Selection Optimization of Bloom Filter-Based Index Services in Ubiquitous Embedded Systems

Zhu Wang[1](✉), Chenxi Luo[2], and Tiejian Luo[3]

[1] Xingtang Telecommunications Technology Co., Ltd., Beijing, China
wangzhu09@mails.ucas.ac.cn

[2] Institute of Software Chinese Academy of Sciences, Beijing, China
luochenxi@iscas.ac.cn

[3] University of Chinese Academy of Sciences, Beijing, China
tjluo@ucas.ac.cn

Abstract. In pervasive systems, data object is stored in distributed storage nodes. High performance indexing service plays an import rule in the efficient utilization of the data in ubiquitous computing. The embedded systems on the ubiquitous nodes, however, have constraint memory space and energy supply. How to design efficient index service with limited resource requirement on the embedded systems is a key technique in pervasive computing. In this paper, we compare two types of Bloom filter-based index services: Lightweight Bloom filter Array and Two-tier Bloom filter Array. The lookup time and the energy consumption are taken into consideration when measuring the performance of the two index services. We analyse the characteristics of the two algorithms with the analytical expressions. Further, experiments under the same conditions are performed and the results are analyzed. Finally, this paper gives the optimization suggestion for selecting one out of the two algorithms under different usage circumstances.

1 Introduction

Pervasive and ubiquitous systems have been adopted in many applications such as environment monitoring [1], sea depth measurement [2] and human behavior study [3]. Pervasive nodes, which are based on embedded systems, are usually short in memory and electricity supply. Therefore, the computation time and space complexity of such systems have to be taken into serious consideration. The ubiquitous system consists of a large volume of nodes. When user access arrives, the first step is to locate resident node of the wanted resource or service. Therefore, the indexing service, which is capable of representing the objects[1] stored on the nodes and performing lookup process, is a key component in the content management of the system. The performance of the embedded index service affects the system response time, node lifetime and system scalability.

[1] In this paper, we interchangeably use "object" and "item".

© Springer International Publishing AG, part of Springer Nature 2018
H. Jin et al. (Eds.): ICWS 2018, LNCS 10966, pp. 231–245, 2018.
https://doi.org/10.1007/978-3-319-94289-6_15

Usually in the pervasive systems, the data objects are stored in the ubiquitous nodes, while the index service is deployed on an index node. Take the wireless sensor network [4] for example. Sensor nodes are responsible of collecting data from the environment and storing the data. The sink node is responsible of gathering data from the sensor nodes and building index. The ubiquitous nodes are deployed in environment isolated from effective management. The embedded systems on the nodes are limited in memory and energy supply. On the other hand, the index node has much more energy supply (sometimes even unlimited) than the ubiquitous nodes, and the hardware configuration is also better. Index service on such ubiquitous embedded systems has to take those conditions into consideration.

State-of-the-art indexing services include table-based index, hash-based approach and Bloom filter-based approach. Table-based index costs too much space in the index node, which can be unacceptable when storage node number is very large. Hash-based approach can have low time complexity. However, it puts limitation on item placement and system scalability. Bloom filter [5] is a space-efficient probabilistic data structure for item representation and lookup in a set. When indexing space is limited, i.e. in the memory, the data structure offers fast item lookup with a low false positive rate. Many systems that emphasize time efficiency are using Bloom filters as their indexing technique when a small false positive rate is tolerable [6]. The low time complexity results in small energy consumption in the index building and item lookup process, as well as fast response rate. The space-efficiency characteristic enables the data structure to be deployed in the embedded devices with limited on-chip resource. In our paper we try to adopt the Bloom filter algorithm in the indexing of pervasive embedded systems.

In this paper, we show two Bloom filter based index algorithms. Both algorithms fit the condition of limited space and energy on the embedded nodes. We want to find under certain environment settings, which of the two algorithms performs better. We use both theoretical analysis and experiments to find the answer.

The rest of the paper is organized as follows. Section 2 describes the related work of our research. In Sect. 3, we give the considerations when designing index for ubiquitous embedded systems. Then, the algorithm and the theoretical analysis of LBA and 2TBA performance is shown in Sects. 4 and 5. Experimental comparison of the two algorithms is presented in Sect. 6. Finally, we conclude the paper in Sect. 7.

2 Related Work

Here we list the previous work of other researchers. Their contributions are also the foundations of our work.

2.1 Bloom Filter

Bloom filter [5] works as an index which records all elements of a set. We may assume that the set $S = \{x_1, x_2, ..., x_n\}$, which consists of n elements. A Bloom

Filter vector (BFV), which consists of m bits, is used to represent elements of set S. All bits of the vector are set to zero initially. For each element, the algorithm uses k hash functions $\{h_i\}_{i=1...k}$ to map the element onto k positions of the vector and sets the bit on the position to 1. The k functions, ranging from 1 to m, are independent from each other and can map elements of the set S to a random place on the vector. During the insertion period, the algorithm maps all elements of the set to load the BFV with all the information of the elements.

In lookup procedure which we want to check whether an element y belongs to the set S, the algorithm uses the same hash functions to map y onto k locations and check whether all $h_i(y)$ equal to 1. If the answer is no, we conclude that y doesn't belong to S, otherwise, we say y belongs to S. The time complexity of Bloom filter lookup is O(C).

It needs to be mentioned that there is a probability that elements don't belong to S be judged as inside S by BF. That is to say, BF has a false positive rate. Research [7] shows that the false positive rate can be represented as follows:

$$f_{FP} = (1 - e^{-\frac{kn}{m}})^k \tag{1}$$

Study [7] also shows that f_{FP} reaches minimal value when

$$k = \frac{m}{n} ln2 \tag{2}$$

Then the false positive is minimized

$$f_{FP} = 0.6185^{\frac{m}{n}} \tag{3}$$

Due to its simple structure and smooth integration characteristic, the mathematical format allows considerable potential improvement for system designers to develop new variations for their identical application requirements. Counting Bloom filters [8–10] can be used to improve network router performance [11]. Other variations are adopted in state machines [12], IP trace back [13], Internet video [14], distributed storage system index [15] and publish/subscribe networks [16].

In pervasive computing area, there are also Bloom filter applications. Research [17] and [18] use Bloom filter to represent items or policies during communication process. The algorithm can also serve as an efficient content management component like [19–21]. The Bloom filters have very compact size, which fit well in embedded systems with limited memory space. Also, the lookup time complexity is O(C), and thus saves energy consumption, especially suitable for pervasive embedded nodes with constraint energy supply.

2.2 Pure Bloom Filter Array for Data Storage Index

In this paper, we show variations of Bloom filter based algorithms. We first introduce the Pure Bloom filter Array (PBA) [22] approach, which is also a comparison of our work.

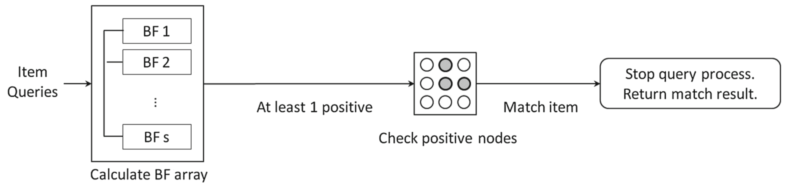

Fig. 1. Lookup procedure of PBA.

Many distributed systems use Pure Bloom filter Array to support item index and lookup. The approach consists of a two-stage process: indexing building and item locating.

Index Building. For each node of the system, the index node builds a Bloom filter for representing all of its items. These Bloom filters are loaded with all the items in the entire system and can act as an indexing system.

Item Locating. The object locating process is described below: when a query for a certain item arrives on the index node, the node first uses the Bloom filters to find the approximate membership relations: it calculates with the Bloom filter of each node and collects the results. The negative result of a certain Bloom filter means that the queried item doesn't exist on the related node. The positive result means that the queried item exists on the node with a probability of $1 - f_{FP}$. Then the system queries the actual node whose Bloom filter check result is positive to check whether the queried item exists in the node. In that way, the false positive occurrence is finally eliminated. Since the Bloom filters have an O(C) time complexity, the method can reduce lookup time remarkably. The item locating algorithm is shown in Fig. 1.

3 Considerations for Item Indexing in Pervasive Embedded Systems

The embedded devices are short in both computing resource and electricity. So in the design of the index algorithm on the node, we need to take both space efficiency and time complexity into consideration (longer time needed for calculation leads to more energy cost).

We give our considerations when designing the index model of pervasive embedded systems. To accelerate the lookup performance, each ubiquitous device generates an index of the objects on its own. Because of the limited energy on the embedded environment, the nodes transmit only a lightweight index of all its items, instead of the entire item list. That reduces the transmission time and hence saves energy. The lookup process, which needs intensive computing, takes place on the index nodes. On finding several candidates that may contain a wanted item, the index node notifies the ubiquitous device to check for the item on the nodes.

In the choosing of indexing algorithm, the candidates are listed in Sect. 2: table-based approach, hash-based approach, Bloom filter and perfect hashing. Table-based approach gives exact answer to queries on where the wanted item is. However, the time complexity of the approach is very high. To make things worse, the index of the table-based approach even exceeds the original data in size, which will cause the transmission energy cost to be very high. Hashes can locate the item by calculating the hash position. The method offers a fast way for indexing objects. However, that high indexing performance is based on the precondition that the candidate places of objects be determined by the hash calculation. The item cannot be moved out of those places. Perfect hashing can only deal with static sets. Those two algorithms do not fit in the ubiquitous environment. At last, we choose Bloom filter as an index of the system. The algorithm has low space and time complexity. It can still achieve high performance when calculation resource is limited.

In algorithms like Pure Bloom filter Array, the Bloom filter calculation takes place on either the device or the index node. The time consumption is rather low considering the O(C) time complexity of Bloom filters, and hence the energy consumption is quite low. On the other hand, the index node usually has better hardware configuration than the ubiquitous nodes, and also better electricity supply. We do not have to take the energy consumption on the index node into consideration. Therefore, the vast majority of time and energy cost comes from the node lookup process, in which the index node communicates with the ubiquitous device and the node checks for queried items in its disk. In the many checks in nodes, only one of the checking procedures can find the needed item. The rest nodes checking end without a match and waste a lot of time and energy on the ubiquitous device. Those redundant (false) checking times are the key factor that lowers system performance and increases node energy cost. The occurrence that the system looks up a query in a wrong node and finds no result is called the *false checking* in nodes. Interestingly, we find reducing the false checking times in nodes is in accordance with decreasing lookup time cost (thus improving system performance) and reducing energy consumption on ubiquitous nodes (thus prolonging nodes' life). That conclusion also holds in other Bloom filter based index algorithm, like Lightweight Bloom filter Array (LBA) and Two-tier Bloom filter Array (2TBA), which use the same mechanism for index lookup and node lookup. In the following sections, we use the false checking times as the key indicator of the performance of the algorithms.

4 Lightweight Bloom Filter Array (LBA)

Pure Bloom filter Array allocates the same index space for each item. However, in the observation of the access frequency of the data objects in the Internet, people find that in most applications, a small part of data objects attract the majority of data access. That is to say, there are "hot" items and "cold" items on the Internet. That phenomenon inspires people to be selective in item insertion process when index space is limited.

The LBA index establishes an index data structure for each ubiquitous node. Unlike the Pure Bloom filter Array algorithm, LBA does not insert all the elements of one ubiquitous node to its Bloom filter. On the contrary, it only inserts the popular items. The corresponding Bloom filters are gathered by the index node from the ubiquitous nodes to form an array, called the Lightweight Bloom Filter Array (LBA), which are loaded with only the “hot” items. By reducing the item number stored in the Bloom filters, the algorithm lowers the false positive rate of the Bloom filter. For the popular items that attract the majority of data access, the decrease in false positive rate can reduce the false checking times in the ubiquitous nodes, and hence improve the system performance. “Cold” items will not find a match in the index lookup procedure. Under that circumstance, the index algorithm uses traditional lookup method to check each node one by one, until finding a match or confirming that the wanted query does not exist in the system. Since most queries for the items are popular items, the new mechanism can still improve system performance. Once the algorithm finds the wanted query in the ubiquitous nodes, it stops the lookup procedure immediately.

4.1 Algorithm Procedure

Like PBA, LBA has two steps: index building and item locating. The algorithm is described in reference [23].

Index Building. The system sets a load factor β, which is the ratio of the item number loaded by the Bloom filter. Each node orders the items by their access time and inserts the items one by one until it reaches the load threshold. For a system of totally N items, the Bloom filter arrays hold βN items.

Item Locating. When a query for a certain item arrives, the index node first calculates with the Bloom filter of each node and collects the results. One possible situation after the calculation is that there is at least one positive result, it means that the queried item exists on the node with a probability of $1 - f_{FP}$. Then the system first queries the actual node whose Bloom filter checking result is positive to verify whether the queried item exists in the node. If it does exist on one of the positive nodes, the lookup procedure stops immediately; otherwise it continues to check on the remaining negative nodes until it finds the queried item. The other possible situation after Bloom filter calculation is that there is no positive match. Under that circumstance, the system checks each unsearched node directly until it finds the item. The lookup procedure is shown in Fig. 2.

4.2 Performance Analysis

We first define the system environment and parameters. Let M be the total size of the entire Bloom filters array. N is the total number of items. Those objects are unique items and their distribution on the nodes is the uniform distribution, so each node has approximately N/s items. s is the node number. The load factor is β. Hr is the ratio of the access number the items indexed by the Bloom filters can absorb to the total number of item access. It can be proved [23] that when

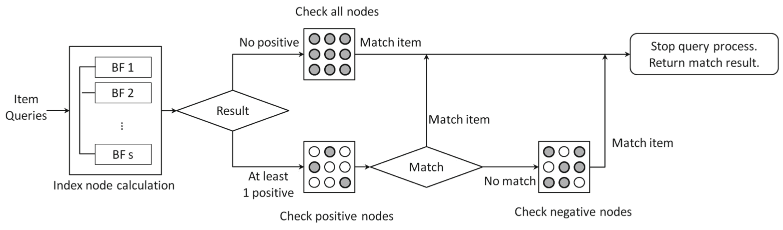

Fig. 2. Lookup procedure of LBA.

the hash function number reaches the optimal value in Eq. (2), the false checking times of LBA for each query is:

$$F_{LBA} = \frac{f_P(s-1) + (1-Hr)(1-f_P)^2(s-1)}{2} \tag{4}$$

Here $f_P = 0.6185^{\frac{M}{N\beta}}$.

Now we use the theoretical deduction to analyse the characteristics of LBA.

Literature [24, 25] recorded the access frequency for resource on the Internet. They pointed out that web access for objects follows Zipf's distribution or Zipf-like distribution (in the remainder of the paper, we call those two distributions Zipf's distribution collectively). In Zipf's distribution, the parameter α indicates the concentration of the object access, which ranges from 0 to 1. The larger α is, the more concentrated the Internet access is to the popular items, and the larger Hr is. On the contrary, the smaller α is, the less concentrated the Internet access is to the popular items, and the smaller Hr is. Now we use theoretical tools to examine the effect of M, α and β on system performance. First we look at the influence of α and β. See Fig. 3.

In Fig. 3 we set N = 1000000, s = 100, M = 2000000. We can see from the figure that when β increases, the false checking times first decreases, and then increases. The algorithm performance has a minimal value in reference to β. The false checking times decreases with the increase of α, which shows that the more concentrated the object accesses are, the higher the system performance. That is, the more Internet access are focused on "hot" items, the more efficient LBA is.

Now we use theoretical deduction to find the relationship between the LBA performance and the total size of Bloom filters array, M. The result is given in Fig. 4.

In Fig. 4 we set N = 1000000, s = 100, α = 0.95, β = 0.5. From the figure we can see that the false checking times decreases with the increase of M, which indicates that the algorithm performs better when the total size of Bloom filters array is larger.

In conclusion of the above two figures we can see that the proper choice of β can optimize LBA performance. The false checking times of LBA decreases with the increase of α and M.

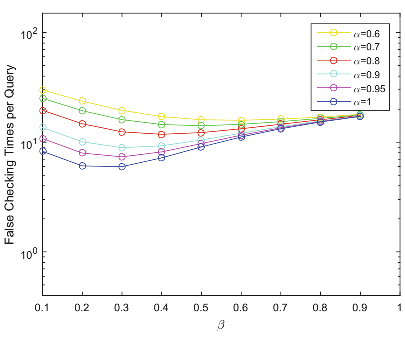

Fig. 3. The relationship between LBA performance and α, β.

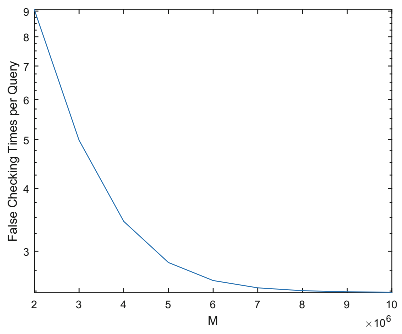

Fig. 4. The relationship between LBA performance and M.

5 Two-Tier Bloom Filter Array (2TBA)

The way in which 2TBA treats pervasive computing index is a little bit different from LBA: LBA uses a single Bloom filter to index "hot" items, while 2TBA uses two Bloom filters - one "hot" Bloom filter to index popular items, and one "cold" Bloom filter to index unpopular items. Now we present the 2TBA algorithm.

5.1 Algorithm Procedure

2TBA also has the procedure of index building and item locating, as given in reference [26].

Index Building. The system sets two global variables: a rank threshold β, which defines how much percentage of the ranked items is "hot" items (the other items are defined as "cold" items); a load factor γ which is the ratio of the length of "hot" Bloom filter to the total size of the "hot" Bloom filter and the "cold" Bloom filter. Each device orders its items by their access time. Then it inserts the "hot" items one by one into the "hot" Bloom filter and the "cold" ones into the "cold" Bloom filter. The two Bloom filters are transmitted to the index node.

Item Locating. The index node collects all the Bloom filters of the devices to form two Bloom filter arrays: a "hot" Bloom filter array and a "cold" Bloom filter array. When a query for a certain item arrives, the index node first checks each Bloom filter in the "hot" Bloom filter array. If there are some Bloom filters that report a match, the index node notifies the matched devices to check if the item really exists (recall that the Bloom filter may give positive response to items it does not contain because of the false positive rate). If it does exist on one of the positive nodes, the lookup procedure stops; otherwise the index node continues to calculate with the "cold" Bloom filter array. Like the previous step, it communicates with the ubiquitous device to check if the query does exist on the positive nodes indicated by its "cold" Bloom filter. The lookup procedure is shown in Fig. 5.

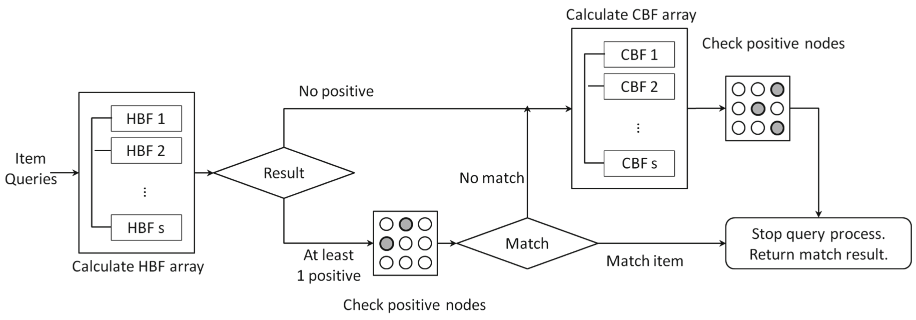

Fig. 5. Lookup procedure of 2TBA.

We can see that the lookup of 2TBA has three steps: “hot” Bloom filter lookup, “cold” Bloom filter lookup and node disk checking. Among the three, the first two steps happen in the memory of the index node, which take little time and cost little energy. The main factor that influence the system performance happens in the third step - the false checking in nodes, which requires communication between the index node and the ubiquitous nodes and takes up much time and energy. Now we analyse the 2TBA performance.

5.2 Performance Analysis

We first define the system environment and parameters. Let M be the total size of the entire Bloom filters array. N is the total number of items. Those objects are unique items and their distribution on the nodes is the uniform distribution, so each node has approximately N/s items. s is the node number. Hr is the ratio of the number of access the items indexed by the Bloom filters can absorb to the total number of item access. γ is the ratio of the size of “hot” Bloom filters to the total size of all Bloom filters, M. We can prove [26] that when the hash function number reaches the optimal value in Eq. (2), the false checking times of 2TBA for each query is:

$$F_{2TBA} = f_{P1}(s-1) + (1-Hr)(1-f_{P1})f_{P2}(s-1) \tag{5}$$

Here $f_{P1} = 0.6185^{\frac{M\gamma}{N\beta}}$, $f_{P2} = 0.6185^{\frac{M(1-\gamma)}{N(1-\beta)}}$.

Now we use theoretical deduction to analyse the characteristics of 2TBA.

We still assume that access for data objects follow Zipf’s distribution. We find the relationship between the system performance and β, γ. The result is shown in Fig. 6.

In Fig. 6 we set M = 8000000, N = 1000000, s = 100, α = 0.95. Each line represents the influence of β on the performance with a same γ. For each γ, there is an optimal β to minimize false checking times. We mark it a red point in the figure. All optimal β corresponding to each γ are connected to form a line, as we see in the figure, the optimal β line. We can find that the false checking times

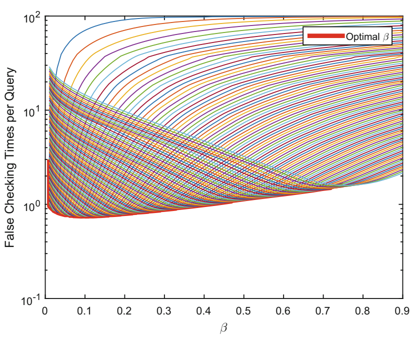

Fig. 6. The relationship between performance and β, γ. (Color figure online)

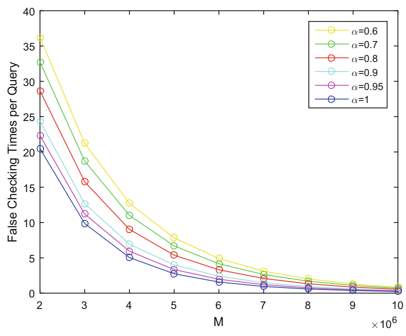

Fig. 7. The relationship between 2TBA performance and M, α.

of 2TBA has an optimal value regarding β and γ. By adjusting β, γ properly, we can optimize the system performance.

Now we want to find the impact of the total Bloom filter size M and the Zipf's distribution parameter α on the performance of 2TBA. See Fig. 7.

In the figure we set N = 1000000, s = 100, $\beta = 0.1$, $\gamma = 0.2$. We can see that the false checking times decreases with the increase of M and α. That indicates that the system performance increases with the size of Bloom filters. Also, the more concentrated the access for popular objects are, the higher the performance is.

From the analysis above we can see that we can choose proper β and γ to optimize 2TBA performance. Similar to LBA, the false checking times of 2TBA decreases with the increase of M and α.

6 Appropriate Application Scope of LBA and 2TBA

From the two sections above we can see that there are some similarities between LBA and 2TBA algorithm. The performance of the two algorithms increases with the concentration of object access and the increase of index space. Both algorithms are variants based on Bloom filters, and are used to deal with situations under which the index space and energy supply is limited.

In this section we want to give the answer to the question: what is the appropriate application scope of the two algorithms? Under a certain circumstance, which of the two has better performance? We use experiments to find the answer. In the experiment, we set N = 1000000, s = 100, M ranges from 2000000 10000000, α ranges 0.6, 0.7, 0.8, 0.9, 0.95, 1. For each group of LBA experiments with fixed M and α, β reaches the optimal value found by actual experiments, therefore we can get the optimal false checking times in that experimental group. For each group of 2TBA experiments with fixed M and α, β and γ reach the optimal value found by actual experiments, therefore we can also get the optimal false checking times in that experimental group. Figures 8 and 9 are the results of the false checking times of the two.

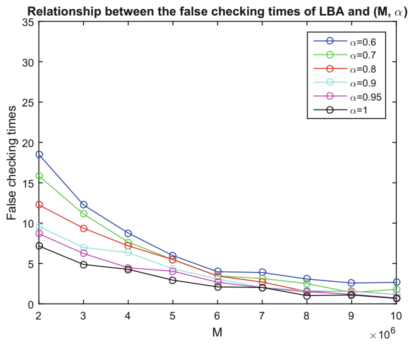

Fig. 8. LBA performance.

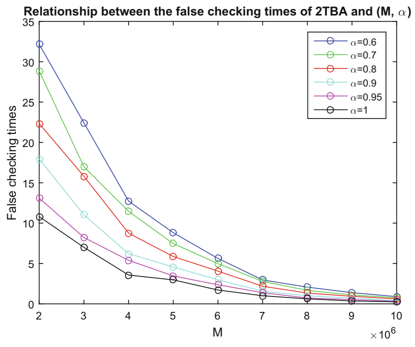

Fig. 9. 2TBA performance.

As expected, the performance of the two rises with the increase of index space and the concentration of data access. The tendency of the two algorithms is the same. In order to make a better comparison of the algorithms, we put them together in one figure, as in Fig. 10.

In the nine figures M equals 2000000, 3000000, 4000000, 5000000, 6000000, 7000000, 8000000, 9000000 and 10000000 respectively. It indicates that we use 2, 3, 4, 5, 6, 7, 8, 9, 10 bits to index one item on average. The horizontal axis is α. The vertical axis is the false checking times. The blue bar represents LBA, and the red bar represents 2TBA. We can see that with the growth of M and α, the false positive rate of 2TBA decreases faster than that of LBA.

To summarize the experiment results, we can see that LBA is more suitable when the index space is small and the access concentration is low. On the contrary, 2TBA is more suitable when the index space is relatively large and the object access is more concentrated. It is worth noting that those results are obtained under the circumstance that the index space for one item is quite low: when using 2 bits to index an object and $\alpha = 1$, LBA will encounter 7.17 false checkings before finding the right resident node, while 2TBA will execute 10.81 false checkings to find a query. When using 10 bits to index an item and $\alpha = 1$, LBA will do 0.67 false checkings before finding the right item, while 2TBA will check the wrong nodes 0.25 times.

After seeing the false positive rate of the two algorithms, we want to evaluate the energy consumption of the two. Since the index node can have abundant power supply, we only look into the ubiquitous nodes. First we give the settings of our ubiquitous embedded system. We make simulations on the active RFID system. The ubiquitous system uses a RFIDImpulse [27] mechanism to wake up the sleeping RFID nodes if there is need for the RFID reader to communicate with the RFID tag. Otherwise, the tag sleeps to save energy. The target platform of our simulation is the MicaZ from Crossbow, which has a CC2420 radio and an ATMEL128 microprocessor. The parameters are given in Table 1.

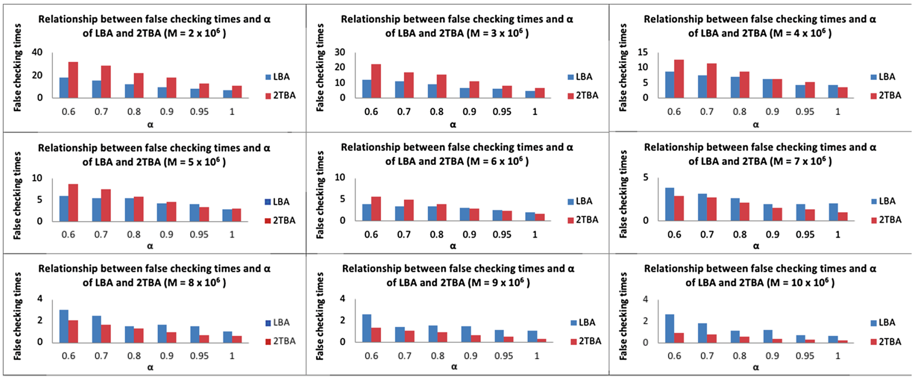

Fig. 10. Performance comparison of LBA and 2TBA. (Color figure online)

Table 1. Parameters of active RFID tag.

Parameter	Abbreviation	Value	Unit
Supply voltage	V	3	V
Transmit mode current	I_t	17.4	mA
Receive mode current	I_r	19.7	mA
Sleep current	I_s	1	μA
Byte transmission time	T_B	32	μSec
Battery energy	BE	200	mAh
Message length	L_M	20000	Byte

With these parameters, we can calculate the battery life of our system. We assume that the query frequency is 1 query per second. When dealing with one query, the energy that battery uses consists of three parts: receiving queries from the RFID reader, sending response to the reader and sleeping during two queries. Energy used to receive a message is

$$E_r = L_M \times T_B \times I_r \times V$$

Energy used to send a message is

$$E_t = L_M \times T_B \times I_t \times V$$

Energy used during the sleeping between two queries is

$$E_s = t_q \times I_s \times V$$

Then we can calculate the battery life of different scenarios in the previous experiment. The result is given in Fig. 11.

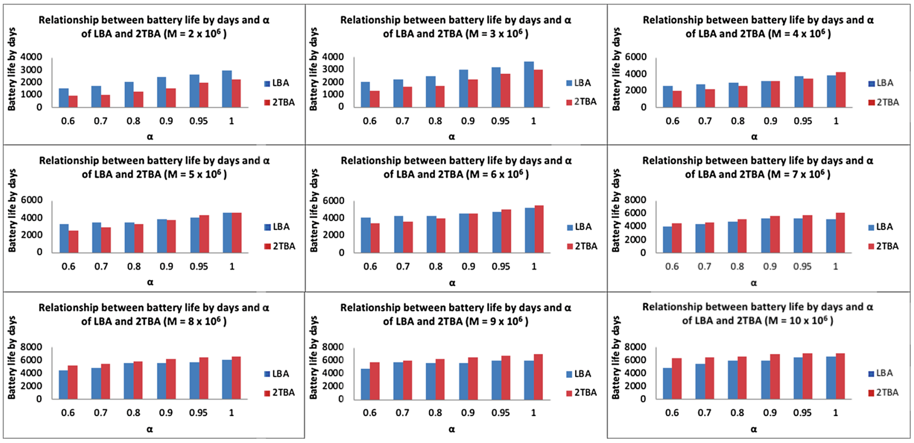

Fig. 11. Battery life.

In the nine figures M equals 2000000, 3000000, 4000000, 5000000, 6000000, 7000000, 8000000, 9000000 and 10000000 respectively. The horizontal axis is α. The vertical axis is the battery life measured by days. The blue bar represents LBA, and the red bar represents 2TBA. We can see that the battery life follows the same tendency with false checking times. It grows with the increase of α and index space. LBA is more suitable when the index space is small and the access concentration is low. 2TBA is more suitable when the index space is relatively large and the object access is more concentrated.

7 Conclusion

In this paper we have introduced two Bloom filter based algorithms for index service in pervasive embedded systems, the LBA approach and the 2TBA approach. Both algorithms use the Internet access pattern to optimize the traditional Bloom filter arrays. We have used theoretical approach to check the characteristics of the algorithms. Further we performed experiments to find the suitable application scope of the two services. Experiment results have shown that the performance of LBA and 2TBA increase with the rise of average index space and access concentration. When the index space is small and the data access is less concentrated, it is better to choose LBA as the index; when the index space is relatively larger and the data access is more concentrated, it is better to use 2TBA algorithm.

References

1. Mainwaring, A., Polastre, J., Szewczyk, R., Culler, D., Anderson, J.: Wireless sensor networks for habitat monitoring. In: Proceedings of the Eighth Annual International Conference on Mobile Computing and Networking, pp. 88–97. ACM (2002)
2. Yang, Z., Li, M., Liu, Y.: Sea depth measurement with restricted floating sensors. In: Proceedings of the 28th IEEE Real-Time Systems Symposium, pp. 469–478. IEEE Computer Society (2007)
3. Rachuri, K.K., Musolesi, M., Mascolo, C., Rentfrow, P.J., Longworth, C., Aucinas, A.: Emotionsense: a mobile phones based adaptive platform for experimental social psychology research. In: Proceedings of the 12th ACM International Conference on Ubiquitous Computing, pp. 281–290. ACM (2010)
4. Liu, Y., He, Y., Li, M., Wang, J., Liu, K., Li, X.: Does wireless sensor network scale? A measurement study on greenorbs. IEEE Trans. Parallel Distrib. Syst. **24**(10), 1983–1993 (2013)
5. Bloom, B.H.: Space/time trade-offs in hash coding with allowable errors. Commun. ACM **13**(7), 422–426 (1970)
6. Tarkoma, S., Rothenberg, C.E., Lagerspetz, E.: Theory and practice of bloom filters for distributed systems. IEEE Commun. Surv. Tutorials **14**(1), 131–155 (2012)
7. Mullin, J.K.: A second look at bloom filters. Commun. ACM **26**(8), 570–571 (1983)
8. Fan, L., Cao, P., Almeida, J., Broder, A.Z.: Summary cache: a scalable wide-area web cache sharing protocol. IEEE/ACM Trans. Netw. **8**(3), 281–293 (2000)
9. Bonomi, F., Mitzenmacher, M., Panigrahy, R., Singh, S., Varghese, G.: An improved construction for counting bloom filters. In: Azar, Y., Erlebach, T. (eds.) ESA 2006. LNCS, vol. 4168, pp. 684–695. Springer, Heidelberg (2006). https://doi.org/10.1007/11841036_61
10. Ficara, D., Giordano, S., Procissi, G., Vitucci, F.: Multilayer compressed counting bloom filters. In: Proceedings of the 27th Conference on Computer Communications (INFOCOM), pp. 311–315. IEEE (2008)
11. Song, H., Dharmapurikar, S., Turner, J., Lockwood, J.: Fast hash table lookup using extended bloom filter: an aid to network processing. In: Proceedings of the 2005 Conference on Applications, Technologies, Architectures and Protocols for Computer Communications (SIGCOMM), pp. 181–192. ACM (2005)
12. Bonomi, F., Mitzenmacher, M., Panigrah, R., Singh, S., Varghese, G.: Beyond bloom filters: from approximate membership checks to approximate state machines. In: Proceedings of the 2006 Conference on Applications, Technologies, Architectures and Protocols for Computer Communications (SIGCOMM), pp. 315–326. ACM (2006)
13. Sung, M., Xu, J., Li, J., Li, L.: Large-scale IP traceback in high-speed internet: practical techniques and information-theoretic foundation. IEEE/ACM Trans. Netw. **16**(6), 1253–1266 (2008)
14. Wang, Z., Luo, T.: Intelligent video content routing in a direct access network. In: Proceedings of the 3rd Symposium on Web Society, pp. 147–152. IEEE Computer Society (2011)
15. Wang, Z., Luo, T., Xu, Y., Cheng, F., Zhang, X., Wang, X.: A fast indexing algorithm optimization with user behavior pattern. In: Zu, Q., Hu, B., Elçi, A. (eds.) ICPCA/SWS 2012. LNCS, vol. 7719, pp. 592–605. Springer, Heidelberg (2013). https://doi.org/10.1007/978-3-642-37015-1_52

16. Jokela, P., Zahemszky, A., Rothenberg, C.E., Arianfar, S., Nikander, P.: LIPSIN: line speed publish/subscribe inter-networking. In: Proceedings of the 2009 Conference on Applications, Technologies, Architectures and Protocols for Computer Communications (SIGCOMM), pp. 195–206. ACM (2009)
17. Chen, T., Guo, D., He, Y., Chen, H., Liu, X., Luo, X.: A bloom filters based dissemination protocol in wireless sensor networks. Ad Hoc Netw. **11**(4), 1359–1371 (2013)
18. Takiguchi, T., Saruwatari, S., Morito, T., Ishida, S., Minami, M., Morikawa, H.: A novel wireless wake-up mechanism for energy-efficient ubiquitous networks. In: Proceedings of the 2009 IEEE International Conference on Communications Workshops, pp. 1–5. IEEE (2009)
19. Qwasmi, N., Liscano, R.: Bloom filter supporting distributed policy-based management in wireless sensor networks. In: Proceedings of the 4th International Conference on Ambient Systems, Networks and Technologies, pp. 248–255. Elsevier (2013)
20. Ghosh, M., Özer, E., Biles, S., Lee, H.-H.S.: Efficient system-on-chip energy management with a segmented bloom filter. In: Grass, W., Sick, B., Waldschmidt, K. (eds.) ARCS 2006. LNCS, vol. 3894, pp. 283–297. Springer, Heidelberg (2006). https://doi.org/10.1007/11682127_20
21. Jimeno, M.: Saving energy in network hosts with an application layer proxy: design and evaluation of new methods that utilize improved bloom filters. Ph.D. thesis, University of South Florida (2010)
22. Zhu, Y., Jiang, H., Wang, J., Xian, F.: HBA: distributed metadata management for large cluster-based storage systems. IEEE Trans. Parallel Distrib. Syst. **19**(6), 750–763 (2008)
23. Wang, Z., Luo, C., Luo, T., Chen, X., Hou, J.: A Bloom filter-based index for distributed storage systems. In: Omatu, S., Malluhi, Q.M., Gonzalez, S.R., Bocewicz, G., Bucciarelli, E., Giulioni, G., Iqba, F. (eds.) Distributed Computing and Artificial Intelligence, 12th International Conference. AISC, vol. 373, pp. 293–301. Springer, Cham (2015). https://doi.org/10.1007/978-3-319-19638-1_34
24. Chierichetti, F., Kumar, R., Raghavan, P.: Compressed web indexes. In: Proceedings of the 18th International Conference on World Wide Web, pp. 451–460. ACM (2009)
25. Breslau, L., Cao, P., Fan, L., Phillips, G., Shenker, S.: Web caching and Zipf-like distributions: evidence and implications. In: Proceedings of the Eighteenth Annual Joint Conference of the IEEE Computer and Communications Societies (INFOCOM), pp. 126–134. IEEE (1999)
26. Wang, Z., Luo, T., Yang, L.: An energy- and space-efficient object representation model in pervasive computing systems. IEEE Syst. J. **PP**(99), 1–11 (2016)
27. Jurdak, R., Ruzzelli, A.G., O'Hare, G.M.P.: Multi-hop RFID wake-up radio: design, evaluation and energy tradeoffs. In: Proceedings of the 17th International Conference on Computer Communications and Networks, pp. 641–648. IEEE (2008)