CodePlugin: Plugging Deduplication into Erasure Coding for Cloud Storage

Mengbai Xiao[1], Mohammed A. Hassan[2], Weijun Xiao[3], Qi Wei[1], and Songqing Chen[1]

[1]*George Mason University*
[2]*NetApp Inc.*
[3]*Virginia Commonwealth University*

Abstract

Cloud storage systems play a key role in many cloud services. To tolerate multiple simultaneous disk failures and reduce the storage overhead, today cloud storage systems often employ erasure coding schemes. To simplify implementations, existing systems, such as Microsoft Azure and EMC Atmos, only support file appending operations. However, this feature leads to a non-trivial and increasing portion of redundant data on cloud storage systems.

To reduce the data redundancy due to file updates by users so as to reduce the corresponding encoding and storage cost, in this work, we investigate how to efficiently integrate the inline deduplication capability into the general context of the Reed-Solomon (RS) code. For this purpose, we present our initial design of CodePlugin. Basically, CodePlugin introduces some pre-processing steps before the normal encoding. In these pre-processing steps, the data duplications are identified and properly shuffled so that the redundant blocks do not have to be encoded. CodePlugin is applicable to any existing coding scheme and our preliminary experimental results show that CodePlugin can effectively improve the encoding throughput (by $\sim 20\%$) and reduce the storage cost (by $\sim 17.4\%$).

1 Introduction

Cloud storage is fundamental to many other cloud services. To provide availability and reliability against simultaneous disk failures, cloud storage systems often employ some replica protocols. To simplify the replica protocol, these systems are mostly designed to allow append-only operations, such as GFS [6] and WAS [4]. Therefore, in these systems, data are sealed in fixed-sized blocks and only read and delete operations are allowed.

To achieve reliability while saving storage cost, the erasure coding is widely used in various storage systems. For example, the Reed-Solomon (RS) code [14] has been adopted in several large cloud storage systems, such as Microsoft Azure [4] and EMC Atmos [1]. In these systems, incoming data is considered as streams composed of fixed-size containers, which are then encoded. The coded blocks are commonly distributed into various storage nodes for future container reconstruction once some storage nodes are temporarily inaccessible. Existing coding schemes in cloud mainly focus on rapidly regenerating coded blocks (e.g., for reducing I/Os) against disk failures [9, 8]. There is little work found on improving the encoding performance.

Today redundant data widely exist in cloud storage systems. For example, the VM images are geared with similar operating systems and commonly used software. Furthermore, due to the append-only feature, the update operations always lead to a great amount of redundancy in the cloud storage systems. As the end-users keep similar text across their editable files in the versioning system [7, 3], and every update to the kernel or application makes a VM image slightly different to the previous version. All these update operations are likely to generate new containers at the lower level of the cloud storage system. Then the system has to re-encode these redundant data repeatedly for keeping availability. It has been found that there is about 30% redundant data in the primary storage workload [5]. Such redundancy leads to several consequences: (1) extra storage has to be used to accommodate such redundant data, (2) extra coding (I/O) cost has to be paid since the redundant data have to be encoded as well. If we can effectively reduce such redundancy in a cloud storage system due to various update operations, the storage efficiency and the encoding (I/O) throughput of the erasure coding can be improved.

This motivates us to consider integrating deduplication characteristic into erasure coding to more efficiently support file updates. A straightforward solution is to deduplicate the input data first, and then encode the remaining data. However, if we look into the deduplication pro-

cess (described in section 2.1), small data pieces will be generated and stored across the storage system. Thus the subsequent reads will be random access with a high probability. So the first challenge is to mitigate such I/O performance degradation for primary storage systems. Furthermore, deduplication usually demands support at the filesystem level. This constrains the applicability of the deduplication technique on the primary storage.

To this end, we propose CodePlugin, which utilizes the standard deduplication technique without demanding extra filesystem support. CodePlugin applies some preprocessing steps to identify redundant data before encoding and strives to keep the original file untouched even if it is chunked and deduplicated. Thus, the read performance of the original file is not degraded. By effectively reducing the amount of redundant data, CodePlugin further reduces the amount of data for encoding. This in turn improves the encoding performance. Our preliminary experiments based on some real-world VM workloads show the effectiveness of CodePlugin by improving the throughput for $\sim 20\%$ and reducing the storage size for $\sim 17\%$.

Note that CodePlugin can be used with any coding scheme, although our following discussion focuses on RS code.

The rest of the paper is organized as follows. We present some background information in section 2, and the design of CodePlugin in section 3. Experimental results are presented in section 4. We make concluding remarks in section 5.

2 Background

2.1 Deduplication

Deduplication is commonly used in backup storage systems. In general, it consists of the following steps: 1) chunking the raw data based on a variable chunk size or a fixed chunk size, 2) calculating the fingerprints, e.g., using MD5 or SHA-1, of each resulting data piece, 3) determining if one data piece is redundant by searching the index table, where all unique fingerprints are kept, 4) adding the fingerprints of new unique data pieces to the index table and replacing the redundant data pieces with the references to the existing ones.

The variable sized chunking usually has higher efficiency in detecting redundant data since it can limit the updates in a small range. Rabin Fingerprinting [13] is often used, such as in LBFS [11]. It has been shown that the primary storage workload has about 10% more duplication if using the Rabin Fingerprinting rather than the fixed sized chunking [10]. However, the fixed sized chunking has much higher throughput and *rsync* [16] uses the fixed size chunking.

The performance bottleneck of a deduplication system is often at the I/O throughput [17, 15]. Bloom filter and cache are often used to help with this problem.

3 CodePlugin Design

To integrate deduplication with erasure coding, CodePlugin introduces some pre-processing steps before the encoding operation. These processing steps are: (1) the de-duplicating step that tries to identify the redundant blocks; (2) the pseudo-shuffling step, which is to virtually re-arrange the positions of data blocks so that encoding is only needed on a subset of blocks; and (3) the optional sub-files exchanging step, which can further reduce the number of blocks to be encoded.

Before the detailed illustration of our design, for ease of presentation, we assume that in *each* encoding process there are N raw data files, each of which is of size S. Without loss of generality, we assume that the (k,m) RS code is used.

3.1 Pre-Processing

3.1.1 Basic Deduplication

For deduplication, we choose fixed chunking. To make it clear, we define the data separated by erasure coding as *sub-files*, which are usually stored in separate devices for availability. For $(4,2)$ code, the file is divided into 4 sub-files. The data pieces chunked by the deduplication technique are referred to as *blocks*. If the block size is s bytes, we can expect that one file is about to be chunked into $\frac{S}{s}$ blocks, and $\frac{S}{k \cdot s}$ blocks for each sub-file. Every block is identified by its fingerprint, i.e. SHA-1 or MD5. A block is marked either as *unique* or *redundant* according to the following processing.

After the file is chunked, the blocks are identified via 3-tuple (fid, sid, cid) *address*, which is composed of the corresponding *file id (fid)*, *sub-file id (sid)* and *column id (cid)*. Figure 1 shows an example.

Since it is difficult and unnecessary to perform deduplication globally in CodePlugin, we choose to use a cache to keep the fingerprints. In the cache, a hash table is responsible for mapping fingerprints of unique blocks to their addresses, and the LRU algorithm is used to evict items when the cache is full.

Figure 1 illustrates the deduplication process. When a file is read into memory, it is firstly chunked and all fingerprints are generated. After comparing these fingerprints to the ones in the cache, CodePlugin can tell which blocks are redundant, as well as the corresponding address of their *source blocks*. Such information and references are kept in the *map-file* table for each file.

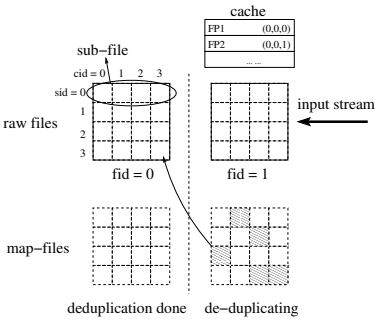

Figure 1: *Deduplication in CodePlugin.* When processing the file with $fid = 1$, some redundant blocks (shadow blocks) and the referring source blocks are found. The calculated fingerprints, e.g. *FP1*, are stored in the cache with their addresses.

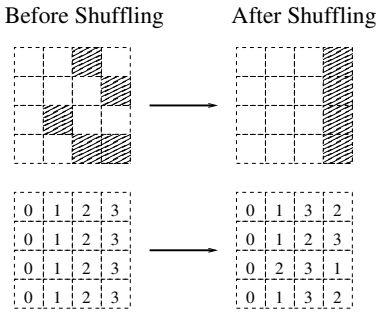

Figure 2: *Pseudo-Shuffling in CodePlugin.* After shuffling, the unique blocks (without shadow) aggregate on the left side. The redundant blocks move to the right side. The matrix shows the new positions of each block.

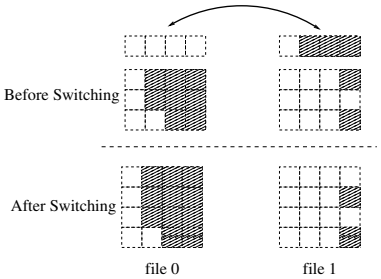

Figure 3: *Sub-files Exchanging in CodePlugin.* Before exchanging, we have to encode 4 stripes for each file – each stripe contains unique blocks. After exchanging, we only need to encode 2 stripes for file 0 and 4 stripes for file 1, a total of 6 stripes.

3.1.2 Pseudo-shuffling

Intuitively, to take advantage of deduplication, we should encode the unique blocks together, and leave the redundant blocks untouched. However, the challenge is that how we can keep the sub-files as the original RS code does. This is because if we encode the blocks from the same sub-file, we will fail to reconstruct this sub-file once it is erased.

Thus, in CodePlugin, we strive to group the *stripes* in such a manner: every stripe must consist of k blocks located in each sub-file, respectively. This leads to the result that every sub-file is re-constructible if no more than m sub-files are lost.

To achieve this goal, we need to move blocks around. Since we want to keep the file untouched, we just need to record the original address of the block instead of moving the actual block around. For this reason, we call this *pseudo-shuffling*.

Figure 2 illustrates the process of pseudo-shuffling. After shuffling, all unique blocks are on the left side, and redundant ones can be found on the opposite side. We only need to encode the leftmost 3 columns, which contain unique blocks. Notice that even if there is a redundant block in the third column, it is still marked as unique due to its unique peer blocks. Furthermore, the actual blocks are never moved. We only record the new positions in the corresponding map-address file. For example, we can find the number 3 at row 0 and column 2 in Figure 2, which means the block is virtually moved to the last spot of this sub-file. Procedure 1 depicts the pseudo-shuffling algorithm of CodePlugin.

3.1.3 (Optional) Sub-files Exchange

Pseudo-Shuffling marks all redundant blocks as unique ones if there are unique peer blocks in the same column. Then if we have a sub-file consisting of unique blocks, we have to encode the whole file no matter how many redundant blocks are found in this file. To deal with this imbalance situation, we can resort to the sub-files exchange across files. We expect that sub-files from different files have more balanced unique block numbers. Figure 3 shows an example.

If we allow exchange sub-files among at most n_{ex} files, we will cache these n_{ex} files first. Sub-files exchange follows the Procedure 2. Basically, it sorts all $k \cdot n_{ex}$ subfiles based on the unique block numbers.

Then according to Procedure 2, CodePlugin recombines the sub-files into another n_{ex} buffers to be encoded. In this way the sub-files will be more balanced and a less number of blocks will need to be encoded. Note that this process is still "pseudo" because it does not change the original data file. We only need to keep the addresses of peer sub-files in the map-address table for future decoding.

Procedure 1 Pseudo-Shuffling(k, S, s, i, map)

```
▷ Shuffle the ith file
n ← S/(s·k)          ▷ n is blocks number per sub-file
for j ← 1, k do
    I_uni ← 1
    I_red ← n
    for l ← 1, n do
        if map[i][j][l].state = UNIQUE then
            ▷ map refers to the map-files
            map[i][j][l].position ← I_uni
            I_uni ← I_uni + 1
        else
            map[i][j][l].position ← I_red
            I_red ← I_red − 1
        end if
    end for
end for
```

Procedure 2 Sub-files Exchanging(k, i, map, n_{ex})

```
▷ Switch sub-files among the n_ex files
oSubs ← NULL
▷ oSubs stores the ordered sub-files
nFiles ← NULL
▷ nFiles represents the re-organized files
for j ← 1, n_ex do
    for l ← 1, k do
        oSubs[(j−1)·k+l] ← map[i+j−1][l]
    end for
end for
oSubs ← sort oSubs
for j ← 1, n_ex do
    for l ← 1, k do
        nFiles[j][l] ← oSubs[(j−1)·k+l]
    end for
end for
return nFiles
```

3.2 Encoding and Decoding

After the pre-processing, the encoding operation is straightforward. With CodePlugin, we only need to encode the data stripe by stripe and distribute the sub-files onto physical devices. The sub-files of data files stay untouched just like the normal (k,m) RS code.

The decoding process is a little more complicated than the normal RS code. We need to follow the map-file to locate the peers of the lost sub-files (if the sub-file exchanging is enabled). Then we need to re-shuffle the blocks as before in the encoding process. Finally we can decode the lost sub-file using the standard decoding operation.

4 Preliminary Evaluation

CodePlugin aims to reduce the redundant data in the primary cloud storage to save storage cost and thus also reduce the amount of data required for encoding (or improve the encoding throughput). For these objectives, CodePlugin introduces some additional pre-processing before the encoding, which is the overhead. In this section, we conduct some preliminary experiments to evaluate the performance and overhead of CodePlugin.

4.1 Workload and Experiment Setup

For our experiments, we directly downloaded fresh VM images based on VMware Player [2]. We downloaded different Operating Systems, including different versions of Ubuntu, Opensuse, Mint, Fedora and Debian, a total of 17 fresh VM images. These are brand new images without applications installed. These images are dumped into fixed sized 300 MB data files. Totally there are about 40 GB data. For the erasure coding, the coding parameters of (k,m,w) are set to $(6,3,8)$ if they are not specified, which is used in WAS [4]. The Jerasure library [12] is used for implementation. In the de-duplicating process, the fixed block size is set to 8 KB, and we use the MD5 digest algorithm to calculate fingerprints of blocks. We set the cache size to 512 MB for fingerprint entries by default.

Table 1: Storage Space of RS and CodePlugin

	Coded Files (GB)	Map-Address Files (MB)
Original RS	19.48	0
CodePlugin-No-EX	17.77	464.6
CodePlugin-2-EX	16.74	464.6
CodePlugin-4-EX	16.10	464.6

All these experiments are run on the machine with a 64-bit Intel Pentium CPU 2.8 GHz dual core, with 6 GB memory, 2 × 32 KB L1 caches, 2 × 256 KB L2 caches and shared 3 MB L3 cache. The Operating System is Ubuntu 12.04 with Linux kernel 3.2.0-53-generic.

4.2 CodePlugin Benefits

We first evaluate how much benefits we can gain from CodePlugin. That is, how much storage space we could save and how much improvement on the encoding throughput. For comparisons, we evaluate the CodePlugin against the original RS code scheme with the same coding parameters. In the experiments, we change the sub-file exchanging number among 0, 2 and 4, and they are denoted as *CodePlugin-no-EX*, *CodePlugin-2-EX* and *CodePlugin-4-EX*, respectively.

We first look into the size of the coded file. Table 1 shows the size of the coded files of different coding schemes in the second column. As we can see from this table, CodePlugin with different configurations can reduce the storage consumption by up to 17.4%. Furthermore, the optional sub-file exchange can indeed improve the deduplication results. Compared to the CodePlugin without allowing sub-file exchanges, CodePlugin-2-EX and CodePlugin-4-EX improves by 5.8% and 9.4%, respectively.

With the redundant data deduplicated, we expect that the encoding throughput would be improved as well. Figure 4 shows the encoding throughput. In this figure, the x-axis represents the encoding progress in terms of the size of input files. The y-axis represents the average throughput of encoding and writes. As shown on the figure, we find that CodePlugin with different configurations has about 20%, on average, higher throughput than the original RS scheme. This indicates that CodePlugin finds about 20% redundant data in our workload. With offline analysis based on the same size fixed-chunking

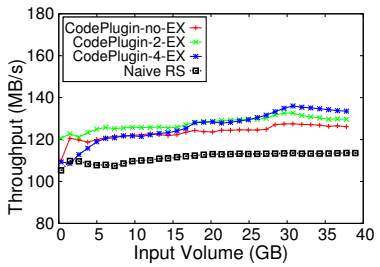

Figure 4: Throughput of encoding and write with different coding schemes

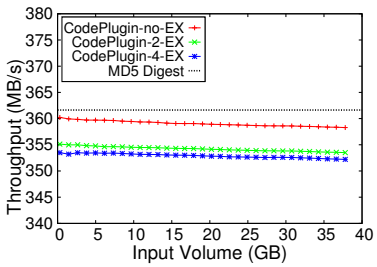

Figure 5: Throughput of pre-processing with different coding schemes

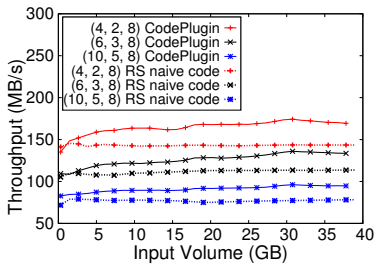

Figure 6: Throughput of encoding and write different coding parameters

technique, we find this is indeed the case – the fresh VM images contain about 23.34% redundant data.

4.3 CodePlugin Overhead

To deduplicate the redundant data, CodePlugin mainly introduces overhead from two aspects: the CPU cost in terms of MD5 based fingerprinting, and the storage cost in terms of the Map-Address file.

The third column of Table 1 shows the size of the Map-Address file in our CodePlugin scheme. It does not change with the sub-file exchanges, and the size is about 1.15% of the original data size.

To study the computation cost of the pre-processing in CodePlugin, Figure 5 shows the pre-processing throughput of different CodePlugin schemes and the MD5 fingerprinting alone. In this figure, the x-axis is also the size of the input file in GB. The y-axis is the throughput of pre-processing. The figure clearly shows that the major computing overhead introduced by CodePlugin is due to MD5 computation. As expected, CodePlugin-No-EX has a little higher throughput than the other two CodePlugin schemes that allow sub-file exchanges.

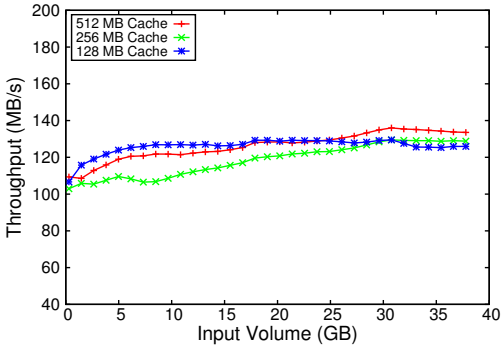

Figure 7: Throughput of encoding and write with different cache size

4.4 Varying Cache and Coding Parameters

Since the cache size may affect the deduplication results, we also evaluate the impact of different cache sizes.

Figure 7 shows the results when the cache size varies from 128 MB to 512 MB, while other parameters are the same as before. Note for clarity of presentation, Figure 7 only shows the results of CodePlugin-4-EX. A cache size of 128 MB performs better at beginning, but was surpassed by the larger size of 256 MB and 512 MB later. On one hand, this indicates that the cache size does impact the deduplication. This is reasonable as a smaller cache size may cause more evictions from the cache. On the other hand, a larger cache size does not always perform better, for example, 256 MB performs worse than 128 MB cache most of the time, as it also relates to the access sequence of the fingerprints.

Last, we also evaluate the impact of different coding parameters. Besides (6, 3, 8), we also evaluate (4, 2, 8), and (10, 5, 8). Figure 6 shows the results of CodePlugin-4-EX. In general, we can observe that for all sets of parameters, the CodePlugin can outperform the original RS code.

5 Conclusion and Future Work

In this work, we explore the possibility of supporting deduplication in cloud storage systems due to the increasing amount of redundant data on cloud storage. For this purpose, we have designed CodePlugin, a mechanism that is applicable to any existing erasure coding scheme. Experiments based on some real-world cloud VM images have been conducted, and the results show that CodePlugin can effectively remove the redundant data and subsequently improve the encoding throughput.

CodePlugin is an initial design, and we are optimizing and testing it from various aspects. We will conduct experiments in real deployed storage systems to examine its effect in practice.

6 Discussion

Primary storage deduplication is attracting more and more attention. Some startups are working in this area. Work [5] also showed that there is a growing rate of redundant data in the cloud storage due to various reasons and the increasing demand of efficient solutions for primary storage deduplication. However, the solution is still open and CodePlugin explores one possible way to address this challenge. The experimental results presented in this paper show that CodePlugin can achieve 17.4% storage saving and about 20% improvement of the encoding throughput. However, this is not the limit of CodePlugin. The performance of CodePlugin is highly workload dependent. In the previous experiments, the workload we use only contains fresh VM images for VMware. Our offline analysis shows that there is about 23% redundant data in the workload, and our preliminary experiments show that CodePlugin can exploit such redundancy.

The first question remained for CodePlugin is whether this is the most efficient and suitable approach for primary storage. What are other possible alternatives? What are their corresponding advantages and disadvantages? CodePlugin aims to introduce the minimum overhead (changes) to existing systems. Is a clean-slate design a better approach?

The second is that inside CodePlugin, currently the fixed sized chunking is used in deduplication. An alternative is to use the variable sized chunking that has higher efficiency in detecting the redundant data. We can expect lower preprocessing throughput due to complexity of the variable chunking algorithms, e.g., by using Rabin Fingerprinting. Also it would be more complicated in implementation, because the deduplication units are not well aligned to be efficiently encoded. However, it may be still worth exploring whether such weakness is addressed while a higher storage efficiency can be achieved.

Lastly, in this work, we mainly introduce the idea of CodePlugin. Our evaluations are centered on the performance when dealing with deduplication. There are other aspects to be evaluated, such as read and recovery after a failure. For unique blocks, we expect they perform not very differently from the traditional erasure coding shcemes. However, the reconstruction of redundant blocks, which reads blocks from the storage nodes, may degrade the I/O performance. Are there any optimizations to mitigate such overhead? One possible solution is to leverage the rendundant data temporal and spacial locality [15]. Another direction is to only mark blocks from distinct storage nodes as redundant so that a higher read speed can be achieved with concurrent I/O requests. It is also very desirable to measure its performance under normal operating environments. As a next step, we plan to conduct more thorough evaluations of CodePlugin.

7 Acknowledgement

We appreciate constructive comments from anonymous referees. The work is partially supported by NSF under grant CNS-1117300.

References

[1] EMC Atmos storage service. `http://www.emccis.com/`.

[2] VMware Player. `http://www.vmware.com/products/player`. Accessed: 2010-09-30.

[3] Dropbox. `https://www.dropbox.com/`, 2007.

[4] CALDER, B., WANG, J., AND ET AL. Windows Azure Storage: a highly available cloud storage service with strong consistency. In *Proc. of SOSP* (2011).

[5] EL-SHIMI, A., KALACH, R., KUMAR, A., OLTEAN, A., LI, J., AND SENGUPTA, S. Primary data deduplication-large scale study and system design. In *Proceedings of the 2012 USENIX Conference on Annual Technical Conference* (Berkeley, CA, USA, 2012), USENIX ATC'12, USENIX Association, pp. 26–26.

[6] GHEMAWAT, S., GOBIOFF, H., AND LEUNG, S.-T. The google file system. In *Proc. of SOSP* (2003).

[7] GOOGLE, I. Google Docs. `http://www.google.com/docs/about/`, 2007.

[8] HUANG, C., SIMITCI, H., XU, Y., OGUS, A., CALDER, B., GOPALAN, P., LI, J., AND YEKHANIN, S. Erasure coding in windows azure storage. In *Proc. of USENIX ATC* (2012).

[9] KHAN, O., BURNS, R., PLANK, J. S., PIERCE, W., AND HUANG, C. Rethinking erasure codes for cloud file systems: Minimizing I/O for recovery and degraded reads. In *Proc. of FAST* (2012).

[10] MEYER, D. T., AND BOLOSKY, W. J. A study of practical deduplication. *Trans. Storage 7*, 4 (Feb. 2012), 14:1–14:20.

[11] MUTHITACHAROEN, A., CHEN, B., AND MAZIÈRES, D. A low-bandwidth network file system. In *Proc. of SOSP* (2001).

[12] PLANK, J. S., SIMMERMAN, S., AND SCHUMAN, C. D. Jerasure: A library in C/C++ facilitating erasure coding for storage applications - Version 1.2. Tech. Rep. CS-08-627, University of Tennessee, August 2008.

[13] RABIN, M. *Fingerprinting by Random Polynomials*. Center for Research in Computing Techn., Aiken Computation Laboratory, Univ., 1981.

[14] REED, I. S., AND SOLOMON, G. Polynomial Codes Over Certain Finite Fields. *Journal of the Society for Industrial and Applied Mathematics 8*, 2 (1960), 300–304.

[15] SRINIVASAN, K., BISSON, T., GOODSON, G., AND VORUGANTI, K. idedup: Latency-aware, inline data deduplication for primary storage. In *Proceedings of the 10th USENIX Conference on File and Storage Technologies* (2012), FAST'12, USENIX Association.

[16] TRIDGELL, A. *Efficient Algorithms for Sorting and Synchronization*. PhD thesis, 1999.

[17] ZHU, B., LI, K., AND PATTERSON, H. Avoiding the disk bottleneck in the data domain deduplication file system. In *Proceedings of the 6th USENIX Conference on File and Storage Technologies* (2008), FAST'08.