

# P2P Directory Search: Signature Array Hash Table

Miguel Jimeno and Ken Christensen

Department of Computer Science and Engineering  
University of South Florida  
Tampa, Florida 33620  
{mjimeno, christen}@cse.usf.edu

**Abstract**— Bloom filters are a well known data structure for approximate set membership. Bloom filters are space efficient, but require many independent hashes and consecutive memory accesses for an element test. In this paper, we develop a hash table data structure that stores string signatures in an array. This new Signature Array Hash Table (SAHT) supports faster element testing than a Bloom filter and requires less memory than a standard hash table that uses linked-list chains. The SAHT also supports removal of elements (which a Bloom filter does not) and addition of elements at the expense of requiring about 1.5x more memory than a Bloom filter with same false positive rate.

**Keywords**- Bloom filter, Signature Array Hash List, P2P

## I. INTRODUCTION

There are many applications where a large directory list of filenames (strings) has to be searched for a match. One such application is in web proxy caches (such as Summary Cache [2]). Another application is P2P file sharing where many files, or file fragments, are stored in each node. P2P nodes are queried by peer nodes for a particular file name. A node must be able to quickly respond to a query (e.g., with a query hit if the queried-for file is stored in the node). Our particular motivation is to make query handling very efficient to be able to execute it on a small outboard processor to support proxying. Proxying can reduce the energy use of P2P file sharing [3].

Common to file search applications are the need for 1) fast search for a string match, 2) low memory use (for storing the list of filename strings), and 3) the ability to add and remove filename strings. Probabilistic data structures for approximate set membership testing, such as the Bloom filter, have been investigated for these applications. Bloom filters have many applications in the area of networks where a small false positive outcome is tolerable [1]. Summary Cache [2] used Bloom filters. In this short paper we explore an alternative probabilistic data structure to Bloom filters. We call this new data structure the Signature Array Hash Table (SAHT) and show that it has many desirable properties and advantages.

## II. OVERVIEW OF BLOOM FILTERS

A Bloom filter is a probabilistic data structure that represents a set of elements (strings in our application) [1]. A Bloom filter consists of an array of  $m$  bits, initially all set to 0, used to represent  $n$  strings. A single hash function with  $k$

Funding acknowledgment will go here.

### mapStrings(list of $m$ strings)

1. (Clear *bloomArray*.)
  - (a) for  $i$  from 1 to  $m$  do  
    *bloomArray*[ $i$ ] = 0
2. (Set bits in *bloomArray*.)
  - (a) for  $i$  from 1 to  $n$  do  
    for  $j$  from 1 to  $k$  do  
        *index* = hash(*stringList*[ $i$ ],  $j$ ) mod  $m$   
        *bloomArray*[*index*] = 1

### testString(string *inString*)

1. (Test for matching  $k$  bits sets in *bloomArray*.)
  - (a) for  $i$  from 1 to  $k$  do  
        *index* = hash(*inString*,  $i$ ) mod  $m$   
        if (*bloomArray*[*index*] == 0)  
            return(FALSE)
  - (b) return(TRUE)

Figure 1. Bloom filter algorithms for add list of strings and test string

different initial values is used to map (or “store”) strings and test for membership. Figure 1 shows the two key algorithms for a Bloom filter: **mapStrings()** and **testString()**. In **mapStrings()**  $m$  strings are hashed  $k$  times each, and the hash values are used to identify bit locations in *bloomArray* to be set to 1. In **testString()** an *inString* is hashed  $k$  times and the  $k$  bit locations are tested. If all  $k$  locations are a 1, then probabilistically, the Bloom filter “contains” *inString*. Bloom filters have a probability of false positive of,

$$\Pr[\text{false positive}] = \left(1 - \left(1 - \frac{1}{m}\right)^{kn}\right)^k \approx \left(1 - e^{-kn/m}\right)^k. \quad (1)$$

The optimum  $k = \ln(2)(m/n)$ . With the optimum  $k$ , half of the bits in *bloomArray* are set to 1. For  $m/n = 16$  (i.e., 16 bits of *bloomArray* allocated for each mapped string),  $k = 11$ .

For each string test that results in a “hit”,  $k$  hashes and  $k$  memory access (or look-ups) are needed. On average, 2 hashes and memory accesses are needed for a string test that results in a “miss” (this is the mean of a geometric sequence when half the bits are set to 1). An element can easily be added to a Bloom filter, the algorithm **addString()** is not shown due to space restrictions. It is, however, not possible to remove elements from a Bloom filter without creating the possibility of a false negative. This is a major shortcoming of Bloom filters.

### III. DESCRIPTION OF THE SIGNATURE ARRAY HASH TABLE

Hash tables with linked-list chains can be used to store string signature values (where signatures are the hashes of the strings). Hash tables support easy addition and removal of values. However, hash tables require a significant amount of memory due to pointers that are stored with each value in the hash table chains. *To eliminate the memory needed for pointers, we have the idea to store hash chains in blocks in a signature array and use another array to contain offsets to the blocks.* The actual values stored are signatures of the strings. Thus, a false positive will occur if an *inString* hashes to the same offset location (i.e., chain) and same signature value as a non-matching stored string.

Figure 2 shows the two key algorithms for our Signature Array Hash Table (SAHT). The function **mapStrings()** maps  $m$  strings into the SAHT and uses a hash function to generate hash values, *hash1* and *hash2*, for each string. On exit, two arrays are defined and comprise the SAHT:

- *ptrArray* consisting of  $m$   $\langle sigOffset, blockLen \rangle$  pairs
- *sigArray* consisting of  $m$  *stringSig* values

where *sigOffset* is the offset into *sigArray* for a chain with index *hash1* modulo  $m$  and *blockLen* is the length of the chain stored in this block. A temporary array *tempArray* is defined within **mapStrings()**. For a 32-bit machine, we can define *hash1* and *hash2* to be 32 bits, *sigOffset* to be 24 bits, *chainLen* to be 8 bits, and *stringSig* to be 16 or 32 bits. The values of *sigOffset* and *chainLen* are derived from *hash1* and *hash2* in the function. A 24-bit *sigOffset* value limits  $m$  to be less than  $2^{24} = 16,777,216$  (i.e., no more than 16,777,216 strings can be mapped into the SAHT with the variables defined in size as they are here). The arrays *ptrArray* and *sigArray* are used in **testString()** to test for membership of an input string.

The probability of false positive of the SAHT can be computed as a balls-and-bins problem where the number of balls and bins are equal. For  $m$  strings and  $b$  bits in *stringSig*,

$$\Pr[\text{false positive}] = \sum_{i=1}^m \left( \binom{m}{i} \left( \frac{1}{m} \right)^i \left( 1 - \frac{1}{m} \right)^{m-i} \left( 1 - \left( 1 - \frac{1}{2^b} \right)^i \right) \right) \quad (2)$$

where the first binomial expression is the probability of a  $i = 1, 2, \dots, m$  strings mapping to a given block in *sigArray* and the second expression is the probability of the string having a unique *stringSig* in the chain at the given block and assumes that there may already be duplicate *stringSig* values in a chain. Equation (2) cannot be simplified into a closed form. However, if we assume that each chain contains only non-duplicated values, then the second expression is  $i/2^b$ . The modified equation (2) then simplifies to,

$$\Pr[\text{false positive}] = \frac{1}{2^b}. \quad (3)$$

In practice, we find that equations (2) and (3) compute to numerical values that are very close to the same.

#### mapStrings(list of $m$ strings)

1. (Create and sort *tempArray*.)
  - (a) for  $i$  from 1 to  $m$  do
    - $tempArray[i].hash1 = \text{hash}(stringList[i], 1) \bmod m$
    - $tempArray[i].hash2 = \text{hash}(stringList[i], 2)$
  - (b) sort *tempArray* by *hash1* values
2. (Create *sigArray*.)
  - (a) Copy all *tempArray*.*hash2* values to *sigArray*
3. (Create *ptrArray*.)
  - (a)  $i = j = count = sumCount = 0$
  - (b) while ( $i < m$ ) do
    - if ( $tempArray[i].hash1 == j$ )
      - increment  $i$
      - increment  $count$
    - else if ( $count == 0$ )
      - $ptrArray[j].sigOffset = 0$
      - $ptrArray[j].blockLen = 0$
      - increment  $j$
    - else
      - $ptrArray[j].sigOffset = sumCount$
      - $ptrArray[j].blockLen = count$
      - $sumCount = sumCount + count$
      - increment  $j$
      - $count = 0$
  - (c) if ( $count > 0$ )
    - $ptrArray[j].sigOffset = sumCount$
    - $ptrArray[j].blockLen = count$

#### testString(string *inString*)

1. (Generate *blockIndex* and *stringSig* values.)
  - (a)  $blockIndex = \text{hash}(inString, 1) \bmod m$
  - (b)  $stringSig = \text{hash}(inString, 2)$
2. (Test for *stringSig* since location *blockIndex*.)
  - (a)  $offset = ptrArray[blockIndex].sigOffset$
  - (b)  $len = ptrArray[blockIndex].blockLen$
  - (c) if ( $len > 0$ )
    - for  $i$  from  $offset$  to  $(offset + len)$ 
      - if ( $sigArray[i] == stringSig$ ) return(TRUE)
  - (d) return(FALSE)

Figure 2. SAHT algorithms for add list of strings and test string

We solve for the mean number of look-ups for the SAHT for the case of a miss and hit. Let  $U$  be a random variable for the number of look-ups. Then,

$$E_{miss}[U] = 1 + \sum_{i=1}^m j \left( \binom{m}{i} \left( \frac{1}{m} \right)^i \left( 1 - \frac{1}{m} \right)^{m-i} \right) = 1 + 1 = 2 \quad (4)$$

where the initial "1" is for the look-up in the *ptrTable* and the summation is the mean chain length. For  $E_{hit}[U]$  we condition on the probability of a block (or bin) being non-empty. Thus,

$$E_{hit}[U] = 1 + \frac{1}{(1 - \Pr[\text{chain length} = 0])} \quad (5)$$

where

$$\Pr[\text{chain length} = 0] = \left( 1 - \frac{1}{m} \right)^m \approx \frac{1}{e} \quad (6)$$

and thus,  $E_{hit}[U] = 2.582$  for large  $m$ .

In the SAHT, a mapped string can be removed by setting its mapped *sigArray* value to zero (this assumes that the hash function always returns a non-zero value). A new string can be added at a somewhat greater cost; *sigArray* needs to be shifted-down to create a space for a newly mapped-in *stringSig*. The algorithms **removeString()** and **addString()** are not shown due to space restrictions.

#### IV. NUMERICAL COMPARISON OF BLOOM FILTER AND SAHT

Using the equations in Sections II and III, we determine the memory trade-off between the SAHT and Bloom filter. The SAHT will require more memory for a given false positive rate. Table I shows the numerical results for an SAHT with 1 million strings mapped-in and  $b=16$  and  $b=32$ . For both cases, a Bloom filter with the closest possible false positive rate was selected. It can be seen that the Bloom filter with approximately matching false positive to an SAHT with  $b=16$  requires about 2.1x less memory (i.e., 2.875 million bytes compared to 6 million bytes) and for  $b=32$  requires about 1.4x less memory. However, the Bloom filters will require at least 6.2x and 12.4x, respectively, greater processing time to determine a hit as the SAHT (and about the same amount of processing time as the SAHT to determine a miss). This is computed as the Bloom filter  $k$  divided by the mean number of look-ups for a hit with an SAHT (which is  $E_{hit}[U] = 2.582$ ).

#### V. IMPLEMENTATION AND EXPERIMENTAL COMPARISON

We implemented a Bloom filter and SAHT in ANSI C to measure the mean number of look-ups (and actual processing time) for hit and miss. For the Bloom filter we used  $m/n = 23$  (similar to numerical analysis in the previous section). We used CRC32 for a hash function. We mapped-in 1 million strings of mean length 50 bytes and tested with 6 million strings of the same mean length. We used the gcc 3.4.5 compiler under Windows XP, on a Dell PC with a 3.2 GHz Pentium 4 and 1 GB of RAM. Table III shows the results. It can be seen that the mean number of look-ups closely match  $E_{miss}[U]$  and  $E_{hit}[U]$  from Section III. The measured CPU time also corresponds to the relative differences in look-up time for miss and hit. This shows that the SAHT requires predictably less processing time than a corresponding Bloom filter.

#### VI. RELATED WORK

A counting Bloom filter was first proposed in [2] as a means of supporting element removal. A counting Bloom filter uses multiple bits (typically 4) for each location. Thus, a 4-bit counting Bloom filter requires 4x the amount of memory. To reduce the amount of time spent in hashing in a Bloom filter, the use of a Random Number Generator (RNG) has been explored [4]. In this approach, one hash is made and then this hash value is used to seed the RNG to generate the  $k-1$  remaining “fake” hash values. This approach will result in higher false positive rates when the initial hash values are the same for different strings.

TABLE I. NUMERICAL RESULTS FOR SAHT

	Pr[false positive]	Memory required
$b = 16$	$1.53 \cdot 10^{-5}$	6 million bytes
$b = 32$	$2.33 \cdot 10^{-10}$	8 million bytes

TABLE II. NUMERICAL RESULTS FOR BLOOM FILTER

	Pr[false positive]	Memory required
$m/n = 23$ ( $k = 16$ )	$1.59 \cdot 10^{-5}$	2.875 million bytes
$m/n = 46$ ( $k = 32$ )	$2.52 \cdot 10^{-10}$	5.750 million bytes

TABLE III. EXPERIMENTAL RESULTS FOR SAHT AND BLOOM FILTER

	Number of look-ups		Time per look-up (ns)	
	SAHT	Bloom	SAHT	Bloom
All hits	2.50	16.00	668	4560
All misses	1.99	2.00	625	740

Our work is not the first to propose the use of an array to store hash chains to eliminate the need for pointers. In [5] the authors used a sparse array to save space by allocating space only for locations in use. Space is allocated as signatures are inserted. Signatures are pointed to by a second array used as a reference. However, it is not clear which data structure they used to implement it. Our work uses a dense array instead, which allocates space for the expected size of the set. We do not waste space since we know the set size from the beginning.

#### VII. SUMMARY AND FUTURE WORK

The Signature Array Hash List (SAHT) is a probabilistic data structure suitable for fast directory search applications. Compared to a Bloom filter, the SAHT requires more memory, but achieves a much faster search and allows for strings to be unmapped, or removed.

Future work includes a deeper analysis of the SAHT. We also plan to implement the SAHT in a proxy subsystem for P2P file search or other network application.

#### ACKNOWLEDGEMENT

The authors would like to thank Allen Roginsky from the National Institute of Standards and Technology for his help.

#### REFERENCES

- [1] A. Broder and M. Mitzenmacher, “Network Applications of Bloom Filters: A Survey,” *Internet Mathematics*, Vol. 1, No. 4, pp. 485-509, 2005.
- [2] L. Fan, P. Cao, J. Almeida, and A. Broder, “Summary Cache: A Scalable Wide-Area Web Cache Sharing Protocol,” *IEEE/ACM Transactions on Networking*, Vol. 8, No. 3, pp. 281-293, June 2000.
- [3] M. Jimeno and K. Christensen, “A Prototype Power Management Proxy for Gnutella Peer-to-Peer File Sharing,” *Proceedings of the IEEE Conference on Local Computer Networks*, pp. 210-212, October 2007.
- [4] A. Kirsch and M. Mitzenmacher, “Less Hashing, Same Performance: Building a Better Bloom Filter,” *Lecture Notes In Computer Science*; Vol. 4168, pp. 456-467, 2006.
- [5] J. Komlós, E. Szemerédi, “Storing a Sparse Table with O(1) Worst Case Access Time,” *Journal of the ACM*, Vol. 31, No. 3, pp. 538-544, July 1984.