a finite set of strings within an input text. Aho-Corasick constructs a deterministic finite automaton (DFA) from the dictionary of strings to be matched. Therefore, it can find target string in text of size \( n \) with running time complexity of \( O(n) \).

In Figure 1 we can see the DFA constructed by Aho-Corasick for the strings he, his, she and hers. The algorithm starts at state 0. Given an input byte \( b \), the algorithm looks for a valid transition (solid edge). If there is one, the algorithm moves to the next node. Otherwise, the algorithm uses the failure transition (dashed edge) to find a node from which a valid transition will be found, and so forth.

![Figure 1 - Aho Corasick DFA](image)
6.2 Double Heavy Hitters algorithm

Heavy Hitters (HH) algorithm is a member of streaming algorithms. The HH is used to find the most frequent (popular) elements in a data stream. The problem of finding the heavy hitters or frequent items in a stream of data is defined as follows: given a sequence of $N$ values $a = \langle a_1, \ldots, a_N \rangle$, using a constant amount of space, find $n_i$, each having a frequency (the number of times it appears in $a$). In their paper, "Automated Signature for High Volume Attacks" (7), Y. Afek, A. Bremler-Barr and S. Landau-Feibish describe a new algorithm called Double Heavy Hitters.

The idea is to define $k$-grams to be a string of chars of length exactly $k$. Then using two independent heavy hitters components, $HH_1$ and $HH_2$ they do as follows:

1. $HH_1$ finds $k$-grams that appear frequently, i.e., that are heavy hitters.
2. $HH_2$ finds varying length strings that occur frequently in the input (which are combinations of the strings found in step 1).

The input to the $DHH$ algorithm is a sequence of $n_p$ packets, a constant $k$ which will determine the size of the $k$-grams used and a constant $r$ which is a ratio that we will be soon explained. Conceptually, the process works as follows: the algorithm traverses the packets one by one. For each index in the packet, a $k$-gram is formed by taking the $k$ characters starting from that index. These $k$-grams are given as an input to $HH_1$. To form the varying length strings which are the input to $HH_2$, while $HH_1$ processes the $k$-grams, the algorithm seeks to find the longest run of consecutive $k$-grams such that: 1) they are all already in $HH_1$ (i.e., at this stage they are heavy hitters), 2) they have similar counters. The objective is that combining two $k$-grams should occur only if they should be part of the same signature. Without this ratio, if some $k$-gram appears very frequently, but the character that usually follows this $k$-gram is inconsistent, then the preferred signature should not combine this $k$-gram with the one that follows it. Specifically, counters of two consecutive $k$-grams maintain a ratio of $r$.

This process of creating a string from consecutive $k$-grams, is a key factor in substantially reducing the substring pollution (If a string $s$, $|s| > k$ appears many times in the input text, then all the $k$-grams which are substrings of $s$ show up as heavy hitters and are output by the heavy hitters algorithm) in the output. For each such consecutive sequence, the process creates a single input to $HH_2$, which is a varying length sequence of values that has been naturally filtered by a preceding heavy hitters procedure, $HH_1$. 
6.3  Slabs based memory management

Modern operating system (such as Linux) often need to allocate temporary storage for non-persistent structures and objects, arrays, structures. These objects that are non-uniform in size and might be allocated and released many times during the life of the program. The best allocation scheme to use is one that is optimized for allocating and releasing pages in multiples of the hardware page size. However, for the small transient objects often required by the programs, these page allocation routines are horribly inefficient, leaving the each program for optimizing their own memory usage.

Therefore a slabs based memory management is implemented and used in the operating system side that is invisible to the program or the programmer. By this method the operating system pre-allocates hardware optimized pages of memory and breaks them down into fragments that are multiple of 2 (often starting with 32 to reduce the bitmap size\(^1\)): 32 Byte, 64 byte, .. called slab.

![Figure 2 - Memory based Slab system](image)

Keeping in memory the slab in bitmaps according to their sizes enables the operating system to find / allocate / free each slab fast and efficient.

The major con of this system is that memory allocation (especially small allocations) often suffer from worse footprint, e.g allocating 10 bytes in Unix based system will take 32 bytes of the free memory. Memory hogs applications often optimize this behavior by replacing the Kernel memory management or by pre-allocation large buffers and allocated them internally. Other method often used is to optimize a couple of same size objects into pre-allocated slabs.

\(^1\) Bitmap is used to keep the fragments in memory that are taken. When each bit referencing 32 bytes, the size of the bitmap is reduced by a factor of \(2^5\) often making is smaller and efficient to keep in memory.
7 Related Work

Performing fast URL matching has recently received extensive attention, mostly focusing on the URL matching time and leaving the storage requirement as a secondary goal. This algorithm focuses on reducing the memory footprint of the URL database by designing the tailored-made compression method. Once the database is compressed, it can be used in conjunction with any of the previously suggested URL matching techniques.

The related work and the related work relevance and analysis is already discussed in the original paper (5) in the "Related Work" section.
8 Algorithm Overview

The algorithm described in the paper (5) is a dictionary compression algorithm. More specifically the problem that the algorithm is solving is how to encode a URL into a small sized code while being lossless and comparable so it can be used as key to store the URLs categorized (such as in database or a hashtable).

The framework's input is a set $S$ of URLs and its output is a specific algorithm (built from a DFA, and a Huffman code) that can encode any URL and optimized to URLs from $S$.

Using the algorithm based on two phases: Building a compression dictionary of common pattern (anchors) which we will address as the "Offline Phase" and creating a database key by compressing the URL which we will address as the "Online Phase". The offline phase by its nature is performed during the offline state of the system (e.g at init or stop operations of the system). Once performed a dictionary is formed in system's memory which later is used in the online phase of querying the database.

8.1 The Online Phase

The online phase is constructed of two stages, it include a compression stage and then a database query with the URL in a compressed form as a key. Notice that this process is done online, and it assumes that the database and auxiliary data structures are given. Moreover, in case that longest-prefix (LPM) matching is required, we first break the URL into components (by “.” and “/” delimiters) and then compress each component separately. First, given the input URL, we extract the anchors which are contained in that URL. This is done by applying a pattern matching algorithm on the URL, where the set of patterns is the set of anchors. The algorithm uses a compressed form (6) of the Aho-Corasick algorithm (8), which is based on traversing a Deterministic Finite Automaton (DFA) representing a set of anchors. Thus, the dictionary is stored as a DFA whose accepting states represent anchors. We note that it is useful to store additional information about each anchor in the DFA, namely the length of the corresponding Huffman code and a pointer to the Huffman code itself.

We notice that at each byte traversal of the DFA, the DFA state represents the set of anchors which are suffixes of the URL up until the scanned byte. We will need to decide deterministically

---

2 The full Offline Phase will be to also encode and insert all the URLs into a database.
3 In practice this is unnecessary since no anchor will originally contain the delimiter which will match only literals.
which of the anchors in this set should be used indeed for compression (e.g., suppose our
dictionary is §1="goo", §2=".com", and §3="ogle"; the URL google.com can be compressed to
either §1gle§2 or go§3§2). As we aim to minimize the total length of the compressed URL, we
are using the following greedy approach, which traverses the DFA one byte at a time, and
(conditionally) pick the anchor that minimize the length of the scanned prefix.

Specifically, for each anchor or literal $a$, let $l(a)$ be its length in bytes and $h(a)$ be its Huffman
code length. Let $u_i$ be the $i$-th byte of the URL, and let $S_i$ be the set of anchors which are suffixes
of the prefix of the URL up until $u_i$ (as represented by the DFA state after scanning $u_i$).

The deterministic selection rule works iteratively by maintaining two vectors $P$ and $V$, such that
$P[i]$ is the minimum length for encoding the first $i$ bytes of the URL, and $V[i]$ is the last anchor
or literal that achieves encoding length $P[i]$. Hence, for each byte $u_i$, we first calculate:

$$V[i] = \arg\min_{a \in S_i \cup \{u_i\}} (P[i - l(a)] + h(a))$$

And using $V[i]$ to calculate:

$$P[i] = P[i - l(V[i])] + h(V[i]), \quad (P[0] = 0)$$

We note that $h(a)$ – is the Huffman length of anchor $a$ and $l(a)$ – is the string length of anchor $a$.

When completing the traversal of the entire URL, we go backwards on the elements of $V[i]$ and
concatenate them, skipping non-useful elements (namely, after adding $V[i]$, we add
element $V[i - l(V[i])]$, skipping all elements in between). It is easy to verify by induction that
this selection results in an optimal-length encoding (given the set of anchors and Huffman code),
thus achieving the best compression ratio. Figure 3 depicts a step-by-step example of
compressing the URL comrgrnetwork.com in a component-by-component manner. Note that, for
example, in the 5th step, the DFA finds a matching with anchor "mrg", however the value of
\[ P[5 - l(mrg)] + h(mrg) = P[2]+3 = 14 \] is larger than choosing the last literal \( g \). When scanning the arrays backwards, the anchors and the literals network, \( g \), \( r \), and \( com \) are selected. The second component \( com \) is compressed by running the first three steps of the above-mentioned execution. The final step is to use the compressed-form of the key to query the database. Since we require accurate results (namely, no false positive and miss categorizations), the database maintains also the compressed form of the URL and not only its category. Comparing the lookup key with the stored key, one avoids miss-categorization due to hash collisions. Longest-prefix matching usually require a component-by-component compression and lookup. Clearly, the framework readily supports such data-structures and matching, albeit with smaller compression ratio as some compression opportunities (e.g., anchors that span more than one component) may be missed.

8.2 The Offline Phase

As illustrated in Figure 4, the offline phase of the framework, consists of three steps:

**Step 1:** Heavy hitters algorithm, in which we find a set of \( k \) frequent substrings in the set of URLs database.

**Step 2:** Anchors selection, in which we pick, from the frequent substrings, a final set of anchors. For each anchor and literal, we also calculate the estimated number of occurrences in the compressed URL database.

**Step 3:** Deterministic URL database compression and Huffman code creation, in which we use the selected anchors to replace substrings in each URL separately. We also create a Huffman code using the given frequencies of literals and anchors, which we then use to encode each URL.

8.2.1 Heavy Hitters Algorithm

The algorithm we use is Double Heavy Hitters (DHH) (7) which is geared to find frequent substrings of variable length. Specifically, it returns all the substrings which have unique frequency larger than \( n=k \), where \( n \) is the number of URLs and \( k \) is a parameter that aims to calibrate the number of frequent substrings we need to find. The algorithm is approximated and the frequency of each substring is only estimated (with an error bound of \( 3n=k \)). Note that, in any case, this frequency is not used later by the algorithm, as subsequent steps will estimate the actual number of times each substring is used for compression.
This algorithm works in time complexity $O(n \cdot L)$, where $L$ is the average URL length, in $O(k \cdot L)$ space complexity. The algorithm requires only one pass on the URL database and its space requirement is proportional to the number of heavy hitters. Yet, the results are only approximated with small error in the estimated frequency of the substrings and thus the algorithm might not find the real $k$ frequent substrings.

Moreover, we note that this algorithm avoid space pollution and if a substring $s$ is selected as a heavy hitter than the algorithm would not counts appearances of a substring $s'$, such that $s' \subseteq s$, when $s'$ is within $s$. Nevertheless, appearances of $s'$ not is within $s$ are counted, and if $s'$ appears frequently alone, it might be selected as a heavy hitter together with $s$. See, for example, Figure 4 that presents component-by-component compression of URLs. The heavy-hitters algorithm with $k = 5$ picks the substrings “network” and “net” as anchors, but not the substring “netwo” that never appears by itself.

8.2.2 Anchor Selection

The fact that frequent substrings intersect implies that a substring might not be used to compress sufficiently many URLs, even though it is frequent. Yet, these substrings increase the size of the dictionary, and therefore, should be eliminated.

Thus, in this step, we pick anchors out of the frequent substrings. Specifically, we first estimate, for each frequent substring, the database compression frequency—the number of times it will be used in the database compression (which is smaller than the frequency attached to it by the heavy hitter algorithm). Then, based on this estimated frequency, we will approximate both the gain in selecting the substring and the loss in terms of dictionary size, so that each substring whose gain
is larger than its loss will be selected as an anchor. Finally, given the definitive selection of anchors, we adjust the frequency of both anchors and literals.

8.2.2.1 Estimating the database compression frequency of a substring

In order to calculate the estimated compression frequency, we try to estimate the compression process. Since, at this point, we cannot know what will be the Huffman code of each anchor or literal, we assume in this phase that the length of encoding each literal and each anchor is the same and, without loss of generality, is set to 1. This implies that the length of a compressed URL is estimated by the sum of the number of anchors in the compressed URL and the number of the remaining literals (e.g., in the example given in section The Online Phase, the estimated length of `gle§2` is 5 and the estimated length of `go§3§2` is 4).

As explained in 1, a single URL compression involves a deterministic selection rule of specific anchors out of a larger set of anchors. In this step, we apply the same rule to select anchors out of the set of frequent strings, which implies we build a temporary DFA for all frequent substrings, set $h(a) = 1$ for each literal and anchor $a$, and run the greedy algorithm of Section The Online Phase, one URL at the time, for all URLs in the list. Each time a frequent substring is selected as an anchor when compressing a single URL, we increase the substring’s database compression frequency by 1. In the end of the process, we will have an estimation of the database compression frequencies of each substring. Notice that this is just an estimation, since not all substrings (with a frequency of at least 1) will be selected as anchors, implying the deterministic selection rule in the final online phase might yield different results. In addition, another difference in selection might be as a results of variable length encoding (with Huffman code).

8.2.2.2 Selecting anchor out of frequent substrings

We note that while replacing a parts of a URL by anchors reduces the URL size, it comes with a price: each anchor increases the size of the dictionary’s DFA and, in addition, the anchor’s encoding needed to be tracked, implying even further memory footprint. Therefore, we need to avoid picking up substrings that are not used sufficiently many times.

Let $A$ be the set of all frequent substrings, let $\Sigma$ be the set of all literals, and let $f(a)$ be the number of times substring $a \in A \cup \Sigma$ was used in the previous step. If an anchor $a \in A$ is selected, for each of these $f(a)$ times, we save $l(a) - h'(a)$ bytes, where $l(a)$ is the length of $a$ in bytes and $h'(a)$ is the length of the Huffman code of $a$. Since we cannot calculate the Huffman code of a yet, we estimate it by anchor $a$’s information content:

$$h'(a) = \frac{1}{b} \log \frac{\sum_{a \in A} f(a)}{f(a)}$$
On the other hand, inserting a to the data structures, requires adding states to the DFA and storing its Huffman code. As explained before, we estimate the Huffman code cost by \( h(a) \) bits as described in (6), the footprint of the DFA in its compressed form, is approximately \( C_{state} = 4 \) bytes per state. Notice that two anchors that share a common prefix, share also common states in the DFA. Hence, we use the following procedure to decide whether a substring is selected as an anchor. We first have an empty DFA, and sort the substrings in descending order of their gain. For each frequent substring \( a \in A \) in turn, we calculate the number of states \( states(s) \) it requires (on top of the existing DFA) and \( h(a) \). If \( f(a) \cdot (l(a) - h(a)) \geq C_{state} \cdot state(a) + h(a) \), we select \( a \) as an anchor, update the DFA, and continue to the next substring. Otherwise, we leave the DFA unchanged and skip substring \( a \). In the end of the process, we will have the set of all anchors and the corresponding DFA.

8.2.2.3 Re-estimating the frequency of anchors and literals

Since only a subset of the frequent strings was selected as anchors, the frequency of anchors and literals can be changed significantly. Thus, we ran the greedy algorithm of Section 3, using the DFA that was created in above and with \( p(a) = h(a) \) for each literal and each anchor \( a \). This will result in an updated frequency estimation of each anchor and literal.

8.2.3 Deterministic URL database compression and Huffman code creation

Now that we have the anchors and their estimated frequency, as well as the estimated frequency of all literals, we construct the Huffman codes in a standard way, treating all anchors and literals as symbols (and, thus, ignoring their original size). The result is stored in the Huffman code data structure (namely, a table with entry for each literal and anchor, where the entries of anchors are pointed out by the corresponding DFA state). We then run once again the algorithm of Section The Online Phase with the correct \( h(a) \) value for each anchor and literal \( a \). This will results in compressing each URL separately. Each compressed URL will be then inserted into the database along with its category.

8.3 Hot-updates Support

Due to nature of the dictionary, all literal are already present in the dictionary. Therefor the algorithm can simply compress any URL. In order to insert a new URL, we first obtain its compressed form by going over all the steps of the datapath. Then, instead of querying the
database, we perform an insert operation with the compressed-form URL as a key and the category as a value.

8.4 Longest Prefix Match

In order to make the framework encode URLs to support Longest Prefix Match, it is enough to split all the URLs by the requested delimiter into substring prior to the beginning of the offline phase, and either manually counting the frequency of the delimiters or living them to be single letter "URL". No other mechanism is needed.
9 Implementation Overview

9.1 Considerations

The implementation's main objective of the framework from the paper (5) is to be both space efficient and run-time efficient. In addition I wished the code to be both easy to integrate with and easy to modify in order that other research team or commercial companies will be able to quickly merge the code into their testing environments and systems.

9.2 General

The framework is written in C++11 without any use of external libraries in order to be cross-platform (Windows, Linux, etc.) and easy to use in existing projects. The framework contains a tester that shows how all the parameters the framework can operate and how to use its API for building a dictionary, encoding and decoding a URL. The tester also contain an example for the main use of the framework as a key for hashtable database queries. In addition I integrated a logging platform called easylogging++ (9) which is an open-source project.

The code itself is object oriented, each part (algorithm) of the framework is divided into independent module (Double Heavy Hitters, Compressed Aho Corasick, Huffman and Statistics) letting any future user of the framework to modify or replace any part of the framework.

The Compressed Aho Corasick implementation (10) that was published was written in C, therefore in my framework I wrapped it by a simple wrapper class.

9.3 Longest Prefix Match

The framework, when used with LPM flag will generate a dictionary that contains anchors without the delimiter. Making the encoded URL be subject to partial compare mechanisms.

9.4 Software design

The code is based on three major classes: UrlCompressor, Pattern, ACWrapperCompressed.
**Pattern**

This is struct describes the pattern, an anchor or literal. It contains all the relevant properties of the pattern, its serial number (i.e. symbol) which is more efficient than storing an 8 byte pointer) frequency, its Huffman code and its string property (e.g "\texttt{.com}").

**UrlCompressor**

This is main module which implement the frameworks API: \texttt{build / encode / decode / store dictionary to file / load dictionary from file}. It stores an array of the patterns that was allocated in single allocation keeping the memory allocation minimal as possible.

**ACWrapperCompressed**

The ACWrapperCompressed is a wrapper class that implements a C++ interface for the Compressed Aho-Corasick algorithm used by the framework. I had to modify the original code of the Compressed Aho-Corasick to support incremental digest of the URL which calls a callback function with the patterns matching each byte of the URL. With this callback function we can calculate the $V_i$ and $P_i$ vectors.

In addition, I had to modify the original Aho-Corasick implementation since it simply printed matching patterns but could not execute any operation on them (i.e report, or call a callback function). My modification made the Aho-Corasick implementation general and usable for integrating with more algorithms.
9.5 Memory efficiency

The implementation of algorithms as programs consume more memory than the theoretical analysis usually used to assess the memory consumption of the algorithm. Programs also use metadata that is needed to handle the algorithm data structures such as reserved location to NULL in char arrays, size variables for arrays, pointers. Some dynamic allocations also costs a size variable that is stored in the beginning of the allocated memory. We expect additional memory lose to alignment of objects in memory and unused memory in slabs. Before any memory improvement the program's footprint was about twice the memory allocated. Since my goal was to keep the framework reusable I avoided any OS specific compiler commands that can optimize the way the code is compiled or executed nor avoided implementing memory allocator that assumes any explicit behavior of the memory slab system described in the background chapter.

9.5.1 Using dynamic memory allocation efficiently

Though the algorithm's main objective is to have a low memory footprint the basic data structures (DFA, lists, arrays) general implementations are based on dynamic memory allocation. In practice due to the slabs based memory management of the operation system which is optimized for speed, this is very inefficient. Consider a DFA in which each state is structured by a struct of 18 bytes. When allocated it requests 18 bytes buffer, but it is allocated from a 32 bytes slab leading to a loss of additional 78% of unnecessary memory (that is now blocked until the item will be free). The framework avoids this lose by implementing an optimized memory allocator (SerialAllocator) that pre-calculate the memory needed for the special inner data structures and allocates a big buffer of aligned memory that later can be pulled from the poll.

Since many real life low level systems implement their own memory management algorithm (e.g. specialized slab system) it will not be wise to try to create more sophisticated memory allocators that tries to manage the buffers themselves as slabs system. Keeping the allocations consolidated and large will help users to optimize its own memory management to the framework without modifying the inner code of the modules.

9.5.2 Using efficient alignment (and avoid padding)

The C++ compiler is built to optimize code performance mainly in terms of execution. This leads the compiler to lay each non primitive (char, int, long, etc.) in an address that is a multiplication of 8 (or 4 in 32bit based operating systems). But in turn making the sizeof() classes and structs to be larger than expected. For example consider this struct:
struct example {
  long eight_bytes;
  char one_byte;
};

Its real time size of 16 bytes though it is constructed by an 8 byte and a 1 byte members. Adding an inner struct after the char will be forced by the compiler to pad the struct until the next free 8 multiple address.

In modern C++ compilers a special instruction are implemented to instruct the compiler to optimize variables in different mode. But these instructions are not OS independent standard and might not be implemented (or implemented differently) in real time lean compilers for embedded systems.

To avoid enlarging the actual memory fingerprint I: 1) optimized the order of members in the classes keeping them aligned by design. 2) Allocated buffers for combined variables and manually assigned them from the buffer. Using these methods I reduced the memory lost to alignment by a significant amount. For example, I reduced the Pattern object's (described later) size from 48 bytes to 32 bytes when its theoretical size is 30 bytes (Noting that due to the slab system 48 bytes struct will real be used in a 64 bytes slab).

9.6  Runtime Optimizations

During the online phase there are two cpu consuming processes which I wanted to make as lean as possible, making the code work fast and efficient. The first is the hot-path of the Aho-Corasick algorithm where the URL is digested byte by byte and the second is converting the $P_i$ and $V_i$ vectors into a selected patterns cover and encoding the patterns with the Huffman code.

9.6.1  Reduced function calls in hot-path

The most CPU consuming step of the online phase is the Aho-Corasick pass on the URL. All the code in this pass is compiled as inline functions except the callback function for handling a pattern. Making the framework to use as little cpu ops per byte.

9.6.2  Fast Huffman encoding

In the final step of the encoding in the online phase we convert the list of anchors in the anchors cover of the URL into Huffman code. To do that efficiently and fast each Huffman code is stored in a 32bit primitive member. Each concat operation of Huffman code requires shifting the code into the next encoded bit and using the OR operator to set it into the output buffer. This yields
for at most two shifts and two OR operations per pattern in the patterns cover (two operation since the code might be splitted between to positions in result encoded buffer). So we get $O(\log a)$ ops for the final encoding, where $a$ is the number of anchors in the encoded URL which is at most the number of bytes in the URL.

9.7 The Tester

Within the framework a tester tool is located. This tool serves as both an example of how to use any of the frameworks API and as tool to repeat the results and graphs in this article. In addition to the tool bash scripts are found that using the tester they produce CSV files that generated the graphs in this article.

9.7.1 Tester CMDs

The tester is consisted of several commands: "build" a dictionary file from URLs list, "encode" URLs from file using prebuilt dictionary file, "compress" / "extract" URLs file (like .zip ), "test" - Compress the whole file once, "article" - Compress 20 times, 10 sets of 10,000 random URLs, "testhash" - Insert all urls into a hashtable by their compressed form.

9.7.2 Tester flags

In addition the tester has several flags that can be used to change it behavior. Choosing the parameters for the Double Heavy Hitters algorithm, $n, r, k$-grams. Defining whether building the dictionary to support the LPM mode. Adding verification to the tests by decoding the encoded URLs. And several debugging flags to unveil the inner parts of the algorithm like the Aho-Corasick state machine and the framework's dictionary.
10 Evaluation and Results

In this section I will present the framework's performance analysis. I will not compare it to other algorithms as that work was already done in the article.

As a test case for the framework I used the same database used in the article (5) (from a different date) available in URLBlackList.com (11). This is also available in the framework's GitHub project (12). This daily-generated list consists of about 2,200,000 unique domain names and 95 different categories.

An important notation which will be used, *memory allocated* means the memory allocated by code design (i.e. the sizes inside a malloc syscall). While *memory footprint* is the actual memory measured in the system assigned for this allocation after padding/alignment/slab system or other compiler and operating system optimizations.

10.1 The effect of the number of anchors on performance

As the focus of this paper is the memory footprint of the framework (namely, size of all the data structures in memory needed for the compression process) the main characteristic is the *compression ratio*, which is the ratio between the memory allocated and the size of the compressed database compared to the uncompressed URLs size. More specifically, we calculated the memory footprint by summing up the size of the dictionary, the size of the Huffman code table, and the length of each URL in the database in its compressed form. We compare it with the total length of all URLs in their uncompressed form. Since the main goal of this framework is to offer an optimized solution for fast hashtable lookups our secondary characteristic is the *throughput* of encoding the URLs into hashtable keys.

Each performance number was measured by applying the datapath compression 20 times on 10 randomly-selected sets of 10,000 URLs (namely, 200 runs per performance number, each representing compression of 10,000 URLs in a batch). All of the experiments ran only on a single core. I did not measure the data-base lookup as this is orthogonal to the framework and can be as a successive pipeline stage and any user might choose different database with different characteristics.

Error! Reference source not found. and Figure 7 show the throughput (per core) and compression ratio of the datapath as a function of the number of anchors. In the graphs we can compare between URLs, regular encoding and Components which is when using the LPM mode with '.'
delimiter. The performance of the datapath depends on both the size of the auxiliary data structure and the number of anchors (per URLs) used for compressions.

As the number of anchors increases, more anchors are matched at each byte of the URL, making the calculation of $V[i]$ longer which takes more ops making the encoding take longer. In the LPM mode the anchors are shorter. This has an effect on the compression since anchors cover less bytes of the URL. And on throughput as the shorter anchors often match more internal suffixes of the URL. On the other hand, more anchors mean more common patterns that can be converted into Huffman, improving the compression ratio.
10.2 The effect of number of anchors on memory footprint

Figure 8 shows the memory footprint ratio of my implementation of the framework from the size of the uncompressed form of URLs. The auxiliary data ratio addresses two memory characteristics: the theoretical memory allocated ratio and real memory footprint ratio as measured. The real footprint of the framework takes under consideration the loss of memory for the slab-based memory management, memory alignment, and other optimizations done by the compiler or the operating system.

In order to measure the theoretical memory usage I added to the framework a statistics class that sums the sizes of all malloc and new ops. To measure the real memory footprint I used a special Unix mallinfo() syscall that among others measures the current memory heap size. I tested the heap size right before and right after the initiation of the framework (right until it is ready for online usage, encoding / decoding URLs).

Figure 8 - The effect of number of anchors on memory footprint

Figure 8 shows that the real memory footprint ratio is 20% larger than the theoretical ratio. The fact that the difference between these two measurements is linear leading to the conclusion that each additional anchor holds a constant additional memory allocated by the operating system. A slight difference is seen between regular and LPM mode which is a result of shorter anchors.
Figure 9 shows the breakdown of the memory allocated by the framework while the liner line is the real memory footprint. This breakdown is measured against the uncompressed size of the URLs in the dictionary. *Patterns and codes* refers to the memory allocated for storing all anchors and their Huffman codes. *Statemachine allocation ratio* refers to the DFA in memory allocation and the *pointers allocated ratio* is the pointer table needed to point on from each pattern in the DFA to its complementary anchor in the dictionary. As seen from this graph, the DFA (Compressed Aho-Corasick) consumes about than 27% of the memory needed for this framework.

Figure 10 shows the *dictionary ratio* which is the minimal dictionary (or say data) needed for storing the dictionary in a file versus the total *auxiliary data ratio* and the *auxiliary data footprint ratio*. When storing the dictionary it sufficient to store only the patterns and frequencies since the Huffman codes and the DFA can be easily re-computed. In addition storing the data into a file doesn't suffer from the loss of memory for padding and etc. like the difference between memory allocated to the actual footprint.
10.3 Real hashtable memory footprint

<table>
<thead>
<tr>
<th></th>
<th>Compressed form</th>
<th>Uncompressed form</th>
<th>ratio</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Key size</strong></td>
<td>21168KB</td>
<td>57197KB</td>
<td>0.37</td>
</tr>
<tr>
<td><strong>Bucket list hashtable</strong></td>
<td>91152KB</td>
<td>120930KB</td>
<td>0.75</td>
</tr>
<tr>
<td>(Huffman only)</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>Bucket list hashtable</strong></td>
<td>84902KB</td>
<td>120930KB</td>
<td>0.70</td>
</tr>
<tr>
<td><strong>Cuckoo hashtable, load factor 0.8</strong></td>
<td>61417KB</td>
<td>91195KB</td>
<td>0.67</td>
</tr>
<tr>
<td>(Huffman only)</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>Cuckoo hashtable, load factor 0.8</strong></td>
<td>55166KB</td>
<td>91195KB</td>
<td>0.60</td>
</tr>
</tbody>
</table>

Figure 11 shows the memory efficiency of using the framework as compressed keys for a hashtable with 16,000 anchors. Both hashtable is constructed by a pair of <URL, IPv4>. *Key size* is the sum the key (URL or encoded URL) plus its container. The compression ratio between the keys is better than seen in "The effect of the number of anchors on performance" since string container has some more metadata such as "\0" character and length. Inserting the URLs as key into *Bucket list hashtable* where the value is IPv4 (4 bytes) each element in the bucket list contains additional pointer (8 bytes) for the list and we get a ratio of 0.70. When we use a
memory efficient hashtable such as Cuckoo hashtable with load factor of 80% we see a much better ratio of 0.60. Also we see 7% improvement in the total memory consumption of the framework comparing to "pure Huffman" encoding for the cuckoo hashtable implementation.

10.4 Evaluate realtime behavior

Since callgrind (13) cannot be used in partial execution of the program I ran it on a special tester that runs the Offline phase and then the Online phase on the entire set of URLs. Shown in Figure 12 is the function call breakdown of the system, the percentage of CPU spent at each module. About 50% of the execution time, the program spent on the online mode, which we can consider the reference point. The handle_pattern is the callback function that handles each anchor found by the DFA (that builds the $V[i]$ and $P[i]$ vectors). Hence we get that for the online phase alone, 8% the CPU time is used for building the Huffman encoding, 87% of the CPU time was used for digesting the URL in the DFA and 5% of the CPU time was used to calculate and update the $V[i]$ and $P[i]$ vectors.

<table>
<thead>
<tr>
<th>Anchors</th>
<th>Encoding Throughput Mbps</th>
<th>DFA Throughput Mbps</th>
<th>Other Throughput Mbps</th>
<th>Encoding/DFA ratio</th>
</tr>
</thead>
<tbody>
<tr>
<td>100</td>
<td>272.21</td>
<td>364.28</td>
<td>1076.94</td>
<td>0.75</td>
</tr>
<tr>
<td>1000</td>
<td>160.35</td>
<td>184.39</td>
<td>1229.74</td>
<td>0.87</td>
</tr>
</tbody>
</table>
Figure 13 shows a comparison of the framework's throughput versus only running the DFA on the same set of URLs. At 1,000 anchors it we get the same 87% ratio as seen above. As expected on small list of anchors the DFA is a bit faster as more of its states fit the closer to CPU cache so memory access between its states are faster. When encoding with the same amount of anchors the framework also needs to access the Huffman encoding table which when accessed is taking also space in the closer L1 and L2 caches.

10.5 The effect of Double Heavy Hitters parameters on performance

In Figure 14 we examine the effect of the Heavy Hitters parameters, k-grams and r on the result performance of the framework. The first observation is that when k-grams is very large or small the HH algorithm doesn't find enough patterns as expected. Second, the difference between r=0.5 and r=0.8 has only a slight compression and throughput effect.
When it comes to the \textit{k-grams} the \textit{compression ratio} and \textit{throughput} have inverse relation. Bigger \textit{k-grams} yields longer patterns that are less common and therefore tend not to participate in the common compression process. In the other hand, the fact that less anchors are matched in the DFA during the compression process yields less calculation and therefore better throughput. Therefore the best practice would be to use lower \textit{k-grams} such as 4 and control our efficiency using the number of anchors.
11 Conclusions

This paper introduces the implementation of our framework that significantly reduce the memory footprint of URL-based databases and forwarding tables, while maintaining the accuracy of the lookup processing (namely, no false positives or miscategorizations) and incurring only a small overhead in time. The framework also allows hot updates of the database and a longest prefix matchings (encoding component by component).

As seen in the "Evaluation and Results" section this implementation is memory efficient and has only 20% operating system overhead. This overhead can be furthermore improved in real life implementation if integrated with specialized memory management. The real-time analysis shows that the algorithm part of the framework described in the paper (5) takes less than 5% of the online process and has throughput at around 1Gbps.

The framework is written so any user, without understanding behind the hood of the code, could redo the test or implementing it in their system. Not using external libraries makes the reuse of the code as simple as copying the source code from the GitHub repository (14). Following the code of the provided tester enables users to understand the API and how to use each and every parameters.
References


“Scalable URL Matching with Smaller Memory Footprint”

דניאל קריאטובסקי

עבודת גמר המבוגשים חלחול מחברותعلم לשכת חינוך ותרבות הפועלים
opus. ספר אפי ארזי למдесяית המחשב, המרכז הבינתחומי הרצליה

אפריל 2016