Software-Based Implementations of Updateable Data Structures for High-Speed URL Matching

Haowei Yuan, Benjamin Wun and Patrick Crowley
Computer Science and Engineering
Washington University in St. Louis
{hyuan, bw6, pcrowley}@wustl.edu

ABSTRACT

URL matching is used in many network applications, including URL blacklisting, URL-based forwarding and URL shortening services. These applications need fast URL queries and updates, thus requiring an efficient updateable data structure. As the processing power of general-purpose multi-core processors increases, software-based approaches are better able to meet the speed requirements of URL matching. In this paper, we present our preliminary performance study of finite-automata- and hash-based URL matching implementations on commodity PCs. The impacts of the cache and memory allocation methods are discussed.

1. INTRODUCTION

As the Internet continues to grow, more and more URLs are generated all the time. URL matching is employed in many network applications, including 1) URL blacklisting [1], which is widely deployed by enterprises to better control their networks; 2) URL-based forwarding [7], which makes forwarding decisions based on the URLs contained in the packets; and 3) URL shortening services [2], where a long URL is represented by a short URL for easy sharing in social networking. As network traffic increases, these applications need to support faster queries. This problem is similar to the extensively studied fast exact string matching problem; however, the URL matching applications require more frequent updates than the traditional exact string matching applications, including signature-based network intrusion detection systems (NIDS). In this paper, we present the preliminary performance results of software solutions based on hash table and deterministic finite automata (DFA) for URL matching. Hash-based solutions provide better lookup speed than DFAs, but their throughput is still affected by cache organization and memory allocation methods.

2. DATA STRUCTURES

2.1 Finite Automata

Finite automata are commonly used data structures for exact string and regular expression matching [6]. In this paper, we use the open source regular expression processor[5], which employs both default transitions and alpha-beta reduction to minimize memory consumption, to build DFAs for URL matching. Note that the current DFA-based solutions do not support URL updates on the fly, so updates require reconstructing the DFA(s).

2.2 Hash Table

Hash tables [4] are conventional data structures for data lookup and retrieval, and they can easily update stored data. We implemented a simple hash table for URL matching in C. Each hash entry has a single bit indicating whether a URL is stored in this bucket or not. Since URL lengths vary greatly , we have found it is more memory-efficient to dynamically allocate the memory to store them. As a result, a pointer pointing to the actual URL is stored in each bucket. Linear probing is used to resolve hash collisions.

3. PERFORMANCE

Since the DFA-based solution does not support URL updating, we only compare the URL lookup performance between the DFA- and hash-based implementations. The impacts of the cache organization and dynamic memory allocation in the hash table implementation are also discussed.

3.1 URL Lookup Comparison

The URLs used for the performance test are from Shalla Secure Services [1]. The dataset contains 111,647 URLs, with an average of 31.46 characters per URL, with the longest URL containing 741 characters. The test program was run on an Intel Core 2 Duo T7300 2.0 GHz processor, with 4,096 KB of L2 cache. The first 10K URLs in the dataset are stored in the DFA or hash table. The benchmark program then attempts to lookup all 111,647 URLs sequentially, and repeats this 1,000 times. The system throughput, memory consumption and average memroy references per URL for the DFA and the hash tables with 25%, 50% and 75% load

Table 1: DFA vs. Hash Table

	DFA	H(25%)	H(50%)	H(75%)
Throughput (Mbps)	156	6,086(P)	4,624(P)	2,123(P)
		2,149(H)	1,852(H)	1,269(H)
Memory(KB)	1,501	305.28	304.98	304.87
Mem Ref	31.20	1.37	2.47	7.82

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.
ANCS'10, October 25-26, 2010, La Jolla, CA, USA. Copyright (c) 2010 ACM 978-1-4503-0379-8/10/10...$10.00.

Table 2: Increasing the # of stored URLs, LD=25%

# of Stored URL	10K	30K	50K	70K
Throughput(Mbps) (P)	6,086	3,043	2,305	1,781
Avg Mem Ref	1.37	1.33	1.29	1.25

factors (LD) are reported in Table 1. The hash table load factor is defined as the ratio between the occupied hash entries and the total amount of entries in the hash table. We notice that the selection of the hash function affects the throughput, thus, we report the throughput of both hashing the URLs on the fly (noted with (H)) and using pre-computed hash values (noted with (P)). We see in Table 1 that the hash table lookup speed is much faster than the DFA approach and consuming less memory. This behavior is largely due to the fact that the DFA needs to make a memory reference for every incoming character, while the hash table requires many fewer memory references for each URL. In addition, there is a large throughput gap between the hash table that computes the hash results on the fly and the one using pre-computed hash values, indicating that more efficient hash functions would be beneficial.

3.2 Hash Table Design Issues

3.2.1 Cache

Processors employ caches to improve performance. However, it is well known that network traffic has poor locality. In Figure 1 we present how the cache miss rate of the entire software changes as the hash table LD increases. The LD is increased by decreasing the hash table size while always storing 10K URLs in the table. We also show how the cache miss rate changes when the number of URLs stored in the hash table varies while the hash table LD is kept as 25%. The corresponding system throughput and average number of memory references per lookup are listed in Table 2. The same test program used in Section 3.1 is used here. The cache miss rate data is collected using cachegrind [3]. Figure 1 shows that the cache miss rate decreases as the LD increases. We suspect this is because there are more hash collisions when the LD becomes higher, resulting in more memory references per URL lookup as shown in Table 1. When walking through these entries, the cache lines may be refilled and provide better hit rates for the subsequent queries. Figure 1 also shows that when the LD is fixed, storing more URLs in the hash table increases the cache miss rate. We suspect the reason is that the average number of

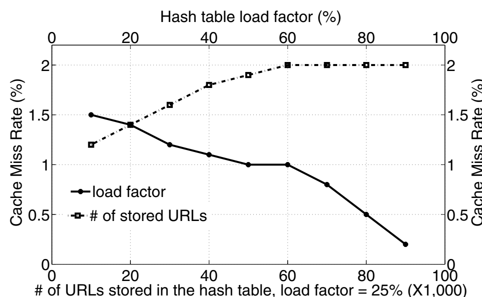

Figure 1: Cache miss rate behavior

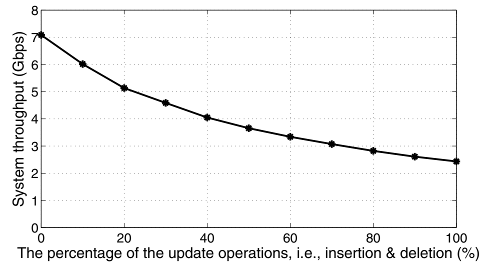

Figure 2: System throughput vs. the percentage of the update operations, LD = 25%.

memory references per lookup decreases as shown in Table 2, so the cache lines are not refilled effectively. Overall, both the cache miss rate and the total number of memory references affect the system throughput, which factor is more important depends on the particular experiment setup. We plan to verify our speculation of the relationship between the cache miss rate and the number of memory references in future work.

3.2.2 Dynamic Memory Allocation

Dynamic memory allocation is used when an URL is inserted in or deleted from the hash table. In our implementation, the standard *malloc* and $free$ functions are used. It takes the processor extra time to allocate memory dynamically, so we have measured how much the system throughput is affected by dynamic memory allocation. In our test program, 10K URLs are stored in the hash table, then the rest of the URLs are queried sequentially and each URL is inserted and deleted once. We vary the percentage of the update operations while maintaining the same amount of hash table operations. Figure 2 shows the system throughput drops significantly as the percentage of update operations increases. This indicates that more efficient dynamic allocation schemes would be worth trying.

4. CONCLUSION

In this paper, we documented our experience of implementing updateable data structures for URL matching on commodity PCs. The hash-based solution outperformed the DFA-based solution. Cache and dynamic memory allocation were found to affect the software system performance.

References

[1] Shalla secure services: http://www.shallalist.de/.

[2] Tinyurl: http://www.tinyurl.com/.

[3] Valgrind: http://www.valgrind.org/.

[4] N. Askitis et al. Cache-conscious collision resolution in string hash tables. In *SPIRE*, pages 92–104, 2005.

[5] M. Becchi. Regex-processor: http://regex.wustl.edu/.

[6] M. Becchi et al. Evaluating regular expression matching engines on network and general purpose processors. In *ANCS*, pages 30–39, 2009.

[7] G. A. David et al. Design, implementation and performance of a content-based switch. In *INFOCOM*, pages 1117–1126, 2000.