Abstract

Hash tables are employed in several network processing algorithms and applications, such as route lookup, packet classification, per-flow state management and network monitoring [1]. In order to avoid possible bottlenecks in the data-path of a high-speed router, the underlying hash table has to be well designed. This overview article summarizes one such approach by H. Song, S. Dharmapurikar, J. Turner, and J. Lockwood. Song et al. present a novel hash table data structure and lookup algorithm which improves the performance over a naïve hash table by reducing the number of memory accesses needed for the most time-consuming lookups. The proposed algorithm extends the multiple-hashing Bloom filter to support exact matches and exploits recent advances in embedded memory technology [1].

1 Introduction

A hash table is a versatile data structure for performing fast associative lookups, which requires $O(1)$ average memory accesses per lookup. In order to do hashing, we use a hash function $h$ to compute the slot of the table from the key $k$. We say that $h$ maps the universe $U$ of keys into the slots of the hash table $T[0..m-1]$: $h: U \rightarrow \{0, 1, \ldots, m-1\}$ [2].

The Bloom filter is a space-efficient probabilistic data structure that is used to test whether an element belongs to a set. It eliminates false negatives whereas false positives are possible and their probability increases as we add more elements to the set. An empty Bloom filter is a bit array of $m$ 0-bits. $k$ different hash functions are defined, each of which maps some set element to one of the $m$ array positions with a uniform random distribution [3]. An illustration of the Bloom filter is presented in Figure 1.1.

2 Old Fast Hash Table Designs

Song et al. first present the basic form of the algorithm which they call Basic Fast Hash Table (BFHT) and then it gets gradually improved upon. A variant of a Bloom filter called Counting Bloom Filter is used in which each bit of the filter is replaced by a counter. The counter in this filter essentially gives us the number of items hashed in it [4]. An illustration of such data structure is shown in Figure 2.1.

It can be seen that in case of BFHT, we need to maintain up to $k$ copies of each item. However, only one copy of each item – the copy associated with the minimum counter value – is accessed when the table is probed. Song et al. used this observation to develop the Pruned Fast Hash Table (PFHT) which requires much less memory compared to BFHT. After pruning is completed, the counter value, however, no longer reflects the number of items ac-
3 Shared-node Fast Hash Table

The Shared-node Fast Hash Table (SFHT) allows easy incremental updates at the cost of a little more memory than the required for PFHT but significantly less than that of BFHT. The main idea of the algorithm is to allow the multiple instances of the items to share the same item node using pointers (see Figure 3.1). The lookup performance is the same as that of the BFHT but slightly worse than the PFHT.

A Shared-node Fast Hash Table data structure can be kept on-line because of its reduced memory requirement compared to BFHT. This is especially useful for applications that involve time critical updates which must be performed as quickly as possible. One such case is the TCP/IP connection context table where connections get set up and broken frequently and the time for table query per packet is comparable to time for addition/deletion of connection records.

4 Conclusion

Engineering a resource efficient and high-performance hash table is a challenging task. In their paper, H. Song, S. Dharmapurikar, J. Turner, and J. Lockwood examine various hash table designs (BFHT and PFHT) and propose the Shared-node Fast Hash Table data structure. Their hash table algorithm extends the multi-hashing technique – Bloom filter – to support exact match. However, unlike the conventional multi-hashing schemes, SFHT requires only one external memory for lookup. As a result, the research output can be used as a module to aid several networking applications.

References