TELKOMNIKA, Vol. 11, No. 6, June 2013, pp. 3220 ~ 3227
e-ISSN: 2087-278X

A Dynamic Hashing Algorithm Suitable for Embedded System

Li Jianwei*, Chen Huijie
School of Computer Science and Technology, Taiyuan University of Science and Technology
Taiyuan, China
Corresponding author, e-mail: ghhong2004@163.com, chenhuijie666@163.com

Abstract

With the increasing of the data numbers, the linear hashing will be a lot of overflow blocks result from Data skew and the index size of extendible hash will surge so as to waste too much memory. This lead to the above two Typical Dynamic hashing algorithm don't suitable for embedded system that need certain real-time requirements and memory resources are very scarce. To solve this problem, this paper was proposed a dynamic hashing algorithm suitable for embedded system combining with the characteristic of extendible hashing and linear hashing. It is no overflow buckets and the index size is proportional to the adjustment number.

Keywords*: dynamic hashing, embedded system, overflow bucket, index size*

1. Introduction

With the rapid development of electronic technology, the performance of embedded hardware is continuous rising. This makes the embedded system can process more complex task [1]. Dynamic hashing has always been an indexing algorithm that widely used in general-purpose computer system.it can use hashing function to calculate the index address of the data should insert into. Besides, it can expand the index size to obtain more data and less overflow block. But it is necessary to get a right address using the same hashing function after expanding operation [2].

The typical dynamic hashing is extendible hashing and linear hashing. For the extendible hashing, the numbers of directory entries will double when the extendible hashing needs to expand index [3]. This will lead to the directory entries number surge. Then it result in waste too much memory. The linear hashing scheme is a directory-less scheme which allows a smooth growth of the hash table. Linear hashing split only one bucket each time. It can avoid the number of directory entries surge after several split like extendible hashing. However, the question is that the split bucket isn't the current split bucket and this will lead to many buckets have to maintain the overflow [4]. The result is decreasing of the query efficiency.

That is to say that both of them are not suitable for embedded environments [5-7]. The dynamic hashing mentioned in this paper is combining with the advantages of extendible hashing and linear hashing of course it can avoid the shortcomings of them and become a better dynamic hash.

2. Research Method

2.1. Basic Concepts and Notation Define

(1) Baseic concepts

An illustration of dynamic hashing is shown in Figure 1. In order to understanding well the following algorithm, we first introduce some basic concepts and define their symbols.

Index: the hashing table.it is the set of hashing table entry. If the index size is L, then it contains L hashing table entry. Hashing table entries are numbered from 0 to L-1.

Index entry: The basic unit of the directory. It should have two properties at least. One is the index entries address. Another one is the pointer to a bucket.

Received January 13, 2013; Revised April 6, 2013; Accepted April 22, 2013

Bucket: the set of data of record. The data or records in same bucket have some similar characteristics. For example, all the key of data in one bucket modulo the index size L equals the index entry. Data organization in the bucket can be varied. Such as BTree, Linked list etc.

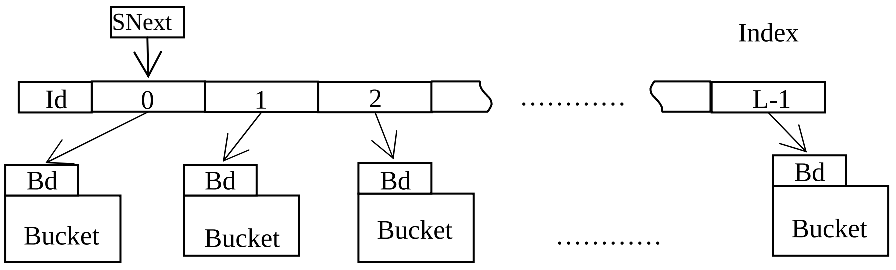

Figure 1. An Illustration of Dynamic hashing

Id: the deep of index or global deep, the properties of index. The value is $1 << (Ld - 1) < L \le 1 << Ld$.

Bd: the deep of bucket and it is the properties of bucket.it means that the left ($Key \div [1 << (WordLength - Bd)]$) or right ($Key\%(1 << Bd)$) Bd bit of data in the bucket are equal [8].

SNext: SNext is the index entry that will be split. The changes rules are as follows [9]:

$$SNext = \begin{cases} SNext++ & SNext \ne [(1 << Id) - 1] \\ 0 & SNext == [(1 << Id) - 1] \end{cases} \quad (1)$$

Value=F (key): the hashing function. It can put Key scattered into a certain range. Such as: $0 \le value \le (1 << n) - 1$.

(2) Notation define

L: the index size in current state. The biggest index entry address is L-1.

<<: Logical Shift Left

==: equals

!=: not equal

A++: A=A+1

%: Modulo

2.2. Addressing Algorithm
Step 1: Key=Get (data).get the key of data.
Step 2: Value=F (key).hashing the key.
Step 3: Get the index entry address E_id as the following function [10]:

$$E\_id = \begin{cases} value\%(1 << Id) & value\%(1 << Id) < L \\ value\%[1 << (Id - 1)] & value\%(1 << Id) \ge L \end{cases} \quad (2)$$

Step 4: then do the search operation in the bucket of E_id pointer to. The search operation algorithm is related to the data organization manner.

2.3. Insert Algorithm
Step 1: Key=Get (data).get the key of data.
Step 2: Value=F (key).hashing the key.
Step 3: Get the index entry address E_id as the function (2).
Step 4: Select the key of inserted data in the bucket of E_id pointer to. If it has been exist, return failure. Otherwise, insert the data into the bucket. At last, it needs to check whether or not the bucket have overflow block. If exist overflow block, need to do split operation.

2.4. Adjustment Algorithm

Get the local deep of split bucket Bd and the index entry address M pointer to it.do the following operation according to the relationship between the local deep of split bucket Bd and the index deep Id.

(1) If Bd==Id

Step 1: Id++; the global depth should plus 1

Step 2: get the directory entry number that the brother buckets of M.B_M=M+ (1<< (Id-1));

Step 3: expand the directory entry number to B_M。

Step 4: Split the nodes of bucket to M and B_M bucket as the manner of bucket split.

Step 5: The local depth of bucket M and brother bucket B_M should plus 1.

Step 6: The directory entry from N to B_M-1 should pointer to their brother bucket.

An illustration of adjustment operation when Bd==Id is shown in Figure 2(a).

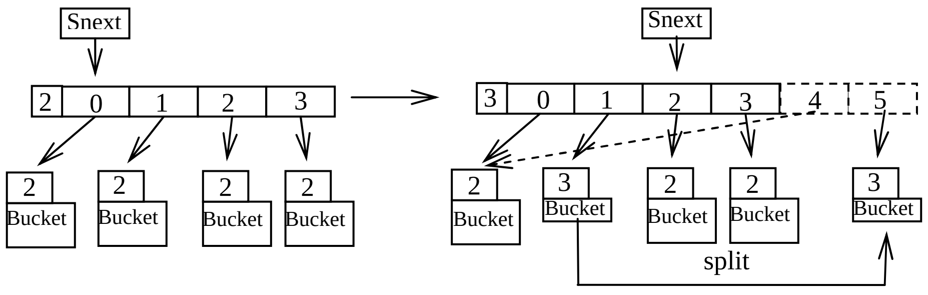

Figure 2(a). An Illustration of Adjustment Operation when Bd==Id

The index will expand 1<<(Id-1) entry in the worst situation. However, the index only expand 1 entry in the best situation.

(2) If Id==Bd+1

In this case, there may be exist two index entry pointing to the split bucket. However, we don't know whether or not the bigger index entry is already exists in the index. So it is necessary to do operation as the situation whether or not the bigger index entry is already exists in the index.

Step1: get the smaller index entry number pointer to bucket M: $\text{mini\_num=M}\&\left(2^{\text{Bd}}-1\right)$ .get the bigger index entry number: $\text{max\_num=mini\_num}|2^{\text{Bd}}$.

Step2: if max_num<L .it means the index entry max_num have already in the index. Split the nodes of bucket to max_num and mini_num bucket.The local deep of max_num and mini_num bucket should puls 1. The index doesn't need to expand the index.An illustration of adjustment operation when Bd+1==Id and max_num<L is shown in Figure 2(b).

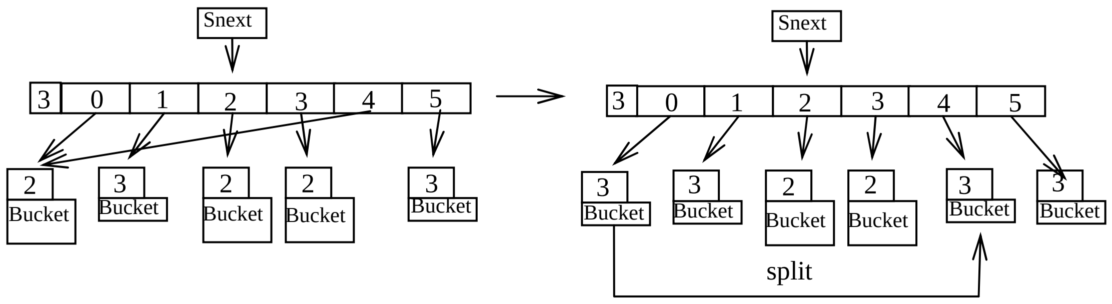

Figure 2(b). An Illustration of Adjustment Operation when Bd+1==Id and max_num<L

If $\max\_num \geq L$ it means the index entry max_num isn't in the index. Expand the directory entry number to $\max\_num$。Split the nodes of bucket to $\max\_num$ and $mini\_num$ bucket as the manner of bucket split. The local depth of bucket $\max\_num$ and $mini\_num$ should plus 1.at last, the directory entry from L to max_num-1 should pointer to their brother bucket. The index need expand 1<<(Id-1) entry in the worst situation.however,the index only expand 1 entry in the best situation.An illustration of adjustment operation when Bd+1==Id and max_num>=L is shown in Figure 2(c).

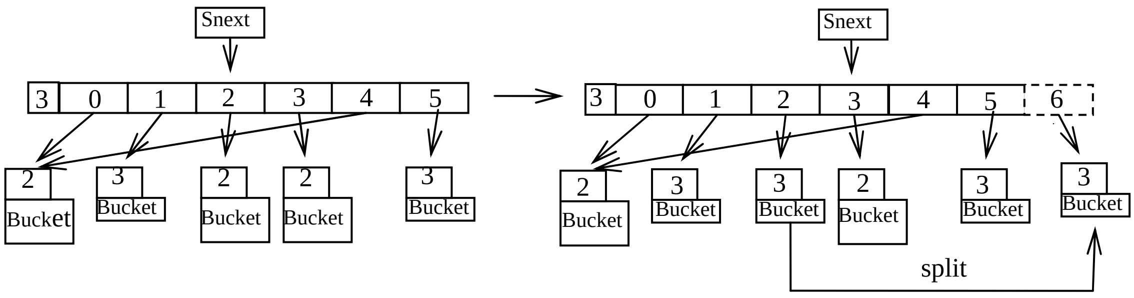

Figure 2(c). An illustration of adjustment operation when Bd+1==Id and max_num>=L

(3) If $Id > Bd+1$

Step 1: get the smallest index entry number pointer to M bucket: $mini\_M = M \& \left[(1 << Bd) - 1 \right]$ Get the brother index entry number closest to the smallest index entry: $B\_mini\_M = mini\_M | (1 << Bd)$

Step 2: Split the nodes of bucket to $mini\_M$ and B_mini_M bucket as the manner of bucket split. The local depth of bucket $mini\_M$ and brother bucket B_mini_M should plus 1.the brother index entry of $mini\_M$ and B_mini_M should ponter to the responded bucket. The index doesn't need to expand the index.An illustration of adjustment operation when Bd+1<Id is shown in Figure 2(d).

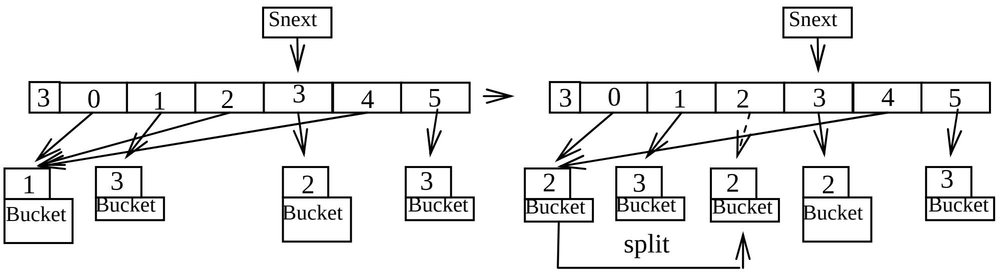

Figure 2(d). An Illustration of Adjustment Operation when Bd+1<Id

3. Experimental Results and Analysis

In this chapter, this paper will compare the overflow bucket number between linear hashing and the dynamic hash proposed in this paper. Then we will test the cost time of insert operation, Unit to adjustment time, addressing time and storage utilization. Besides, we will test the relation of the index size and split number. And analyze the causes of the growth trend. The average time (T) use the number of clock frequency (N) to characterize, computer frequency is represent by F. the conversion method with the actual time is as follows:

$$T(\text{ms}) = \frac{N(\text{number})}{F(\text{Hz})} \quad (3)$$

3.1. The Comparison of Overflow Bucket Number

No matter using what data organization method to manage the data in bucket. the sum of data number will have an influence on the search efficiency. The character of real time in embedded system require this operation can finish in a deadline time.so we define a bucket size to respond the deadline value. As soon as the sum data in the bucket reach to this value, the overflow happened. The compartion of overflow bucket number is show in Figure 3.

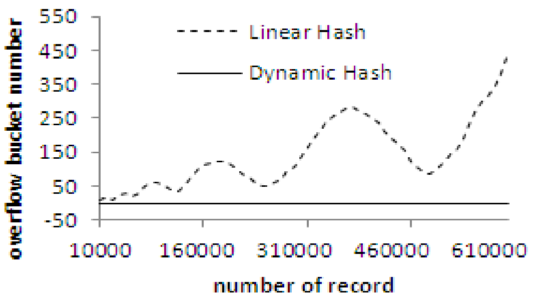

Figure 3. The Compartion of Overflow Bucket Number

From the refence [11], we can draw the conclution that the bucket will be split isn't the overflow bucket insert a record, and it is decide by the next pointer as cyclic manner. However, with the increasing of data number, the next pointer finish a cyclic need more time. That is to say, more and more bucket need to split have overflow block. The trend of overflow bucket number of linear hashing is show figure 3. From the dynamic hashing algorithm proposed in this paper, it is clear that as soon as a insert operation produce a overflow block, the split operation will be executed immediately. So there is no overflow block exist in the dynamic hashing proposed in this paper.

3.2. The Time of Addressing Cost

As the increasing of data numbers,the trend of addressing cost time is shown in Figure 4(a).

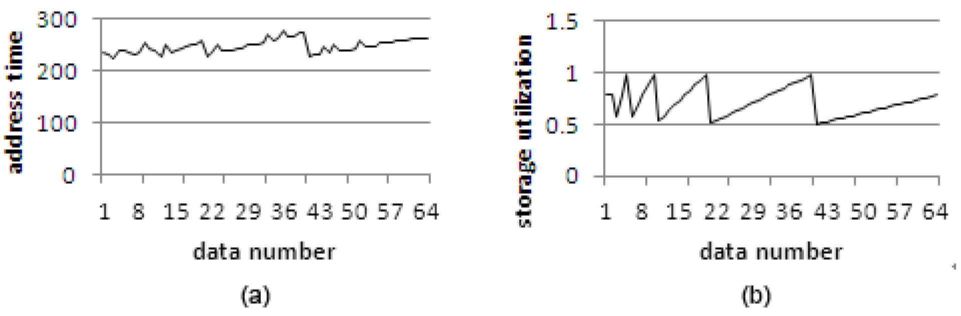

Figure 4(a). The Average Address Time and, (b) The Storage Utilization According to the Data Number

From the Figure 4(a), with the growing of data numbers, the average cost time of addressing tends to parallel and exsit some periodic fluctuations。The reasons is mainly depend on the storage utilization of bucket。The storage utilization trend is shown in Figure 4(b).and the storage utilization is calculated by the following manner:

$$\text{storage utilization}=\frac{\text{the sum of data}}{\text{the number of bucket}\times\text{bucket capacity}} \quad (3)$$

From the Figure 4(b), when the total amount of data is 400000, the storage utilization rate of bucket is nearly 98%. However, when the total amount of data is 410000. The utilization rate of bucket is about 51%. Then the former average depth of the bucket is deeper than the later bucket. Therefore,the average addressing time is decrease when the numbers of data increase from 400000 to 410000.with the amount of data continue to increasing, the average bucket utilization rate will be growing; the average tree depth of the bucket will be deeper, so this well lead the average addressing time to periodic fluctuations.

3.4. The Time of Adjustment Operation Cost

The adjustment operation is related to index table expansion, bucket split and index entry pointer to respond bucket. Bucket split is executed in every adjustment operation. However, the other two operation may be don't run in adjustment operation.besides, the time of adjustment cost is decide by the sum data number and the data organization manner. the index table expansion don't double the index size. So, it can reduce the index size. At last, in the worst situation, there is up to half of index size entry must pointer their conrespond bucket.

The total adjustment time trend is shown in Figure 5(a). The growth trend of total adjustment numbers is shown in Figure 5(b). The unit to adjustment is shown in Figure 5(c). The average adjustment time is shown in Figure 5(d). The average adjusting time is caluculated as follows:

$$\text{average adjustment time}=\frac{\text{the total time of adjustment cost}}{\text{the sum of data}} \quad (4)$$

Unit to adjustment time refers to the average time once adjustment. Both the total of adjustment time and the total of adjustments number are statistic from the experiment. The unit to adjustment time is calculated as follows:

$$\text{average unit to adjustment time}=\frac{\text{the total time of adjustment cost}}{\text{the total of split number}} \quad (5)$$

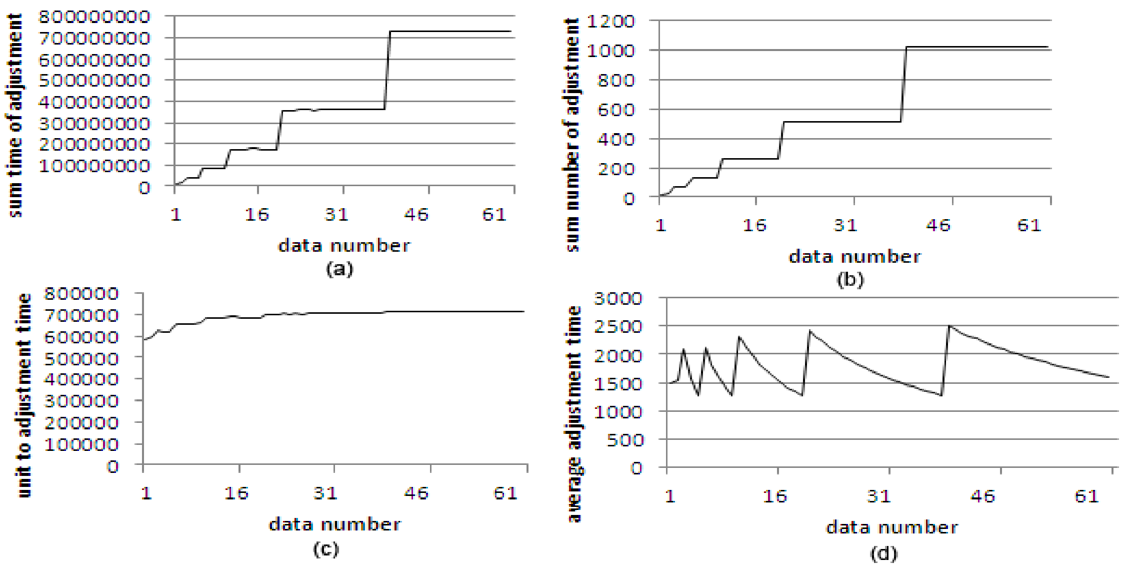

Figure 5(a). Sum Adjustment Time, (b) Sum Adjustment Number, (c) Unit to Adjustment Time and, (d) Average Adjustment Time According to the Data Number

3.4. The Relation of Index Size and Split Numbers

From the chapter 2, we can draw the conclution that the index expantion may don’t run in adjustment operation. In order to investigate the speed of index expansion when runing adjustment operation. We test the index size and adjustment number after inserting many data that ranging from 10000 to 640000.

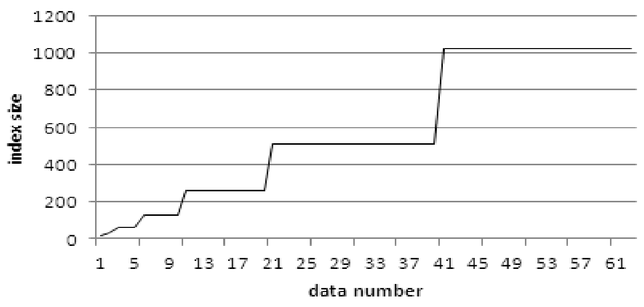

Figure 6. The Index Table Size According to the Data Number

Figure 6 depicts the index size according to the number of inserted data. Figure 7 depicts the index size according to the number of adjustment operation run. We can see that the index size is proportional to the adjustment number.

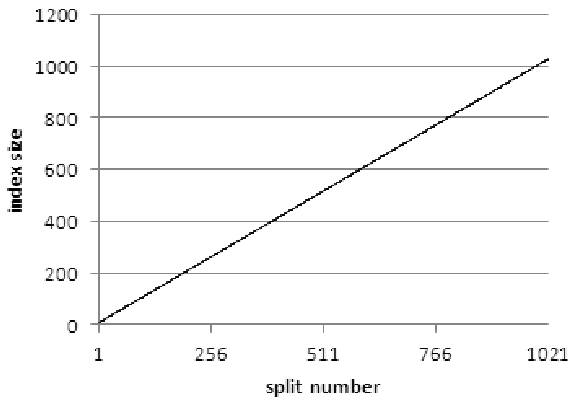

Figure 7. The Index Table Size According to the Split Number

4. Conclusion

In this paper, we are proposed a new efficient dynamic hashing algorithm for embedded system. The bucket will immediate split when bucket overflow occurs. So there is no overflow in this new efficient dynamic hashing algorithm compared with linear hashing. Besides, there are four adjustment operation strategies according to the different relationship of bucket deep and index table deep. Not all of the adjustment operation need to expand the index table. Bedsides, in the adjustment operation that need to do adjustment operation, The index will expand 1<<(Id-1) entry in the worst situation. However, the index only expand 1 entry in the best situation.

From the various experimental results, we showed that the proposed dynamic hashing doesn’t have overflow blocks and the index size is proportional to the adjustment number. That is mean that it is outperforms traditional dynamic hashing algorithm on the embedded system that require real-time and memory resources are very scarce. Finally, we plan to implement and analyze the practical performance of the proposed hash inde on the practical real-time embedded system in the future.

Acknowledgement

This work is under the support of the “Shanxi Province nature Foundation (No.2012011027-3)”; authors hereby thank them for the financial supports.

References

[1] Mullally Adrian, McKelvey Nigel, Curran Kevin. Performance comparison of enterprise applications on mobile operating systems. *TELKOMNIKA Indonesian Journal of Electrical Engineering.* 2011; 9(3): 503-514.
[2] Rammohanrao K, Lioyd JK. Dynamic Hashing Schemes. *Computer Journal.* 475-485.
[3] Ronald Fagin, Jurg Nievergelt. Extendible Hashing-A Fast Access Method for dynamic Files. *ACM Transactions on Database Systems*. 1979; 4(3): 315-344.
[4] Witold Litwin. *Linear hashing: a new tool for file and talbe addressing*. Proceedings of the 6rd Intl. Conference on Very Large Data Bases. Canada. 1980: 212-223.
[5] Wang Xibo, Li Nan. *Embedded System Memory Management Mechanism Based on uC.OS*–II. Proceedings of the 2010 International Conference on Communications and Mobile Computing. 2010; 258-262.
[6] Woochul Kang, Sang H Son. Power and time aware buffer cache management for real-time embedded databases. *Journal of Systems Architecture: the EUROMICRO Journal*. 2012; 58(2-3): 233-246.
[7] Yu Hu, Zhang Weihuai. Research on real-time and dynamic urban traffic: Information service system. *TELKOMNIKA Indonesian Journal of Electrical Engineering*. 2012; 10(4): 806-811.
[8] Askok Rathi, Huizhu Lu. *Performance comparison of extendible hashing and linear hashing techniques.* ACM Press 1515 Broadway, 17th Floor New York, NY.DOI:10.1145/122045. 122048.19-26.
[9] Larson P. Performance Analysis of a Single-file Version of Linear Hashing. *Computer Journal.* 319-326.
[10] Hul-Woong YANG, et al. An Efficient Dynamic Hash Index Structrue for NAND Flash Memory. *IEICE Transactions on Fundamentals of Electronics, Communications and Computer Sciences*. 2009; E92-A 7: 1716-1719.
[11] Xiang Li, et al. A New Dynamic Hash Index for Flash-Based Storage. Proceedings of the 9rd Intl. Conference on Web-Age Information Management. 2008; (20-22): 93-98.