Two-tier Bloom filter to achieve faster membership testing

M. Jimeno, K.J. Christensen and A. Roginsky

Testing for element membership in a Bloom filter requires hashing of a test element (e.g. a string) and multiple lookups in memory. A design of a new two-tier Bloom filter with on-chip hash functions and cache is described. For elements with a heavy-tailed distribution for popularity, membership testing time can be significantly reduced.

Introduction: Bloom filters [1] are a space-efficient, probabilistic data structure for representing a list of elements (for example, a list of strings). A Bloom filter is an array of m bits. A string is mapped into a Bloom filter by inputting it to a group of k hash functions resulting in k array positions. Each indexed array position is set to 1. A string is tested for membership by inputting it to the same group of k hash functions. If all k generated array positions are determined to be set to 1, then the string is probably a member. False positives can occur with Pr [false positive] $= (1 - 1/m)^{kn}$ for n elements mapped into a Bloom filter.

Bloom filters are widely used with many applications in the domain of networks [2]. One application of interest is representing large file lists; for example lists of shared files in servers or caches to enable determination if a given file name is in a list of shared files. The key performance measures for a Bloom filter are membership testing time (also called computation time in [3]), memory requirements, and probability of false positive. Membership testing time is the time to determine if an element belongs to the set represented by the Bloom filter. The key motivation for using Bloom filters (over more conventional data structures) is reduced memory requirement and faster membership testing [3]. Membership testing time is a function of the time to (a) compute up to k hashes and (b) perform up to k lookups in memory where the Bloom filter array is stored. Memory lookup times depend on the type of memory in which the Bloom filter is stored (e.g. high-speed localised static random access memory (SRAM) or slower main memory dynamic random access memory (DRAM). In this Letter, the membership testing time for a Bloom filter is reduced by implementing hashing directly in specialised hardware and by introducing a second tier cache Bloom filter to reduce the number of accesses required into slower main memory.

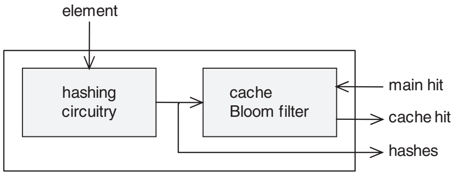

Fig. 1 *Design of two-tier Bloom filter*

If cache hit is false then hashes are used for lookup in external Bloom filter in main memory. If main hit is true then cache learns the element

Two-tier Bloom filter: A two-tier Bloom filter design is proposed to reduce membership test time. Fig. 1 shows the basic design of a single component (such as a specialised chip) containing hashing circuitry and a Bloom filter. The two tiers might exist in a hardware implementation of a Bloom filter, or between processor cache and main memory in a computer. An on-chip Bloom filter can be implemented using fast SRAM, but it is limited in size to about $m = 4$ Mb [4]. Using $m/n = 32$ and $k = 22$ (as recommended in [5]) to achieve a low probability of false positive, the number of elements that can be mapped into the on-chip Bloom filter is $n = 131\,072$. For applications with more than 131 072 elements, the on-chip Bloom filter can be used as a cache for a larger off-chip Bloom filter stored in the main memory (typically implemented in DRAM, with slower lookup time than SRAM) of a system. The two-tier Bloom filter takes as input the element (e.g. a string) to be mapped into, or tested for membership, and outputs the k hash values and a single test output to indicate if the element being tested for was found (or 'hit') in the on-chip cache Bloom filter. If the cache hit output is false, then the computed hash values are used to test the external Bloom filter in main memory. If the element is found in the external Bloom filter the main hit input causes the cache Bloom filter to learn the element (if the cache is not yet full). Thus, the cache Bloom filter contains a subset of the elements mapped into the main memory Bloom filter (the main memory Bloom filter contains mappings for all elements). How well the cache learns the most popular elements determines the performance (measured in membership testing time) of the two-tier Bloom filter.

Membership testing time (T_{test}) is a function of hashing time (t_{hash}), cache Bloom filter testing time (t_{cache}), main memory Bloom filter testing time (t_{main}), and probability of a successful membership test in the cache Bloom filter (p_{cache}). The membership testing time is $T_{test} = t_{hash} + t_{cache} + (1 - p_{cache})t_{main}$ for the two-tier Bloom filter and $T_{test} = t_{hash} + t_{main}$ for a single Bloom filter in main memory. In order for the two-tier Bloom filter to have a smaller membership testing time than that of a single Bloom filter in main memory, $t_{cache} - p_{cache}t_{main} < 0$ must hold. The speed-up (S) of T_{test} is the ratio of the time required by the two-tier Bloom filter divided by the time required by a single Bloom filter stored in main memory. The speed-up expresses the relative (percentage) reduction in membership testing time by using the two-tier Bloom filter. Speed-up is

$$S = \frac{t_{hash} + t_{main}}{t_{hash} + t_{cache} + (1 - p_{cache})t_{main}} \tag{1}$$

Application to file search: The specific target application for the two-tier Bloom filter is membership testing for a file system containing millions of files, each file with a unique identifier (e.g. path plus file name). It is well known that the distribution of the requests for files in some applications such as P2P (peer to peer) file sharing and web caching follow a Zipf-like distribution [6, 7] where the probability of requesting the element ranked jth in popularity among a population of N elements is $\Pr[j] = \Omega/j^{\alpha}$, where Ω is the normalisation constant and α is the shape parameter. The normalisation constant is the inverse sum of $1/j^{\alpha}$ for $j = 1, 2, \ldots, N$. For $\alpha = 0$ the distribution is uniform and as α increases the distribution becomes skewed and heavy tailed. For P2P file sharing, α values between 0.60 and 0.83 have been measured [6].

Performance evaluation: In this performance evaluation we use simulation to study the speed-up, S, of the two-tier Bloom filter compared to a single Bloom filter stored in main memory. The speed-up of the two-tier Bloom filter for a stream of Zipf distributed membership tests is a function of t_{hash}, t_{cache}, t_{main} and p_{cache} (the subscripts *cache* and *main* describe the parameter as applying to the cache and main memory Bloom filters, respectively). The probability p_{cache} is a function of the size of the cache (n_{cache}), the popularity of the elements (modelled here with a Zipf distribution with parameters $n_{main} = N$ and α), and how well the cache learns the n_{cache} most popular elements. If the cache Bloom filter learns (and thus represents) the n_{cache} most popular elements – called a 'perfect cache' in this Letter – for the population of N elements, then the probability of a cache hit is the cumulative probability mass of elements $1, 2, \ldots, n_{cache}$:

$$p_{cache} = \sum_{j=1}^{n_{cache}} \frac{\Omega}{j^{\alpha}} \tag{2}$$

In reality, the cache will learn a set of elements that are less than perfect. This is called the 'realistic cache' in this Letter and its cumulative probability mass can be computed as follows. Given N distinct elements that are sampled with replacement, let x_j be the probability of drawing an element of type j. Sampling continues until M elements of different types are sampled (this corresponds to the cache being fully loaded). Here M is n_{cache} and N is n_{main}. The cumulative probability, T, corresponding to p_{cache} for a realistic cache is

$$T = \sum \left(x_{u_1} + x_{u_2} + \cdots + x_{u_M}\right) \cdot \Pr[\text{subset } G \text{ is sampled first of all } M \text{ subsets of } N \text{ types}] \tag{3}$$

where G is the subset $\{u_1, u_2, \ldots, u_M\}$ and where the summation is taken over all M subsets of N types of elements. This summation is likely to be intractable to compute given the very large number of subsets possible for a large N. Given this intractability, a simulation model of the two-tier Bloom filter was created. From this simulation model p_{cache} for a realistic cache could be experimentally estimated.

The following parameters were the control variables for the evaluation. For the elements, $N = 8 \times 10^6$ and $\alpha = 0, 0.1, 0.2, \ldots, 1.0$, modelling 8 million elements in the population with popularity ranging from uniform ($\alpha = 0$) to highly skewed ($\alpha = 1$). For hashing and memory access time, $t_{hash} = 0$ (to focus on speed-up from caching effects only), $t_{cache} = 1$, and $t_{main} = 5$ (modelling SRAM as 5 times faster than DRAM). For the cache Bloom filter the parameter values were $m_{cache} = 4\,\text{Mb}$ and $m_{cache} = 16\,\text{Mb}$, and $k_{cache} = 22$. For the main memory Bloom filter the parameter values were $m_{main} = 256 \times 10^6\,\text{b}$ (or about 30.5 MB to be able to represent $N = 8 \times 10^6$ elements with 32 bits for each element), and $k_{main} = 22$.

Fig. 2 shows the results for p_{cache} from (2) for a perfect cache and from simulation for a realistic cache. The simulation results for p_{cache} for a realistic cache are the average of 30 trials for each value of α. It can be seen that p_{cache} increases when α increases. It can also be seen that p_{cache} for the perfect and realistic caches could be up to 100% different for α between 0.4 and 0.7. Fig. 3 shows the results for speed-up. It can be seen that the speed-up is achieved for values of $\alpha > 0.7$ and the larger m_{cache} is, the greater the speed-up. The results show that even with a small cache memory, the two-tier Bloom filter can achieve faster membership testing for a heavy-tailed distribution of elements. The relationship between t_{cache} and t_{main} affects the possible speed-up.

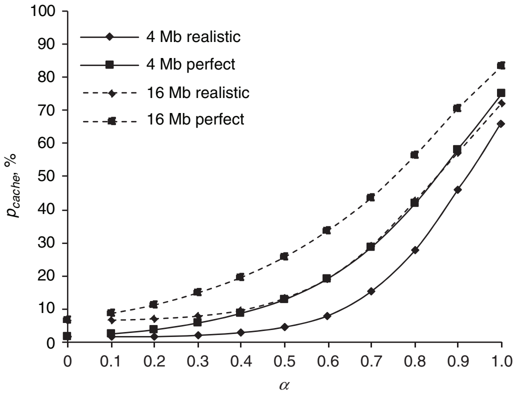

Fig. 2 *p_{cache} against m_{cache} and α*

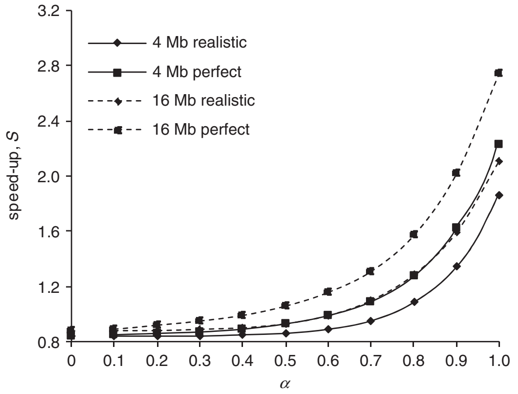

Fig. 3 *S against m_{cache} and α*

Conclusions and future work: A new two-tier Bloom filter architecture has been developed that can achieve a shorter membership testing time than a single-tier Bloom filter. The two-tier Bloom filter implements a caching Bloom filter in on-chip SRAM. Future work includes (*a*) studying cache learning to determine when to invalidate the cache and re-learn it (for example, when the popularity of elements changes) and (*b*) developing greater insight into (3), including finding an approximation that is computable.

8 January 2008
Electronics Letters online no: 20080081
doi: 10.1049/el:20080081

M. Jimeno and K.J. Christensen (*Department of Computer Science and Engineering, University of South Florida, 4202 East Fowler Avenue, ENB 118, Tampa, FL 33620, USA*)

E-mail: mjimeno@cse.usf.edu

A. Roginsky (*Computer Security Division, National Institute of Standards and Technology, Gaithersburg, MD 20899, USA*)

M. Jimeno: On leave from Universidad del Norte, Barranquilla, Colombia

References

1 Bloom, B.: 'Space/time trade-offs in hash coding with allowable errors', *Commun. ACM*, 1970, **13**, (7), pp. 422–426

2 Broder, A., and Mitzenmacher, M.: 'Network applications of Bloom filters: a survey', *Internet Math.*, 2004, **1**, (4), pp. 485–509

3 Mitzenmacher, M.: 'Compressed Bloom filters'. Proc. 20th ACM Symp. on PODC, August 2001, pp. 144–150

4 Dharmapurikar, S., Krishnamurthy, P., and Taylor, D.E.: 'Longest prefix matching using Bloom filters', *IEEE/ACM Transactions on Networking*, 2006, **14**, (2), pp. 397–409

5 Fan, L., Cao, P., Almeida, J., and Broder, A.: 'Summary cache: a scalable wide area web cache sharing protocol'. ACM SIGCOMM, 1998, pp. 254–265

6 Chu, J., Labonte, K., and Levine, B.: 'Availability and locality measurements of peer-to-peer file systems'. *in* 'ITCom: scalability and traffic control in ip networks', Proc. of SPIE, July 2002, Vol. 4868

7 Breslau, L., Cao, P., Fan, L., Phillips, G., and Shenker, S.: 'Web caching and Zipf-like distributions: Evidence and implications'. Proc. IEEE INFOCOM, April 1999, pp. 126–134