Turkish Journal of Electrical Engineering & Computer Sciences

http://journals.tubitak.gov.tr/elektrik/

Research Article

Turk J Elec Eng & Comp Sci
(2018) 26: 1417 – 1429
© TÜBİTAK
doi:10.3906/elk-1708-336

High-speed data deduplication using parallelized cuckoo hashing

Jane Rubel Angelina JEYARAJ[*], **Sundarakantham KAMBARAJ**, **Velmurugan DHARMARAJAN**
Department of Computer Science and Engineering, Thiagarajar College of Engineering, Madurai, Tamil Nadu, India

Received: 29.08.2017 • **Accepted/Published Online:** 03.04.2018 • **Final Version:** 30.05.2018

Abstract: Data deduplication is a capacity optimization technology used in backup systems for identifying and storing the nonredundant data blocks. The CPU intensive tasks involved in a hash-based deduplication system remain as challenges in improving the performance of the system. In this paper, we propose a parallel variant of the standard cuckoo hashing that enables the hashing technique to be performed in parallel. The CPU intensive tasks of fingerprint insertion and lookup operations are performed in parallel and distributed among the nodes of the deduplication cluster. Furthermore, the uniform handling of the blocks by the cluster nodes involved in the process of duplicate identification provides good load balance. Experimental evaluations using real-world backup and Linux kernel data sets reveal that the proposed deduplication system achieves up to 100% higher backup speed, up to 28% reduced lookup latency, and up to 24% reduced backup time than the other deduplication systems.

Key words: Deduplication, parallelized cuckoo, backup

1. Introduction

In this day and age, data deduplication has become one of the most essential components of existing storage systems. It will gain even more importance, as data exploitation occurs, while the growth of data is estimated to reach 44 trillion gigabytes by the year 2020. Furthermore, the International Data Corporation (IDC) says nearly 75% of these data will be copies of original data. It gives rise to the need for a system that could effectively reduce and manage this huge volume of data efficiently. One such space saving technique is a data deduplication system, which could reduce the data footprint even up to and beyond 80%.

Data deduplication is a dedicated data compression technique with which duplicate copies or redundant data can be eliminated [1]. Traditional compression techniques [2] provide data reduction just by comparing the redundancies within a file. Moreover, the compression techniques need a considerable amount of time to regenerate the data. Data deduplication, on the other hand, eliminates redundancies not only by comparing the data segments inside a single file, but also across multiple files. Furthermore, it regenerates the original data in less time.

Duplicates can be eliminated either at whole file level [3] or even effectively at block level [4,5]. In a block level deduplication process, unique data blocks called chunks are identified, analyzed, and then stored. During analysis, upon arrival of a new chunk, it is compared against all the existing chunks. The occurrence of a redundant chunk is replaced with a small reference that points to the existing stored chunk. Through this process, the amount of data that must be stored or transferred can be greatly reduced.

[*]Correspondence: janerubel@tce.edu

A parallel approach to deduplication can improve the throughput of the deduplication backup system [6]. Parallel indexing techniques [7,8] can accelerate the fingerprint lookup in a deduplication system. The tasks involved in the deduplication process can be assigned to different nodes, which can further enhance parallelism.

The actual speed of traditional backup is pretty high by simply sending the files to a backup server as it is, but the consumption of resources like storage and network bandwidth is high. The objective of this proposed work is to build an efficient data deduplication system that minimizes resources such as storage and network bandwidth and that could parallelize the CPU intensive tasks such as fingerprint insertion, lookup, and duplicate identification to speed up the backup process.

The contributions of this paper are as follows:

- A new parallelized variant of the standard cuckoo hashing is proposed that parallelizes the key insertion and lookup operations across indexing structures
- A novel deduplication technique utilizing the proposed parallel version of cuckoo hashing is presented that includes the following features:
    - The CPU intensive tasks of fingerprint insertion and fingerprint lookup operations are distributed across the nodes of the deduplication cluster
    - The parallel execution of the tasks involved in deduplication yields substantial improvement in the performance of the deduplication system
    - The allocation of chunks to different nodes in the deduplication cluster provides good load balance improving resource utilization

The performance of this proposed novel technique is evaluated using four different chunk sizes and two different data sets.

The remainder of this work is organized as follows. First, Section 2 describes the state of the art of existing schemes, introducing the general concept of deduplication. Section 3 presents the proposed system architecture, describes the standard cuckoo hashing, and introduces the proposed parallelized cuckoo hashing. Section 4 presents our parallelized deduplication technique with a modified version of cuckoo hashing. Finally, the performance evaluation is presented in Section 5, before the conclusions are given in Section 6.

2. Research background

Existing data deduplication techniques are broadly classified based on the factors such as granularity of duplicates, method utilized, and where deduplication is performed.

Based on granularity, deduplication may be performed at whole file level or at block level. In whole file deduplication [3], duplicates are eliminated if and only if the files taken for deduplication exactly contain the same content. This situation is very rare, and it may not offer much reduction in storage, i.e. deduplication rate. Hence it is preferred to use block level deduplication, in which the deferral blocks alone are stored, if there exist two or more files with slight modification.

In a block level deduplication system, the identification of unique blocks can be effected by different techniques such as hash-based comparison [9] or simply byte-by-byte comparison [10]. A byte-by-byte comparison of chunks becomes practically impossible for very large deduplication systems as there may be millions of chunks. A hash-based comparison, on the other hand, can summarize the content of the chunks by creating a fingerprint for each chunk using any of the cryptographic hash-based algorithms [11]. Thus the process of identifying the duplicate blocks need not access the disk every time, resulting in considerable improvement in overall performance of the deduplication system.

A hash table-based deduplication (HT-dedupe) involves a hash-based data structure to support fingerprint lookups [12,13]. Various hashing schemes [14] are used in the literature to perform an effective lookup operation in the hash-based structure. Cuckoo hashing [15], a variant of open addressing, performs better than conventional hashing algorithms like chained hashing, linear probing, and double hashing [14] in providing an assurance of constant worst-case lookup time. Hence cuckoo hashing (cuckoo-dedupe) [16] and its variants are used in deduplication systems. Cuckoo hashing, however, suffers from hash collisions and endless loops, which results in throughput degradation. To mitigate collisions, locking mechanisms are employed on the hash table [17,18]. The problems of endless loops in cuckoo hashing have been handled by variants of cuckoo hashing [19,20]. Multithreading approaches are used to parallelize cuckoo hashing [18,20]. The proposed parallelized cuckoo builds upon the cuckoo hashing techniques employing multithreading. Cuckoo hashing techniques using locking mechanisms lock the hash table and support multiple reads/single writes; thereby the writes to the table are sequential. Moreover, locking approaches introduce deadlocks. The proposed parallelized cuckoo hashing is a lock-free approach, employing multithreading to support parallelism. Furthermore, the hash tables are placed in two different nodes, allowing simultaneous writes to the hash tables, and thus improve the throughput.

A centralized deduplication system relies on a single node to perform the deduplication [21], even if it gets the input from numerous nodes. In contrast, a distributed deduplication includes multiple nodes to perform deduplication, thus parallelizing the deduplication process. Some distributed deduplication systems have separate indexing structures [22]. Some deduplication systems have a centralized index structure. A distributed deduplication helps improve the scalability and throughput of the resulting deduplication system.

Cloud-based storage services such as DropBox and Google Drive accomplish storage capacity optimization using file-level deduplication across the files stored by various users [23]. Such commercial cloud storage services can achieve significant reduction in storage costs and bandwidth requirement when chunk-level deduplication is performed. Moreover, the deduplication throughput can be increased when our proposed parallelized deduplication technique is employed.

In order to exploit the combined benefits of the distributed deduplication approach and the cuckoo hashing method used in a hash-based data structure, we propose a parallelized approach to cuckoo hashing. We distribute the computationally intensive tasks of fingerprint insertion, lookup, and duplicate identification involved in a hash-based deduplication system among the nodes of a cluster. A variant of cuckoo hashing is proposed that uses two hash indexing structures present in different nodes of a cluster. Both the indexing structures are involved simultaneously in fingerprint insertion, lookup, and duplicate identification, thus parallelizing the deduplication process.

3. System design/concepts

In this section, we depict the architecture design of our proposed system. Then we illustrate the design of our proposed variant of cuckoo hashing, which implants a parallelized approach to the hashing technique.

3.1. Architecture overview of the proposed system

Figure 1 illustrates the architecture of our proposed system. The proposed system consists of two key functional components, namely client and deduplication cluster.

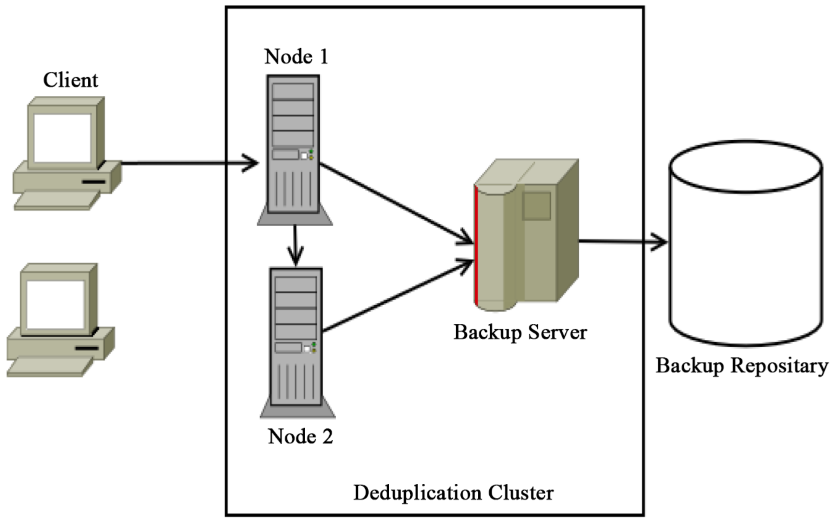


Figure 1. Architecture of the proposed system.

- *Client* The Client machines provide an interface to the backup or restore operations. It is responsible for sending the input meant for backup to the deduplication cluster for processing. It also consists of subfunctional components, namely a fixed size chunker and a hash engine. The fixed size chunker is responsible for breaking the backup files into chunks of fixed size. We consider fixed size chunking owing to its speed and computational simplicity. Extensive experiments are carried out using varied chunk sizes such as 10 KB, 100 KB, 1 MB, and 10 MB. The hash engine is responsible for generating summary information for the created chunks. Thus the fingerprints generated serve to uniquely identify every chunk. Generally, a variety of cryptographic hash algorithms [11] such as MD5, SHA-1, and SHA-256 are used to generate the fingerprints in a deduplication system. Although the SHA variants are more secure than MD5, we use MD5, as it generates the smallest fingerprint and it is faster than the SHA variants. The generated hash value called fingerprint along with the data block is transmitted over TCP to the deduplication cluster.

- *Deduplication cluster* The deduplication cluster involves machines that are coupled using a high speed dedicated network interface card (NIC), so that the processing does not lag on network delay. The deduplication cluster is composed of distributed indexing structures located on the different nodes of a cluster. It is responsible for parallelized fingerprint insertion and lookup operations performed on the distributed indexing structures. The duplicate chunks are identified using the hash collision on fingerprint lookup. The fingerprint of unique chunks is inserted into the indexing structure and the unique chunks are forwarded to the backup server and get stored there. In the event of detecting a duplicate, the chunk is discarded, replaced by a small reference to the already existing chunk in the backup server. In order to achieve better performance in duplicate identification, we deploy a modified version of cuckoo hashing that is designed to work in parallel. Thus through parallel processing and elimination of dependency between the nodes on the cluster, duplicate identification is performed at a higher speed than the existing systems.

3.2. Cuckoo hashing

Cuckoo hashing is an efficient hashing technique proposed by Pagh [15] in which hash collision is addressed by adhering to the behavior of the cuckoo bird while allocating room for new keys kicking out the existing keys. The main advantage of using cuckoo hashing is that it provides constant worst case lookup time, O (1), unlike various hashing techniques like linear probing and quadratic probing, where the worst case lookup time is around O (n), where n is the number of entries in the hash table.

Cuckoo hashing uses two hash tables, T_1, T_2, and two hash functions termed as h_1, h_2. Functioning of cuckoo hashing can be described with an illustration as given in Figure 2. When a new key K_{new} arrives, it always gets inserted in the first table T_1 at location $l_1 = h_1 (K_{new})$. If T_1 is already occupied by another key K_{old}, that is, $h_1 (K_{old}) = h_1 (K_{new})$ but $K_{new} \neq K_{old}$, then we "kick out" K_{old} and move it to its alternate position on T_2, $l_2 = h_2 (K_{old})$. If T_2 is already occupied by another key, then we proceed with this "kick out" procedure until an unoccupied location is found in either T_1 or T_2, where the key K_{new} is stored.

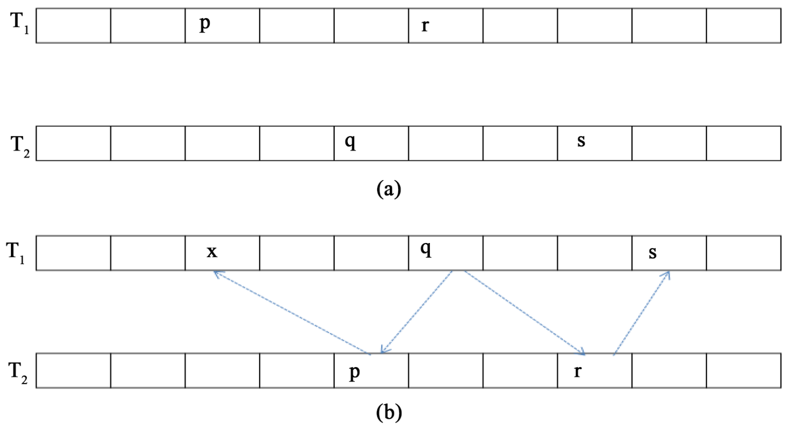


Figure 2. Cuckoo hashing: (a) Present state of hash tables T_1 and T_2; (b) Key insert and kick-out operations.

Rehashing is a most common but very expensive operation in almost every hash-based data structure. Cuckoo hashing can operate at its best even when the tables T_1 and T_2 are filled over 90%. It is the unique advantage of using cuckoo hashing over other hashing techniques proposed in the literature [14], where the load, i.e. no of entries on the table, going high degrades the performance of the hashing technique.

3.3. Proposed parallelized cuckoo

We propose a variant of a cuckoo hashing with two hash tables T_1 and T_2 of equal size present in different nodes of a cluster, and have two different hash functions h_1 and h_2. A key can be present either in table T_1 or in table T_2. Hash table T_1 is designed like a linear table like data structure having m slots to accommodate the m keys. Hash table T_2 is designed as a chained-hash table having m slots. Each slot points to the head of a linked list. The different keys that are hashed to the same slot are placed in the same linked list. The presence of a key, x, can then be found as illustrated in Algorithm 1.

Algorithm 1 procedure lookup(x)

```
1: At Node 1: if (key x, is in $T_1$ [$h_1$(x)]) then return true
2: At Node 2: else if (key x, is in $T_2$ [$h_2$(x)]) then return true
3: else return false
```

The lookup procedure initially uses the hash function h_1 and searches for the key x, in table T_1 at slot T_1 [h_1(x)]. If the key is available in this slot, then the lookup is successful. If the key is not available in this slot, then the search is continued in table T_2 using the hash function, h_2 at slot T_2 $[h_2(x)]$. If the key is not present in both the hash tables, it results in an unsuccessful search. It can be noted that the lookup procedure needs only two table accesses.

The insertion of a key, x, into a hash table follows the procedure as given in Algorithm 2.

Algorithm 2 procedure insert(x)

```
Processing: Steps 1 and 2 follow sequential processing at Node 1
Steps 4 and 5 follow sequential processing at Node 2
Parallel processing is done in Node 1 and Node 2 at the same time, with different keys, x
1: At Node 1: if ($T_1$ [$h_1(x)$] = x) then return
2: At Node 1: else if (the slot, $T_1$ [$h_1(x)$] is NULL) then $T_1$ [$h_1(x)$] $\leftarrow$ x
3: else Node 1 hands over the key, x to Node 2 and fetches the next key, x
4: At Node 2: if (the slot, $T_2$ [$h_2(x)$] contains x) then return
5: At Node 2: else $T_2$ [$h_2(x)$] $\leftarrow$ x
```

The insert procedure initially checks for the existence of the key, x, in hash table T_1 in the slot, T_1 $[h_1(x)]$ at Node 1. If the key is available, no insert action is performed. If the slot is empty, then the key is inserted in that slot. In contrast, if that slot is occupied by some other key rather than x, then Node 1 hands over the key to Node 2 and the insert procedure tries to insert the key x, in the hash table T_2 in the slot, $T_2[h_2(x)]$ at Node 2. At the same time, Node 1 fetches the next key and tries to insert it in the hash table T1 using the steps 1 and 2 of the insert procedure. The insertion of the key, x, is continued in parallel at Node 2. If the key is already present in the slot, T_2 $[h_2(x)]$, no insert action is performed. Otherwise, key x is inserted in the slot, T_2 $[h_2(x)]$.

4. Deduplication using the proposed parallelized cuckoo hashing

Hash table-based deduplication involves using a hash table and corresponding hash functions. The essential steps involved in such a deduplication system are chunking, fingerprint generation, hash table maintenance, fingerprint insertion and lookup in the hash table, duplicate fingerprint identification, and elimination. This section discusses in detail the proposed deduplication system using the proposed parallelized cuckoo hashing illustrated in Section 3.

Even though variable size chunking yields a better deduplication ratio [5], it is CPU demanding in nature. Hence we employ fixed size chunking, which operates at high speed as the incoming data stream is chunked at a fixed size. Fingerprints are generated for each generated chunk using cryptographic hash algorithm MD5.

The next step is inserting the generated fingerprints into the indexing structure and finding the duplicates. We employ our proposed parallelized cuckoo hashing algorithms depicted as Algorithm 1 and Algorithm 2, to insert the fingerprints into the hash tables and also for the fingerprint lookups. Our proposed parallelized deduplication approach using the proposed parallelized cuckoo hashing operates as shown in Algorithm 3.

In deduplication cluster node 1, index position for the fingerprint is computed using hash function h_1, and a lookup is made on the slot T_1 $[h_1$ (fingerprint)] in table T_1. If the position is vacant, the fingerprint is inserted in that free position considering it as the chunk's unique fingerprint. The chunk is stored in the backup server. If the position returned by hash function h_1 is already occupied by some other fingerprint, then the current fingerprint is handed over to Node 2. Meanwhile, Node 1 continues fetching the next fingerprint and

Algorithm 3 Parallelized deduplication using the proposed parallelized cuckoo

```
Client:
1: Break data into chunks of fixed size
2: Generate chunk fingerprint using MD5 / SHA-1 / SHA-256
3: Invoke deduplication cluster Node 1
Deduplication cluster node 1:
4: Check the availability of fingerprint in the hash table, $T_1$
   a. If free slot found at $T_1$ [$h_1$ (fingerprint)], store the fingerprint in $T_1$ and move the chunk to storage
   b. If found occupied, compare the fingerprint with existing fingerprint
      i If matches, duplicate is identified. So pointer to the chunk is added and the actual chunk is not stored.
      ii If not matches, invoke deduplication cluster node 2 and hand over the fingerprint. Fetch the next fingerprint from client and continue with step 4.
Deduplication cluster node 2:
5: Check the availability of fingerprint in the chained hash table, $T_2$
   a. If free slot found at $T_2$ [$h_2$ (fingerprint)], store the fingerprint in $T_2$ and move the chunk to storage
   b. If found occupied, compare the fingerprint with existing fingerprint.
      i. If matches, duplicate is identified. So pointer to the chunk is added and the actual chunk is not stored
      ii. If not matches, the fingerprint is entered as a new bucket entry at position $T_2$ [$h_2$ (fingerprint)]
```

performs the same operation.

Node 2, which receives the fingerprint handed over from Node 1, makes a comparison of this fingerprint with the already stored fingerprint held in the slot, T_1 [h_1(fingerprint)]. If the comparison matches, then the chunk is identified as a duplicate and a reference pointer is set to point to the chunk stored in the backup server. In the case the fingerprint does not match, then Node 2 generates a new index with hash function h_2 and tries to insert it in chained-hash table T_2. If the slot T_2 [h_2 (fingerprint)] in table T_2 is vacant, the fingerprint is simply added to T_2 considering it as unique chunk. If the slot T_2 [h_2 (fingerprint)] in T_2 is not vacant, the search for the fingerprint goes through the entire list attached with the slot T_2 [h_2 (fingerprint)]. If a match is found, then the chunk is identified as a duplicate and a reference pointer alone is set in the backup server. In contrast, if a match is not found, then the fingerprint is entered as a bucket entry at position T_2 [h_2 (fingerprint)]. At this point, Node 2 waits for another handover from Node 1.

Thus Node 1 and Node 2 operate in parallel in fingerprint insertion, lookup, and duplicate identification in our proposed deduplication system using our proposed parallelized cuckoo hashing technique.

The computational complexity of the proposed method would be O(1) in the best case, i.e. when the fingerprints of the chunks are stored in the first hash table T_1 and when fingerprint lookups happen in T_1. The complexity of the proposed method would be O(n) in the worst case, i.e. when all the fingerprints of the chunks are inserted into the same bucket of hash table T_2 and when fingerprint lookups happen in T_2, scanning all the entries. We have performed an experiment showing the fingerprint distribution among the two hash tables as shown in Figure 3. The number of fingerprints handled by the second hash table is always less than the first hash table. Moreover, since fingerprint insertion in both the hash tables occurs simultaneously, the execution time is reduced.

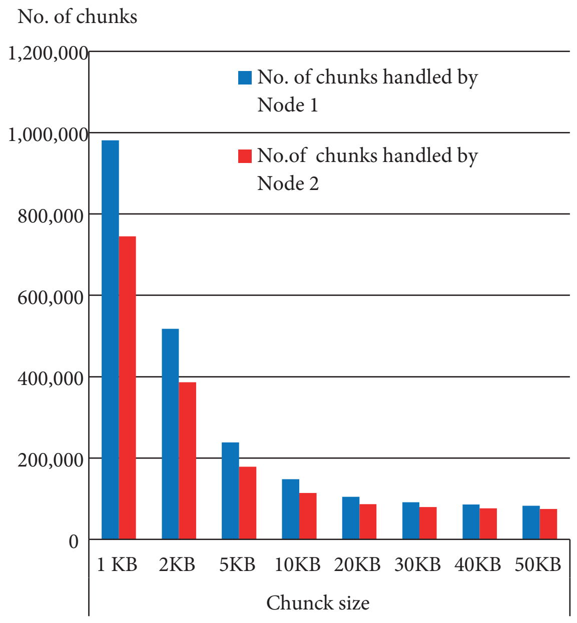


Figure 3. Load distributions of chunks among the cluster nodes.

4.1. Load distribution

Node 1 keeps on running for almost every incoming chunk; meanwhile Node 2 is initiated only when a collision occurs in Node 1. However, the computationally intensive task like fingerprint comparison is being carried out only at Node 2, which happens for almost every duplicate chunk that keeps Node 2 active. Hence, Node 1 coupled together with Node 2 provides performance enhancement, eliminating the bottleneck in the process of deduplication.

Figure 3 depicts the load distribution among the two nodes of a cluster. Based on the results shown, it is clearly seen that the number of chunks handled on Node 2 is always less than that of Node 1. The number of chunks handled by Node 2 is almost equivalent to the number of chunks handled by Node 1. Also in none of the test conditions is Node 2 left idle or unutilized. From the observations, we can conclude that Node 1 and Node 2 have equal contribution in the system, working together in parallel to yield better performance.

5. Experiment

5.1. Experimental setup

We have implemented our proposed research prototype backup deduplication system using our proposed parallelized cuckoo hashing technique in a cluster system. The client machines providing an interface to avail the input to the deduplication system are powered by Intel Dual Core CPU @2.2 GHz, 2 GB DDR2 RAM modules, running Windows XP operating system. The cluster nodes of the deduplication server are powered by quad-core Intel i5 @2.60 GHz with total 4 GB DDR3 RAM modules, running Ubuntu 64 bit Desktop Operating system version 14.2. The backup server used for chunk repository is powered by Intel i7 CPU with 16 GB RAM modules.

JAVA JDK1.80 is used as the programming language, in which the default libraries are utilized for fingerprint generation. As we use two different machines to achieve parallelism, we use a controller to pass on the generated fingerprint to deduplication cluster machine 1 using RMI of Java, where the given fingerprint is compared with the entries in the hash table. If the fingerprint received is not a duplicate, then deduplication cluster machine 2 gets invoked using RMI and the duplicate finger print identification is performed in the chained hash table. The multithreading and RMI approaches in Java are used to achieve parallelism.

For our experiment and analysis two datasets were considered. We describe the characteristics of the data sets.

- *Linux kernel.* Extracted Linux kernel, downloaded from the Linux Kernel Archives (https://www.kernel.org/) being the first data set is made up of three consecutive releases of Linux-3.18.21, Linux-4.1.9, and Linux-4.22. This data set consists of C files, header files, Python, Perl and shell scripts, text documents, and others. Size of this data set is 1.57 GB.
- *User set.* The second data set is real world backup data consisting of documents, lab programs, executable, power point presentations, mails, etc., stored in one of our institution's servers. Its size is 11.97 GB.

5.2. Experimental evaluation

In order to perform comprehensive analysis, we compare our proposed deduplication system using parallelized cuckoo hashing with the already existing systems, hash table based deduplication (HT-dedupe), and cuckoo hashing based deduplication (cuckoo-dedupe). The factors such as running time, lookup latency, and backup speed are considered.

5.2.1. Backup speed analysis

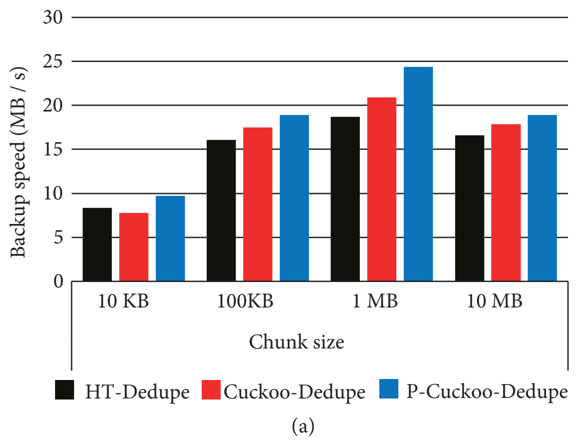
Backup speed is the speed at which the backup operation is being carried out. When the speed of backup is improved it reduces the overall deduplication backup time. Generally the speed of the backup is computed in megabytes per second (MBps). Higher the MBps, the higher is the speed of backup.

We examine the backup speed under varying chunk sizes of 10 KB, 100 KB, 1 MB, and 10 MB and it is depicted in Figure 4. The deduplication system using parallelized cuckoo hashing increases backup speed over HT-dedupe by 33% and 85% on the Linux kernel and user data sets, respectively, and over cuckoo-dedupe by 25% and 100% on the Linux and user data sets, respectively. From the obtained results it is clearly noticeable that the backup speed reaches its maximum when the chunk size is 1 MB under both the data sets.

5.2.2. Execution time analysis

We compute execution time to be the total time taken for the entire backup process. Figure 5 shows the execution time taken by the deduplication systems HT-dedupe, cuckoo-dedupe and our proposed P-cuckoo-dedupe. The experiments are carried out using the above-mentioned data sets and with varying chunk sizes.

From the results, it can be found that the deduplication system using parallelized cuckoo hashing on the Linux kernel and user data sets reduces the execution time by 23% when compared to HT-dedupe and reduces the execution time by 20% and 24% on the Linux kernel and user data sets respectively when compared with



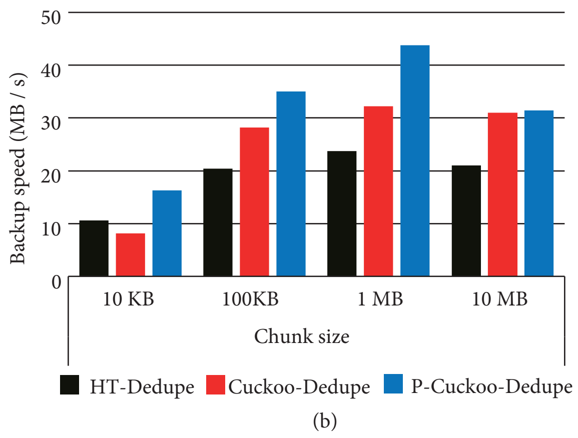
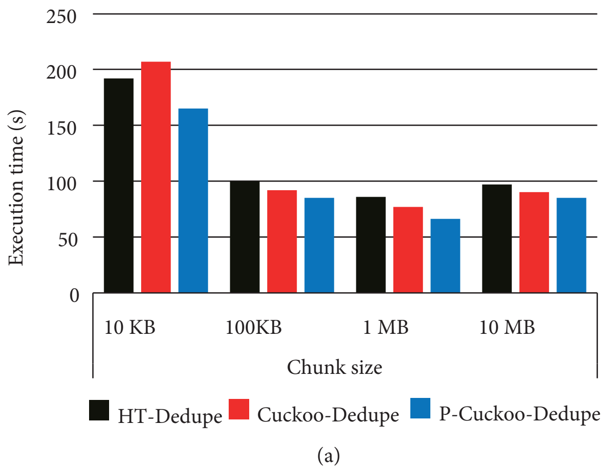


Figure 4. Backup speed under varying chunk sizes (a) with Linux kernel (b) with user set.



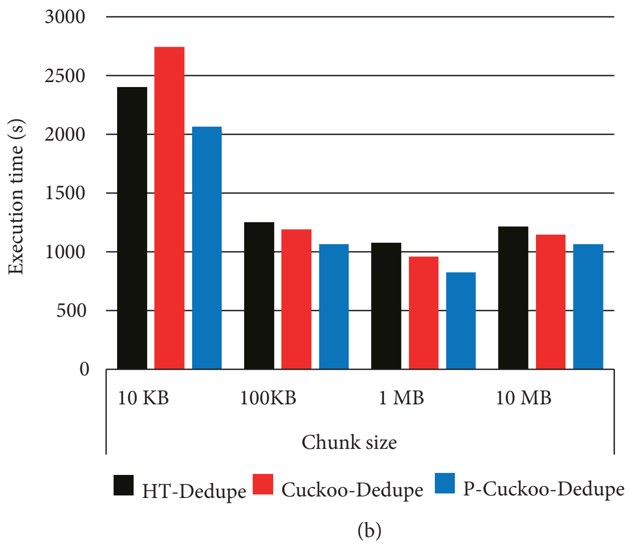


Figure 5. Execution time for backup deduplication (a) with Linux kernel (b) with user set.

cuckoo-dedupe. It can also be seen that P-cuckoo-dedupe takes less execution time in all the experiments carried out. This is because the parallelized cuckoo hashing works in such a way that it distributes the chunks among the cluster nodes and carries out the fingerprint insertion in parallel, whereas the regular cuckoo hashing does every process in a sequential way.

5.2.3. Lookup latency analysis

Lookup latency is the time taken to search whether the particular chunk's fingerprint is already present in the hash table or not. We examine the average lookup latency under varying chunk sizes as shown in Figure 6.

From the results obtained, we could see that the P-cuckoo-dedupe shows only a marginal improvement of average lookup latency when compared to that of HT-dedupe and cuckoo-dedupe in almost all the cases. However, under the chunk size of 1 MB, a substantial improvement of 28% in lookup latency is recorded. This proves that the proposed parallelized cuckoo hashing approach is able to perform as well as the standard cuckoo hashing approach.

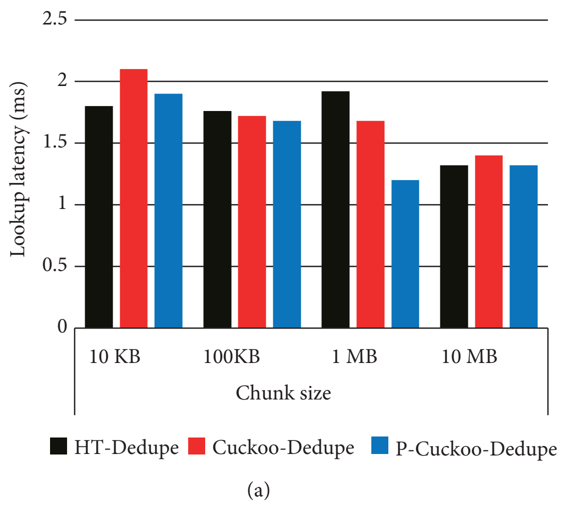


Figure 6. Average lookup latency with different chunk sizes (a) with Linux kernel (b) with user set.

5.2.4. Comparison between the proposed parallelized cuckoo and variants of cuckoo hashing

We examine the insertion throughputs of the proposed parallelized cuckoo and compare them with the insertion throughputs of the variants of cuckoo hashing such as libcuckoo [18] and MemC3 [17] using the Bag-of-words dataset (http://archive.ics.uci.edu/ml/datasets/Bag+of+Words) as shown in Figure 7. We use the open-source implementation (https://github.com/efficient/libcuckoo) for libcuckoo and implement the MemC3 Cuckoo Insert Procedure [18].

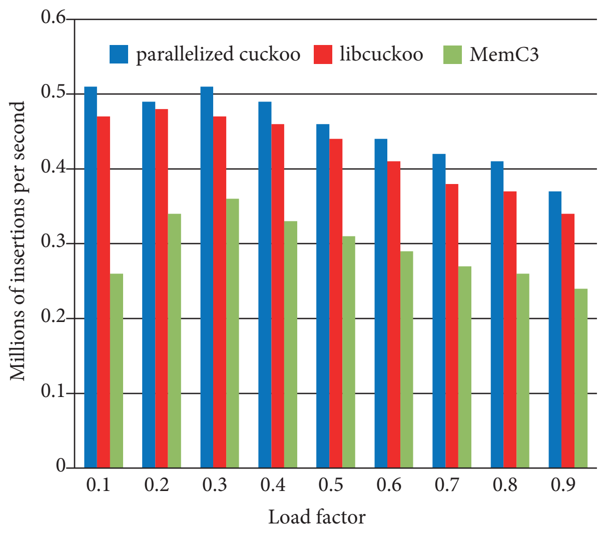


Figure 7. Insertion throughputs with Bag-of-words data set.

The proposed parallelized cuckoo obtains an average of 7.3% improvement over libcuckoo and 54% over MemC3. The improved performance of the proposed parallelized cuckoo algorithm over the other variants of cuckoo presented in the literature clearly justifies that the proposed deduplication system using the parallelized cuckoo hashing will outperform deduplication systems developed using the other variants of cuckoo hashing.

6. Conclusion

In this work, we proposed a parallel variant of the standard cuckoo hashing technique. We also presented our proposed parallel deduplication system, which employs our proposed parallel cuckoo hashing technique. The CPU intensive tasks of fingerprint insertion and lookup operations involved in a hash table-based deduplication system are distributed among the deduplication cluster nodes.

The experimental evaluation shows that our proposed research prototype backup deduplication system can improve the speed of backup. The total time taken for backup and the delay involved in the fingerprint lookup are reduced. Furthermore, a good load balance is maintained across the deduplication cluster nodes.

References

[1] Xia W, Jiang H, Feng D, Douglis F, Shilane P, Hua Y, Fu M, Zhang Y, Zhou Y. A comprehensive study of the past, present, and future of data deduplication. P IEEE 2016; 9: 1681-1710.

[2] Sayood K. Introduction to Data Compression. 4th ed. Waltham, MA, USA: Elsevier, 2012.

[3] Meyer DT, Bolosky WJ. A study of practical deduplication. ACM T Storage 2012; 4: 1-13.

[4] Ha JY, Lee YS, Kim JS. Deduplication with block-level content-aware chunking for solid state drives (SSDs). In: IEEE 10th International Conference on High Performance Computing and Communications & IEEE International Conference on Embedded and Ubiquitous Computing; 13–15 November 2013; Zhangjiajie, China. New York, NY, USA: IEEE. pp. 1982-1989.

[5] Li J, Li YK, Chen X, Lee PPC, Lou W. A hybrid cloud approach for secure authorized deduplication. IEEE T Parall Distr 2015; 5: 1206-1216.

[6] Wei J, Jiang H, Zhou K, Feng D. MAD2: A scalable high-throughput exact deduplication approach for network backup services. In: IEEE 26th Symposium on Mass Storage Systems and Technologies; 3–7 May 2010; NV, USA. New York, NY, USA: IEEE. pp. 1-14.

[7] Xia W, Jiang H, Feng D, Hua Y. Similarity and locality based indexing for high performance data deduplication. IEEE T Comput 2015; 4: 1162-1176.

[8] Lillibridge M, Eshghi K, Bhagwat D, Deolalikar V, Trezise G, Camble P. Sparse indexing: large scale, inline deduplication using sampling and locality. In: USENIX 7th Conference on File and Storage Technologies; 24–27 February 2009; San Francisco, CA, USA. Berkeley, CA, USA: USENIX. pp. 111-123.

[9] Won Y, Lim K, Min J. MUCH: multithreaded content-based file chunking. IEEE T Comput 2015; 5: 1375-1388.

[10] Arcangeli A, Eidus I, Wright C. Increasing memory density by using KSM. In: Linux Symposium; 13–17 July 2009; Montreal, Canada. pp. 19-28.

[11] Stallings W. Cryptography and Network Security: Principles and Practice. 6th ed. Harlow, United Kingdom: Pearson Education Limited, 2013.

[12] Min J, Yoon D, Won Y. Efficient deduplication techniques for modern backup operation. IEEE T Comput 2011; 6: 824-840.

[13] Bhagwat D, Eshghi K, Long DDE, Lillibridge M. Extreme binning: scalable, parallel deduplication for chunk-based file backup. In: IEEE International Symposium on Modeling, Analysis & Simulation of Computer and Telecommunication Systems; 21–23 September 2009; London, UK. New York, NY, USA: IEEE. pp. 1-9.

[14] Cormen TH, Leiserson CE, Rivest RL, Stein C. Introduction to Algorithms. 3rd ed. Cambridge, MA, USA: Massachusetts Institute of Technology, 2009.

[15] Pagh R, Rodler FF. Cuckoo hashing. J Algorithm 2004; 2: 122-144.

[16] Debnath BK, Sengupta S, Li J. ChunkStash: speeding up inline storage deduplication using flash memory. In: USENIX annual technical conference; 23–25 June 2010; Boston, MA, USA. Berkeley, CA, USA: USENIX. pp. 16-16.

[17] Fan B, Andersen DG, Kaminsky M. MemC3: compact and concurrent MemCache with dumber caching and smarter hashing. In: USENIX annual technical conference; 2-5 April 2013; Lombard, IL, USA. Berkeley, CA, USA: USENIX. pp. 385-398.

[18] Li X, Andersen DG, Kaminsky M, Freedman MJ. Algorithmic improvements for fast concurrent cuckoo hashing. In: Ninth European Conference on Computer Systems; 14–16 April 2014; Amsterdam, Netherlands. New York, NY, USA: ACM. pp. 1-14.

[19] Kirsch A, Mitzenmacher M, Udi Wieder. More robust hashing: cuckoo hashing with a stash. Siam J Comput 2009; 4: 1543-1561.

[20] Dan AA, Sharf A, Abbasinejad F, Sengupta S, Mitzenmacher M, Owens JD, Amenta N. Real-time parallel hashing on the GPU. ACM T Graphic 2009; 5: 1-9.

[21] Rhea S, Cox R, Pesterev A. Fast, inexpensive content-addressed storage in foundation. In: USENIX Annual Technical Conference; 22–27 June 2008; Boston, MA, USA. Berkeley, CA, USA: USENIX. pp. 143-156.

[22] Paulo J, Pereira J. A survey and classification of storage deduplication systems. ACM Comput Surv 2014; 1: 1-30.

[23] Bellare M, Keelveedhi S, Ristenpart T. Message-locked encryption and secure deduplication. In: Springer Annual International Conference on the Theory and Applications of Cryptographic Techniques; 26–30 May 2013; Athens, Greece. Berlin, Germany: Springer. pp. 296-312.