Lockless Hash Tables with Low False Negatives

J. Ros-Giralt, A. Commike, R. Rotsted, P. Clancy, A. Johnson, R. Lethin
Reservoir Labs
New York, NY, US
{giralt, commike, rotsted, clancy, johnson, lethin}@reservoir.com

Abstract—**Hash tables are efficient storage data structures widely used in many types of high-performance computer-related problems. In their design, optimal trade-offs must be made to accommodate for the specific characteristics of the application. In this paper we present *lock-free low-false-negative* (LFN) tables, a family of hash tables designed to address one such type of trade-off. LFN tables sacrifice a low probability of false negatives and a very low (or negligible) probability of false positives to achieve higher performance access time in concurrent shared memory applications. LFM tables are structurally biased towards false negatives and therefore are more suitable for applications that tolerate better false negatives than positives. In this paper we provide a mathematical analysis of their performance and provide use cases where they can be deployed to eliminate shared memory access bottlenecks in the context of very high-speed computer networks.**[1]

I. Introduction

The origins of hash tables date back to the 1960s, when authors of compilers used the concept of hashing as a mechanism for maintaining lists of user-defined names and associated properties. These initial developments led later to the formalization of the classic hash-table techniques used today such as *separate chaining* and *open addressing* [1]. Since in many cases hash tables are more efficient than other storage data structures, they are widely used in many types of computer-related problems, including associative arrays, indexed databases, caches or sets.

When designing cache tables, one needs to look at the specific aspects of the application and identify trade-offs that can lead to optimal configurations. In this paper we present one such type of trade-off. In particular, we present a family of hash tables that allow concurrent access without requiring locks, at the expense of sacrificing a small probability of error. The proposed trade-off is therefore one between *performance* and *accuracy*. By avoiding locks, the table can be accessed concurrently without any memory contention. The probability of error is modeled in our analysis in terms of the probabilities of false negatives and positives, where a false negative is defined as the scenario in which the reading of an object previously stored in the table returns a null value, and a false positive corresponds to the scenario in which the reading of an object from the table returns a different object. More concretely, this paper shows that it is possible to design lock-free hash tables that allow concurrent access while sacrificing a small probability of false negatives and a very small (or negligible) probability of false positives. The bias towards false negatives is due to an asymmetry in the formulation of the problem, which will be formally explained and quantified in the analysis sections of this work. Such bias gives also the name of the proposed data structure: *lock-free low-false-negative (LFN) tables.*

[1] This work was funded in part by the US Department of Energy, contract number DE-SC0006343.

II. Background Work: Brief History of High-Performance Hash Tables

To set the ground of our work, we provide a summary of some of the key milestones produced in the context of high-performance hash tables. Four particular historical developments are of special interest: Bloom filters (1970s), d-left hashing (1990s), cache awareness (late 1990s), and lock-free algorithms (2000s).

Bloom Filters. In 1970, Burton Bloom introduced a new hashing technique to identify an element as a nonmember of a given set [2]. The technique—known as Bloom filters—allows for the reduction of the amount of space required to contain the set by trading off a small and controllable fraction of false positive errors. Bloom filters have also been used to optimize the performance of hash tables. For instance, in 2005, Kumar [7] and Song [8] independently applied bloom filters to reduce the number of memory accesses of a hash table.

D-Left Hashing. In 1994, Azar et al. introduced two-way chaining as a method to reduce with high probability the size of the fullest bucket in a chaining-based hash table [3]. By doing so, the maximum access time to the table is also reduced with high probability. This work provided the fundamental theory upon which d-left hashing tables would eventually be based. In 1999, Vocking observed that introducing asymmetry could help to further improve the performance of Azar's two-way chaining solution and referred to this new, improved method as the "always-go-left" algorithm [4]. The term d-left hashing was later introduced by Broder et al. who used it to refer to the generalization of Vocking's algorithm to the case of d-way chaining [5].

Cache Awareness. While most of the theoretic work on hash tables focuses on the algorithmic aspects of the problem, the specific details of the implementation can play as crucial a role in determining the bottom-line performance of the system. One important aspect of the implementation is the interaction between software and hardware resources and, specifically, the memory access patterns derived from stressing the hash table: to minimize expensive memory accesses, it is important that the system is properly designed or tuned to maximize cache hit ratios. Srinivasan et al. proposed early on (1998) the concept of designing hash data structures that fit in hardware cache lines [6]. For instance, in separate chaining, they argue that hash tables can be tuned to have a specific maximum number of collisions per entry that should be equal to the number of elements that fit into a cache line. By doing so, one can guarantee that a table operation (*put, get*, and *remove*) will only require at most one memory access. Broder et al. (2001) observed that d-left hashing can be used to tune the maximum number of collisions per table entry in order to address Srinivasan's problem of fitting a table operation into a single cache line—since the "d" parameter in d-left hashing has a direct relationship with the number of collisions [5]. Both Kumar and Song provided also in 2005 independent works for using Bloom filters as a preliminary membership check of an element [7, 8]. By keeping the Bloom filter in the cache, this technique reduces the number of memory accesses to zero for those *get* and *remove* operations invoked on elements that are not in the table.

Lock-Free Algorithms. Concurrent access to a hash table has also been the subject of intense research. Using compare and swap (CAS) atomic operations, Harris (2001) introduced the first correct lock-free algorithm for linked lists [9]. Later, a similar algorithm was used by Michael (2002) to provide a lock-free chaining-based hash table [10]. In 2006, Shalev et al. presented *split order* [11], a technique which provides another method to implement a lock-free chaining-based hash table; Shalev's method, in addition, allows for the growing of the hash table in a lock-free manner. All of these efforts are based on chaining hash tables. In 2008, Click [12] introduced a novel lock-free algorithm that works for open addressing hash tables.

III. LFN TABLES

A. *Problem Definition*

Suppose that we need to track the state *s* of an object *o,* for an arbitrary number of different objects, where *s* is expressed as an arbitrary data structure. Our objective is to design a high-performance storage system with two methods, *put(o, s)* and *get(o),* which are used to store and retrieve state *s* of object *o*, respectively. We assume that a *put()* operation can always successfully store the state of an object into the storage system, whereas a *get()* returns *null* if the state of the passed object cannot be found in the storage system. In order to achieve higher degrees of performance, we will be willing to tolerate a small amount of inaccuracy. To this end, we introduce the concepts of false negatives and positives within the premises of the storage system as follows:

False negatives and positives. Assume that we call *get(o)* after invoking *put(o, s)*. We will say that the get operation generates a false negative if it returns a *null* value. Likewise, we will say that a get operation generates a false positive if $s \neq$ *get(o)* and *get(o)* is not *null*.

Therefore, a false negative corresponds to a situation in which after storing the state of an object, attempting to retrieve such state results into an empty (*null*) value, as if the object had never been stored. On the other hand, a false positive corresponds to a situation in which attempting to retrieve the state of an object results into the state of another object.

Consider a storage system with the following properties:

1. *Concurrency*: multiple invocations of the methods *put()* and *get()* can be concurrently issued;
2. *Lock-free*: no locks can be used to resolve correct access to the storage system;
3. *Accuracy:* we can tolerate false negatives with a probability $p_{fn} \ll 1$ and false positives with a probability $p_{fp} \ll p_{fn}$.

We will refer to storage systems that satisfy the above properties as *lock-free low-false-negative tables* or simply as *LFN tables* (colliding the acronym LFLFN).

We notice that LFN tables are storage systems that accept a low probability of false negatives but are much less tolerant to false positives. The idea is that in their design, one should aim for a probability of false positives that is negligible or zero, making a low probability of false negatives the only practical trade-off in the design. Hence the name LFN.

The first property speaks for the nature of the system, which aims at allowing concurrent access. The second and third property can be interrelated as follows. The second property is a constraint that eliminates from the design any possible contention produced by a concurrent access to the storage system. This is a property tailored at achieving higher degrees of performance, since the elimination of locks implies the lack of any performance penalty as a result of allowing concurrent access. Because there cannot be a "free lunch," we need to trade off this property with some other positive quality, in this case accuracy. That is introduced into the design via the third property. Therefore, properties two and three are two faces of the same coin, in which a bit of accuracy is traded off by a higher degree of performance.

The above specifications define a family of problems that can generally be found in a variety of fields. For instance, in the area of computer networks, one often needs to track the status of connections using binary flags. Such is the case of a firewall application that needs to make accept-or-drop decisions on a per-connection basis; or flow-control network protocols that take different actions based on whether a connection is marked as congested or not congested. The asymmetric constraint $p_{fp} \ll p_{fn} \ll 1$ can also be understood in the context of problems where costs are biased toward a certain region of the solution set. For instance, in the firewall example, the cost of accepting a malicious connection that exploits a fatal cyber-security vulnerability (a false positive situation) could be potentially much larger than the cost of performing additional due diligence on a benevolent connection (a false negative situation).

To simplify our initial analysis, we will start assuming that *s* is of binary type. That is, that the state *s* of each object *o* can be represented by a binary flag (e.g., states such as "on" or "off," "black" or "white," "up" or "down," "allowed" or "rejected," etc.). Later in the paper we will generalize the solution to accept object states of arbitrary data structure types.

B. *LFN Algorithm for Binary Object States*

In this section, we introduce an LFN table that can be used to store binary object states. A typical application of this type of tables can be found in the construction of sets storing objects: if an object's state is *true*, it is stored in the table;

otherwise, it is left outside. This type of tables covers similar types of applications as those addressed by *Bloom filters* [2].

Let o be an arbitrary object whose state s can take two values, true and false; $h_x(\,)$ a hash function that takes an arbitrary input and returns as output an integer between zero and *x-1*; and T[ ] a one-dimensional table with capacity to store n values. We will call this table the LFN-b table ("b" for binary).

The *put* and *get* methods are as follows:

```
Initial state: $T[e] = NULL$ for all e such that $0 \leq e < n$;
put(o, s):
  If s == TRUE:
    $T[h_n(o)] = h_l(o)$;
get(o):
  If $T[h_n(o)] == h_l(o)$:
    Return TRUE;
  Otherwise:
    Return NULL;
```

Lemma 1: LFN-b accuracy. An element in a data structure constructed using the LFN-b algorithm is in a positive state with probability p_t, in a false negative state with probability p_{fn}, and in a false positive state with probability p_{fp}, where $p_{fp} \ll p_{fn} \ll p_t$ and $p_t \approx 1$.

Proof. Let e be an entry in the LFN-b table. To prove the lemma, we will explore all the potential states element e can possibly take along with the probability of such states. Element e starts with a *null* value, $T[e] = NULL$. Assume that there exists an object *o* with a *true* binary value such that $\boldsymbol{h_n(o)} = e$. Then according to the *put()* algorithm, we have that $T[e] = h_l(o)$. This state is a true state as entry e stores the binary value true.

Let's also assume that the assignment $T[h_n(o)] = h_l(o)$ in the *put()* algorithm can be atomically executed. Now there exist only two events that can affect the state of entry $T[e]$, each with a different probability of occurrence. These two events are as follows:

- With a probability $p_{t,fn}$, there is another object o' with a *true* value such that $h_n(o') = e$ and $h_l(o') \neq h_l(o)$. Then, we have that $T[e] = h_l(o') \neq h_l(o)$. This state generates a false negative because $get(o)$ returns *null.*
- With a probability $p_{t,fp}$, there is another object o'' with a *false* value such that $h_n(o'') = e$ and $h_l(o'') = h_l(o)$. Then, we have that $T[e] = h_l(o'') = h_l(o)$. This state generates a false positive because $get(o'')$ returns *true.*

A third and last scenario is one in which none of the two events above occur; in this case, the state of $T[e]$ does not change. We will assume this happens with a probability $p_{t,t}$, which by construction, must be such that $1 = p_{t,t} + p_{t,fn} + p_{t,fp}$.

Based on the above, we observe:

- $p_{t,fn} \ll 1$, because $p_{t,fn}$ corresponds to the collision probability of the hash function $h_n(\ \ )$. While this probability is not negligible, it is by construction of the hash function much smaller than 1.
- $p_{t,fp} \ll p_{t,fn}$, because $p_{t,fp}$ corresponds to the collision probability of the hash function $h_l(\ \ )$ which is by construction much smaller than the collision probability of the hash function $h_n(\ \ )$—since in practical scenarios the size of the table *n* is much smaller than the size of the hash value space *l*.

Since $1 = p_{t,t} + p_{t,fn} + p_{t,fp}$, it must be that $p_{t,t} \approx 1$.

In our notation, $p_{x,y}$ can be understood as Bayesian probabilities representing the odds that a table entry goes into state *y* assuming that its current state is *x*, where each state can be one of three possibilities: true (t), false negative (fn), or false positive (fp). If we assume that the table is populated once for all available objects and not modified anymore (i.e., excluding the possibility of existing elements to leave the table and of new elements to enter it), then we have that the final state probabilities of each table entry are equal to the Bayesian probabilities:

$$\begin{aligned} p_t &= p_{t,t} \\ p_{fn} &= p_{t,fn} \\ p_{fp} &= p_{t,fp} \end{aligned} \tag{1}$$

proving the lemma.

§

In the Appendix we present a generalization of this proof showing that the lemma also holds when objects are allowed to enter and leave the storage system at any time, as is the case of real time applications where the objects are streamed.

Lemma 2: LFN-b cost. The computational cost of the LFN-b algorithm for both the *put()* and the *get()* operations is

$$O(1). \tag{2}$$

The amount of storage required to implement the table is

$$O(n \cdot log(l)). \tag{3}$$

Proof. The computational cost of *put()* and *get()* is trivially $O(1)$ since both require just one single access operation to the table. The storage cost is equal to the size of the table *n* times the amount of bytes stored in each table entry. This latter number corresponds to the minimum number of bits required to encode a space with *l* elements, which is $log(l)$.

§

The two lemmas combined give us another interpretation as to the asymmetries of the problem and the reason why the probability of false positives can be reduced much further than the probability of false negatives. The accuracy lemma shows that the probability of an entry in the table being in a false negative state is equivalent to the collision probability of a hash function operating on a space with *n* elements; similarly, the probability of being in a false positive state is equivalent to the collision probability of a hash function operating on a space with l elements. On the other hand, the cost lemma shows us that the storage requirements are $O(n \cdot log(l))$. Therefore, given a certain fixed amount of storage space, designs aiming at optimal performance will tend to be biased towards solutions in which l is much larger than n.

In the next section, we quantify the performance of the LFM algorithm to help identify appropriate values for n and l.

C. *Sizing LFM-b for Performance and Accuracy*

Let k be the number of objects stored in a LFM-b table tuned with parameters n and l. We observe that the expected number of objects that are in a false negative or positive state is equivalent to the expected number of ball collisions occurred when throwing k balls into n or l bins, respectively. This is a well-known result in combinatorics—it is also commonly known as the *birthday paradox* [1]—and results into the following mathematical expression:

Let X be the number of ball collisions occurred when throwing k balls into b bins, then the expected value of X is

$$\mathrm{E[X]} = \frac{1}{b}\binom{k}{2}. \tag{4}$$

Using the above result, the expected number of false negatives in a LFM-b table populated with k objects over the total number of objects is

$$\mathrm{E[FN]} = \frac{1}{n \cdot k}\binom{k}{2} = \frac{k-1}{2n}. \tag{5}$$

and the ratio of expected false positives over the total number of objects is

$$\mathrm{E[FP]} = \frac{1}{l \cdot k}\binom{k}{2} = \frac{k-1}{2l}. \tag{6}$$

Fig. 1 and Fig. 2 plot expressions (5) and (6) for various values of k, n and l allowing us to cherry pick some concrete data points. For instance, storing one thousand objects ($k = 1000$) in a table of 1 milion entries ($n = 10^6$) and with $l = 2^{64}$(that is, a total memory size of 8 million bytes), we have that the chances of an element in the table to be in a false negative or positive state are $5 \cdot 10^{-4}$ and $2.7 \cdot 10^{-17}$, respectively. Or to accommodate for 1 million objects ($k = 10^6$), one would need to employ a table of 80 million bytes ($n = 10^7$ and $l = 2^{64}$) in order to achieve a false negative rate of $5 \cdot 10^{-2}$ and a false positive rate of $2.7 \cdot 10^{-14}$.

Before concluding the analysis of the binary case, we note that an important assumption of Lemma 1 is the atomicity of assignment $T[h_n(o)] = h_l(o)$ in the *put()* algorithm. This is needed in order to ensure that the value of any entry $T[e]$ is not corrupted by the concurrent access of various *put*() operations executed in parallel. We observe that this assumption is easy to guarantee as most modern architectures perform integer (both 32 and 64 bit) writes in an atomic manner. When generalizing this algorithm to support arbitrary data structures, we need to be careful to ensure that the equivalent write operation is also performed atomically, as explained in the next section.

D. *LFN Algorithm for Arbitray Object States*

We will now consider the case of objects that can take states represented by arbitrary data structures. Let o be an object with a state s, $h_x()$ a hash function that takes an arbitrary input and returns as output an integer between zero and x-1, and T[ ] a one-dimensional table with capacity to store n values, where these values can be of any arbitrary data type. We will call this table the *LFN-g table* ('g' for general).

The *put* and *get* methods are as follows:

```
Initial state: $T[e] = NULL$ for all e such that $0 \leq e < n$;
put(o, s):
    $T[h_n(o)].state = s$;
    $T[h_n(o)].hash = h_l(o, s)$;
get(o):
    If $T[h_n(o)].hash == h_l(o, T[h_n(o)].state)$:
        Return $T[h_n(o)].state$;
    Otherwise:
        Return NULL;
```

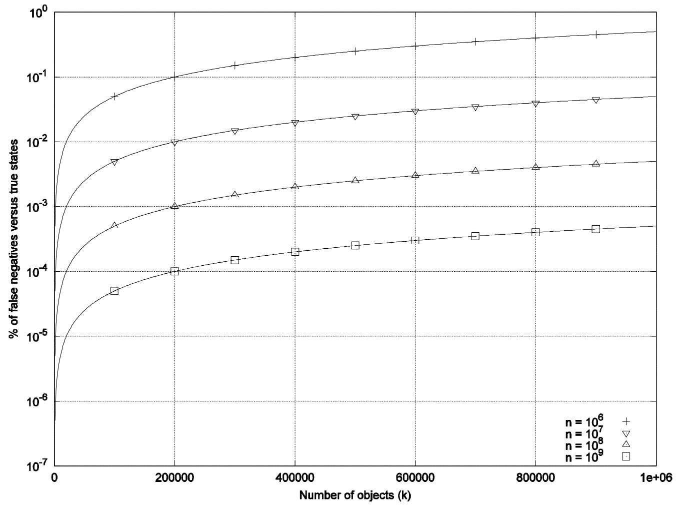

Fig. 1. Chances that an entry in the LFM-b table is in a false negative state

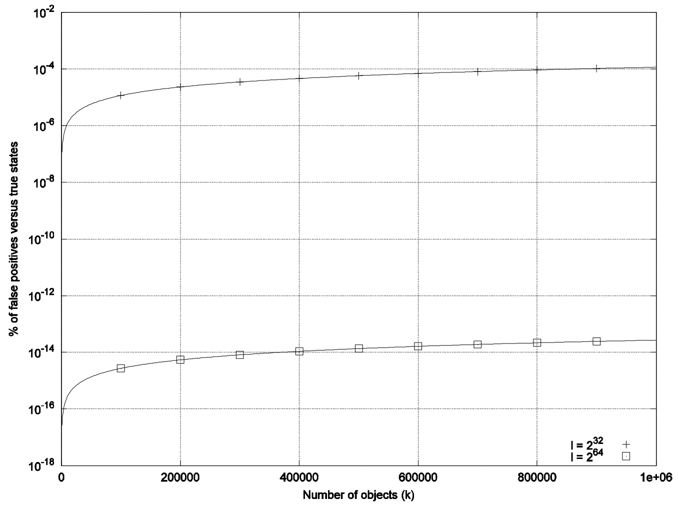

Fig. 2. Chances that an entry in the LFM-b table is in a false positive state

Lemma 3: LFN-g accuracy. An element in a data structure constructed using the LFN-g algorithm is in a positive state with probability p_t, in a false negative state with probability p_{fn}, and in a false positive state with probability p_{fp}, where $p_{fp} \ll p_{fn} \ll p_t$ and $p_t \approx 1$.

Proof. Let o and o' be two different objects with states s and s'. We observe that the LFM-g and the LFM-b algorithms employ the same two-level hashing strategy by using the functions $h_n(\,)$ and $h_l(\,)$. Following the arguments made in the proof of Lemma 1, we have that in the LFM-g algorithm a false negative event occurs when $h_n(o) = h_n(o')$ and $h_l(o', s') \neq h_l(o, s')$, while a false positive event occurs when $h_n(o) = h_n(o')$ and $h_l(o', s') = h_l(o, s')$. Since l will be by construction much larger than *n,* we have that $p_{fp} \ll p_{fn} \ll p_t$ and therefore $p_t \approx 1$.

§

Lemma 4: LFN-g cost. The computational cost of the LFN-b algorithm for both the *put()* and the *get()* operations is

$$O(1). \tag{7}$$

The amount of storage required to implement the table is

$$O\left(n \cdot (\log(l) + |s|)\right), \tag{8}$$

where $|s|$ is the number of bits needed to encode the state of each object in the table.

Proof. Following the same reasoning used in the proof of Lemma 2, the above lemma holds too.

§

From a performance perspective, the LFM-g table has the same structure as the LFM-b table: two hash functions $h_n(\,)$ and $h_l(\,)$ are used to control the expected number of false negatives and positives, respectively. The LFN-g represents a generalization of the LFM-b in the sense that the function $h_l(\,)$ uses both the object and its state to generate a unique hash value among the set of integers ranging from 0 to $2^l - 1$. By including the state of the object into this computation, the algorithm can reduce false positives for the case of arbitrary state values down to the probability of two hash values colliding in a space of 2^l values, just like in the case of the LFM-b table. As in the binary case, since the operation $T[h_n(o)].hash = h_l(o, s)$—i.e., the storing of an integer value into memory—can be executed atomically, the value $T[h_n(o)].hash$ corresponds always to a valid object.

IV. A Note on Applications

We have implemented the algorithms presented in this paper in the context of a high-performance network appliance called R-Scope [13]. The specific application is in the field of very high-speed, real-time network analysis, targeting rates up to a few 100Gbps or even Tbps. The appliance is connected as a tap into an IP network and receives a copy of the traffic being transmitted. For each packet it receives, the appliance runs a set of analytics programmed in a high-level specification language called Bro [14] to help detect events of interest within the network. Because the type of sophisticated analytics run by Bro are very computationally demanding, providing a high-performance networking layer to support Bro is crucial in order to sustain the targeted rates. Further, R-Scope runs on a manycore platform running up to 144 core workers; hence efficient parallel access to shared memory data structures is essential. This motivates the design of shared data structures that require no locking to avoid prohibitive concurrent memory access contention in the manycore architecture.

To achieve scale, R-Scope implements a *shunting cache* which stores flows that can be dropped from the analysis. The rationale is that a large number of connections (or a large number of packets within a connection) may become no longer interesting to the analysis. For instance, the analytics may decide that a certain *elephant* flow (a large high-bandwidth flow) is no longer interesting and solicit the networking layer to no longer forward packets of that flow. Such type of data shunting can dramatically reduce the amount of compute required by the analyzer workers without reducing their effectiveness.

The shunting cache can be implemented as a hash table, but as shown in this paper, to avoid using locks, sacrifices must be made in terms of the false negative and positive rates of the table. The shunting problem provides an interesting asymmetry: the risk costs of mistakenly not shunting a connection that should be shunted (a false negative) are much smaller than the risk costs of shunting a connection that should not be shunted (a false positive). A typical scenario is one in which the analytics concern with cyber security, because the cost of erroneously shunting a malicious connection from the network analyzers can be very high. This asymmetry is precisely the type of trade-off the LFM tables exploit. For a more detailed description of how the R-Scope system works and how an LFM table is used to implement the shunting cache, see [13].

A shunting cache can be understood as a set (rather than a table) that stores flows, and therefore it could also be implemented using other high-performance data structures like Bloom filters. Bloom filters provide excellent performance and space efficiency, but they would have the following limitations for the above-mentioned use case:

- Bloom filters have no false negatives but do have false positives. The shunting flow problem however tolerates better false negatives than positives.
- Bloom filters require a considerable number of hash functions, which could impose additional performance penalties in the *get()* and *put()* operations.
- Bloom filters cannot be generalized to support arbitrary object states. As an example, in our newer implementation of the shunting table, we have implemented the LFM-g algorithm so that other per-connection metrics (such as byte count) can be tracked.

V. Conclusions

When it comes to designing high-performance hash tables, trade-off decisions must be made. One classic approach is to trade a bit of accuracy in exchange of higher performance. In this paper, we present LFM tables, a family of hash tables that are based on such type of trade-off. By allowing the system to tolerate a small probability of false negatives and a very small (or negligible) probability of false positives, the tables can be implemented lock free. We show that LFM tables can be tuned using two parameters, *n* and *l,* each controlling one of the two hash functions employed by the algorithm to target a specific level of accuracy—expressed in terms of the rate of false negative and positives produced by the table. Towards their design and implementation, we provide closed mathematical expressions for *n* and *l* as a function of the targeted accuracy. LFM tables are suitable for applications that are more tolerant of false negatives than positives, which can be seen as the counterpart of the problem set addressed by Bloom filters. Although Bloom filters are more space efficient, LFM tables can also address the more general problem of storing arbitrary data structures into a hash table.

APPENDIX. BAYESIAN PROBABILITIES IN NON-STATIC REGIMES.

The proof for Lemma 1 assumes that once all objects are stored in the table, they are never removed from it and that no new objects are inserted in it again. In what follows, we provide the sketch of a proof for the general case in which objects are allowed to enter and leave the table at any time and perpetually.

Allowing for objects to dynamically enter and leave the table, the set of possible transitions between the three possible states (true, false negative and false positive) is presented in Fig. 3. The figure provides a simplified model in that there ought to be further possible false negative and positive states coming out from the two double-circle states. However, the probability of reaching such states diminishes geometrically and, for the sake of simplifying the analysis, we will consider them negligible.

For each state, the sum of the probabilities of all possible events must add up to one:

$$\begin{cases} 1 = p_{t,t} + p_{t,fn} + p_{t,fp} \\ 1 = p_{fn,fn} + p_{fn,t} \\ 1 = p_{fp,fp} + p_{fp,t} \end{cases} \tag{9}$$

The probabilities of going into each of the states can be calculated using the law of total probability as follows:

$$\begin{cases} p_t = p_t\, p_{t,t} + p_{fn}\, p_{fn,t} + p_{fp}\, p_{fp,t} \\ p_{fn} = p_t\, p_{t,fn} + p_{fn}\, p_{fn,fn} \\ p_{fp} = p_t\, p_{t,fp} + p_{fp}\, p_{fp,fp} \end{cases} \tag{10}$$

and from the proof in Lemma 1, we know that:

$$1 = p_t + p_{fn} + p_{fp} \tag{11}$$

Through mathematical manipulation of the above equations (which we leave as an exercise to the reader), we have:

$$\begin{aligned} p_{fp} &\approx \frac{p_{t,fp}}{(1-p_{fp,fp})} \\ p_{fn} &\approx \frac{p_{t,fn}}{(1-p_{fn,fn})} \end{aligned} \tag{12}$$

and since $p_{t,fp} \ll p_{t,fn} \ll p_{t,t}$, and by symmetry $p_{fn,fn} = p_{fp,fp}$, we have that $p_{fp} \ll p_{fn} \ll p_t$.

§

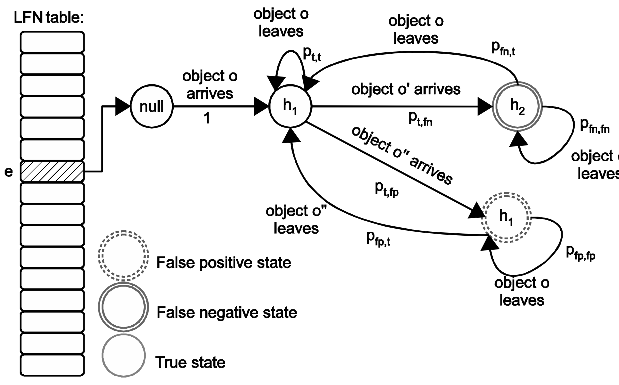

Fig. 3. Life cycle of an entry in the LFN table

REFERENCES

[1] T. H. Cormen, C. E. Leiserson, R. L. Rivest, C. Stein, "Introduction to Algorithms," pp. 253-285, third edition, The MIT Press 2009.

[2] H. Bloom, "Space/Time Trade-offs in Hash Coding with Allowable Errors," Communications of the ACM, Vol. 13, Num. 7, July 1970.

[3] Y. Azar, A. Z. Broder, A. R. Karlin, E. Upfal, "Balanced Allocations," Symposium on Theory of Computing, Montréal, Québec, Canada, May 1994.

[4] B. Vocking, "How Asymmetry Helps Load Balancing," Symposium on Foundations of Computer Science, New York City, N.Y., USA, October 1999.

[5] A. Broder, M. Mitzenmacher, "Using Multiple Hash Functions to Improve IP Lookups," Proceedings of INFOCOM, Anchorage, Alaska, USA, April 2001.

[6] V. Srinivasan and G. Varghese, "Fast Address Lookups Using Controlled Prefix Expansion," ACM Transactions on Computer Systems, Vol. 17, Issue 1, pp. 1-40, 1999.

[7] Sailesh Kumar, Patrick Crowley, "Segmented Hash: An Efficient Hash Table Implementation for High Performance Networking Subsystems," Symposium on Architectures for Networking and Communications Systems (ANCS). Princeton, N.J., USA October, 2005.

[8] Haoyu Song, Sarang Dharmapurikar, Jonathan Turner, John Lockwood, "Fast hash table lookup using extended bloom filter: an aid to network processing," ACM SIGCOMM Computer Communication Review archive, Vol. 35 , Issue 4 , October 2005.

[9] T. Harris, "A Pragmatic Implementation of Non-Blocking Linked-Lists," International Conference on Distributed Computing Systems, Phoenix, Arizona, USA, April 2001.

[10] M. Michael, "High Performance Dynamic Lock-Free Hash Tables and List-Based Sets," ACM Symposium on Parallel Algorithms and Architectures, Winnipeg, Manitoba, Canada, August, 2002.

[11] O. Shalev, N. Shavit, "Split-Ordered Lists: Lock-Free Extensible Hash Tables," Journal of the ACM, Vol. 53, Issue 3, May 2006.

[12] C. Click, "A Lock-Free Wait-Free Hash Table", work presented as invited speaker at Stanford, May 2008.

[13] J. Ros-Giralt, B. Rotsted, A. Commike, "Overcoming Performance Collapse for 100Gbps Cyber Security," ACM Changing Landscapes in HPC Security 2013, New York City, NY, USA 2013.

[14] V. Paxson, Bro: A System for Detecting Network Intruders in Real-Time, Computer Networks, 31(23-24), pp. 2435-2463, 14 Dec. 1999.