NEW DEVELOPMENTS ON COMPRESSION AND TRANSFER OF SIMULATION DATA WITHIN AN SDM SYSTEM

Dr. rer. nat. M. Büchse, Dipl.-Ing. M. Thiele, Dr.-Ing. H. Müllerschön
(SCALE GmbH, Germany);

Abstract

In the past 10 years SCALE developed a comprehensive simulation and test data framework (SCALE.sdm) in very close collaboration with AUDI and other brands of the VW group. There are several apps to cover the entire CAE design process. Today the system is being intensively used at many different simulation units, with currently over 800 registered users at the VW Group and at a large number of contracted service providers. An important app for simulation data and process management of FEM models is the innovative software solution LoCo. In addtion, there is a substantial app called CAViT, which does complex data management of result data from simulation as well as of physical test data.

The system applies several new approaches to simulation and test data management, such as strict offline capabilities with permanent synchronization of relevant data, consequent version management of all involved objects by means of simulation models and processes, novel ontology based approaches for the assembly of components as well as easy customizability. The SDM solution is an open system for the integration of any third party or in-house CAE-product, such as pre-/post processors, FE-solvers, queuing systems, process scripts, etc.

This paper focuses on recent developments and new approaches on data compression and how to make effective data transfer available. There are new implementations on data deduplication which will cut down the storage and bandwidth requirements by another factor of 6-8 compared to the current implementation. Related to uncompressed data this is a factor of compression of about 20-25. Another example is the development of encrypted local data storage by using smart cards allowing for two factor encryption. Essential contributions of the new developments are achievements of a big data research project founded by the German government (BMBF).

Finally, the paper concludes with asummary.

1 Introduction

The SDM system offers users a graphical working environment for daily tasks of simulation engineers, see **Fehler! Verweisquelle konnte nicht gefunden werden.**. Access to simulation, test and process data has to be easy and with good performance. There should be no limitations on sharing model data or related documentation – either within the company or with external business partners. On this, data compression and effective transfer of data is essential.

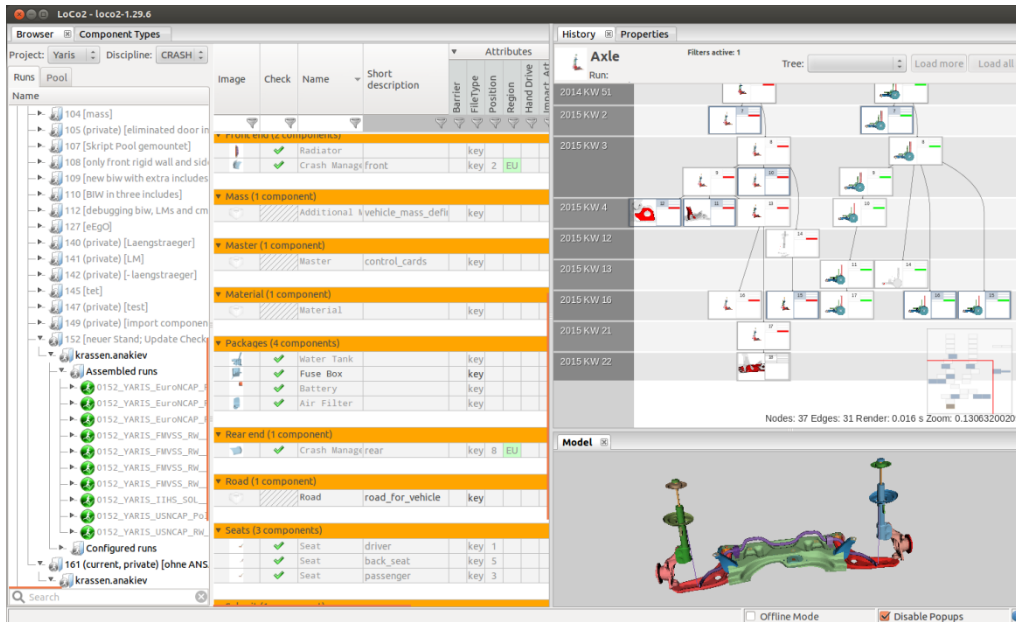

Figure 1: LoCo user interface - workbench for simulation engineers

Models in Simulation Data Management (SDM) systems have grown tremendously in recent years (see Figure 2:). At the same time, these models typically exhibit a great deal of redundancy. This is not being fully exploited by established compression techniques, such as ZIP. In view of the size of state-of-the-art SDM systems, data storage and transfer cause large costs, which make more advanced compression approaches necessary.

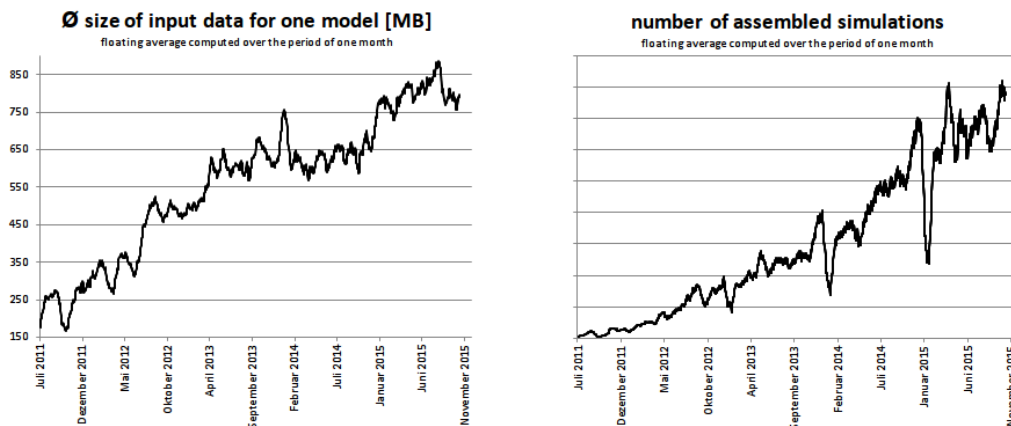

Figure 2: Model size and assembled simulations at an automotive company (absolute numbers withheld) over time.

Data deduplication exploits the mentioned redundancy. It reduces space by removing repetitive data patterns. Every pattern is saved only once, and wherever it reappears, it is replaced by a link to its first occurrence. So far, this approach has largely been applied to backup scenarios, where data is assumed to be immutable and throughput is considerably more important than latency; or it has been applied to large-scale computing with multiple nodes. In the SDM domain, however, we need random access to the data, including deletion, and we usually deal with a single machine, even for the server. Therefore, existing solutions cannot be readily applied.

We implemented a deduplicated storage system and incorporated it into SCALE’s SDM solution LoCo, which runs on both Linux and Windows. In the process we solved challenges such as choice of parameters, storage, deletion, data integrity, concurrency, deduplicated transfer, and encryption.

We measured runtime performance and deduplication gain (i.e., space saved compared to non-deduplicated storage) on several datasets. In summary, the runtime performance is completely adequate for an SDM client (around 50 MiB/s write and 150 MiB/s read) and promising for an SDM server. Figure 3: shows the compression ratio improvements of data deduplication in comparison to the state of the art.

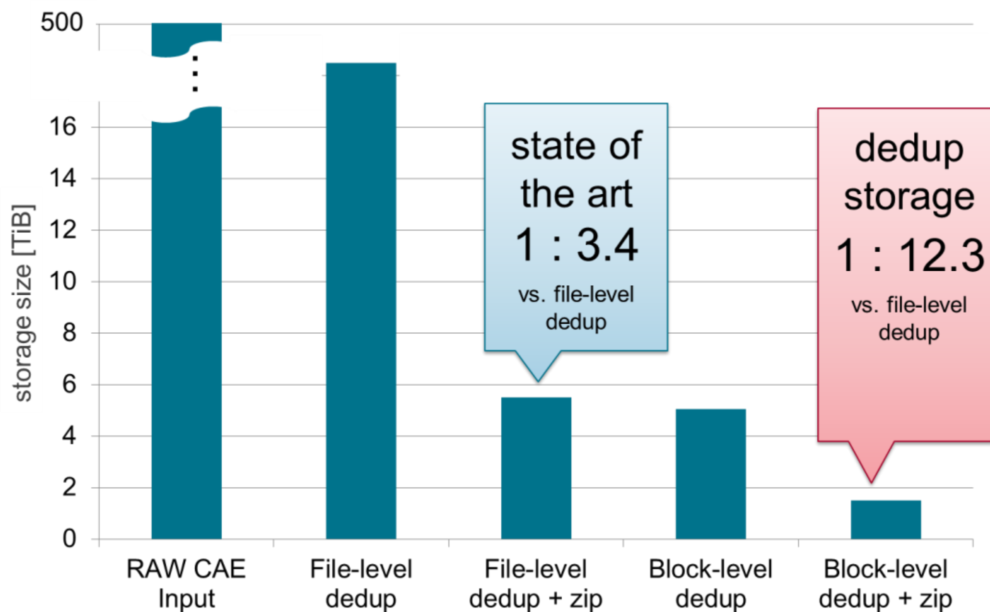

Figure 3: Compression ratio improvements achieved using data deduplication.

The following section will detail our implementation of data deduplication in the context of the SDM system LoCo. The final section will provide a summary and an outlook.

2 Data deduplication

2.1 Previous work

Paulo and Pereira [7] define “deduplication as a technique for automatically eliminating coarse-grained and unrelated duplicate data.” Deduplication is not new: variants have been described as early as 1996 [11]. By now, many implementations exist, with various application scenarios in mind – cf. Ref. [7] for a comprehensive overview. However, none of these implementations match our requirements.

For instance, most implementations address backup and archival scenarios, where data is assumed to be immutable, and throughput is considerably more important than latency [7,Table 1:]. Many implementations come packaged as a product, such as a network-attached storage or a file system. Other systems are targeted at large-scale installations with multiple nodes, or their licence is unsuitable for commercial purposes. In addition, no system covers deduplication of both storage and transfer.

The bottom line is that no off-the-shelf deduplication solution can be integrated into a single-node SDM server (or client) application. However, as we shall see

in the following, the techniques and systems described in the literature provide a copious supply of valuable ideas and concepts. Roughly speaking, the general approach of data deduplication consists of two major steps:

- Chunking Partition an input file into chunks.
- Indexing Look up in an index which chunks are already in store.

In a third, but less demanding step, add the new chunks to the store. In many cases, one ought to also store a recipe for reconstructing the file from the chunks.

The deduplication gain is the space saved by deduplication divided by the space required by the raw data. This quantity depends on two factors: (a) the number of duplicates found and (b) the storage overhead incurred by the index data structure. For instance, we may find more duplicates by using smaller chunks, but then we need a bigger index (which may also pose problems with respect to latency or throughput).

Chunking

The simplest of chunking merely partitions the file into fixed-size chunks. The problem with this method is that inserting or deleting bytes in the file will lead to completely new chunks. A widely used alternative is content-defined chunking.

Here we need five parameters: the minimum chunk size, the maximum chunk size, the checksum method (such as the one from rsync [11] or Rabin-Karp [5]), the window size w and the divisor d (both nonnegative integers). We proceed as follows.

We move a “sliding window” of size w over the input data, and at each position we compute the checksum of the window. For the above-mentioned checksum methods, this computation can be done in constant time: we only have to consider the byte that leaves the window and the byte that enters the window. The current right-hand window boundary marks a chunk boundary if (a) the minimum-size requirement is satisfied and the checksum is divisible by d, (b) moving the window further would violate the maximum size requirement, or (c) we reached the end of the file.

The divisor can be construed as a target chunk size, if we assume that the window checksums are more or less evenly distributed.

Indexing

As noted in Ref. [7, Sec. 2.4], systems generally summarize the chunk content via a cryptographic hash (such as SHA-1), and the hash values are used to

query or update the index. In its classical form, the index then is a mapping that associates, for each chunk in store, its hash value with its storage address. This is also called a full index. In contrast, a partial index may keep only a subset of the chunks; this saves index space and lookup time, but if a chunk is not present in the index, it may be added to the store a second time, i.e., the deduplication is partial.

A more elaborate approach to partial deduplication is the sparse index [6]. Here chunks are grouped together into segments, and each segment is represented by a sample of its chunk hashes. The sparse index then is a mapping that associates each sampled hash value with storage information (manifest) about the segment that contains the chunk data with that hash value. A chunk can be referenced in several segments, but the chunk data is stored in exactly one segment.

For an incoming segment, one samples its chunk hashes in order to find and select matching segments in the sparse index. Before the segment is stored, every chunk that occurs in any of the selected segments is replaced by a reference. We note that this manner of deduplication is partial, for the new segment can still contain a chunk that is present in some segment that did not get selected.

Ghosh [3] proposes a similar technique, but instead of sampling the hashes, he uses a similarity sketch.

Challenges

Parameter values	The choice of minimum chunk size, maximum chunk size, checksum method, window size, and divisor is not obvious, and it may depend on the characteristics of the data.
Storage and recipe	Chunks should be stored in a way that facilitates reconstructing files; e.g., we might store adjacent chunks in a file in adjacent locations, thereby reducing seeks. Depending on the storage layout, a dedicated recipe may be necessary for reconstructing a file.
Deletion	Before a chunk can be deleted, we have to make sure that it is not referenced any more. Common techniques to achieve this are reference counting as well as mark and sweep [4, Sec. 3.3].
Data integrity	The data structure is more complex than a simple collection of files, and a damaged chunk can affect a number of files.

Concurrency Files are no longer independent: the index acts as a synchronization barrier. The index must be protected from concurrent access (in particular, if multiple processes are involved).

Deduplic. transfer We want to transfer only the chunks that the target side lacks, with a reasonable amount of roundtrips (cf. rsync). The deduplicated representations on client and server have to be compatible (support the same abstraction).

Encryption Storer et al. [9] show how to achieve end-to-end encryption for multiple parties via convergent encryption. Their setting assumes a full index. It is unclear whether this approach is compatible with, e.g., a sparse index.

Parameter	Values considered	Value at optimal point
checksum method	rsync, Rabin-Karp	Rabin-Karp
minimum chunk size (bytes)	$2^7, 2^8, \ldots, 2^{13}$	2^{12}
maximum chunk size (bytes)	2^{20}	2^{20}
window size (bytes)	$2^6, 2^7, \ldots, 2^9$	2^6
divisor (target chunk size; bytes)	$2^{13}, 2^{14}, \ldots, 2^{23}$	2^{14}

Table 1: Restricted parameter space, optimized values.

3 Approach and implementation

Content-defined chunking is employed for its superior deduplication gain. In conjunction with our deletion requirement, this decision has consequences for the index design: with variable-size chunks, overwriting deleted chunks is not a viable option. For defragmentation, chunks must be moved, and so the storage address is variable as well. Therefore, it cannot be used as a long-lived chunk

reference. Instead, we reference chunks via their hashes. This, in turn, forces us to use a full index.

When the index becomes prohibitively large, we permanently “freeze” (i.e., mark read only) the current store and open a new one. In that case, our deduplication is partial, because new files are not deduplicated against the frozen stores. This approach is trivial to implement and, provided that related files end up within the same store, leads to reasonable deduplication gain and runtime performance.

3.1 Parameter values

In order to explore the parameter space, we considered a dataset of 436 GiB of uncompressed real world data (mostly simulation models). As objective function we used the function that maps every point in the parameter space to the deduplication gain that is obtained by using the corresponding parameter values to deduplicate our dataset. Our aim was to maximize the objective function.

However, this function is quite expensive to compute. Therefore, we first restricted the parameter space as shown in Table 1: . Second, we used a meta model approach: we computed the objective function on a 100-point random sample of the restricted parameter space and fitted a radial basis function to that sample. Then we maximized the latter function, obtaining the values shown in Table 1: . The whole optimization problem was modelled and solved using LS-OPT [8].

3.2 Storage and recipe

Our abstraction for storing and retrieving chunks is the chunk store. This is a data structure that basically maps chunk hashes (or, more generally, keys) to the respective chunk data, plus a reference count.

In addition to individual chunks, it supports named chunk sequences of bounded size and with arbitrary ancillary data. Each chunk in such a sequence can be accessed individually via its key, but the sequence can also be read or written like a stream. The sequence is preserved on disk, so that seeks are reduced. The sequence size is bounded to facilitate defragmentation. Finally, the chunk store supports transactions for atomically adding or deleting multiple chunks.

When we chunk a file, we obtain a list of chunks. In order to reconstruct the file, we need the list of chunk hashes, or recipe. Our abstraction supports three strategies for storing the chunks and the recipe:

Index chunk	We can encode any sequence of chunk hashes in binary form and store that representation as a chunk in the chunk store. This kind of chunk we call index chunk. We store each chunk individually, and we store the whole recipe as one index chunk.
Hierarchical index	Since an index chunk is merely a special chunk, it can refer to other index chunks, which gives rise to a hierarchical index. We factor parts of the recipe out as “subordinate” index chunks; this approach can be linked to a deduplication of index chunks.
Chunk stream	We store the incoming new chunks in named sequences, commencing a new sequence whenever the size boundary is hit. With each sequence we store (as ancillary data) the corresponding part of the recipe, thus accommodating references to existing chunks. We treat the list of sequence names like an index chunk. The recipe is the concatenation of the partial recipes.

In order to implement the chunk store, we created the following concepts and mechanisms:

Flatstore	A file of bounded size that is basically a concatenation of chunks, plus checksums. With the exception of reference counts, flatstores cannot be updated. Existing flatstores are opened read only; new flatstores can be read from and appended to. Concurrent writes are not supported.
Composite flatstore	Combines multiple flatstores to overcome their limitations: it removes the size boundary, it can always be appended to, and it permits concurrent writes. Also, it encapsulates the bulk of our defragmentation procedure.
Journal	Provides transactions for atomically manipulating multiple reference counts.
Dictabase	The dictabase maps chunk keys to flatstore addresses; currently we mainly use leveldb [1].
Dictabase locker	Allows selectively locking the dictabase for given keys.

Put bluntly, “chunk store = composite flatstore + journal + dictabase + dictabase locker”.

In order to store a chunk with some key, we lock the dictabase for that key and look up the address. If the lookup succeeds, we merely increase the reference

count for the chunk. Otherwise, the data is written to the composite flatstore, and the resulting address is put into the dictabase under said key. Finally, we release the lock on the dictabase. The deletion of chunks is analogous, but the key is not removed from the dictabase; this happens only at defragmentation.

In order to retrieve a chunk, we query the dictabase to obtain the address, and then we read the chunk data from the composite flatstore. Note that there is no need to lock the dictabase for the key.

Technically, a new chunk is first stored at a reference count of zero, and then this count is increased by one. Consequently, the journal allows atomically adding or removing multiple chunks.

3.3 Deletion

We use the reference counting facility provided by the chunk store to keep track of the number of times that a chunk occurs in the stored recipes. That is, a chunk has a reference count of zero precisely when there is no recipe that refers to it. Such a chunk is essentially garbage. In order to reclaim the space occupied by garbage, we need to defragment the composite flatstore.

During defragmentation, the storage address of any chunk can change, and that has to be reflected in the dictabase. Moreover, if the defragmentation process is interrupted, we have to make sure that the composite flatstore and dictabase are left in a state that is consistent or easily made consistent.

For this reason, we do not defragment "in place", but we copy chunks from one flatstore into a new one. Before we start, we flag the old flatstore as "under defragmentation". The new flatstore is only incorporated into the composite flatstore when it is finished, and only then is the old flatstore deleted. Finally, we update the dictabase.

Any interruption is easily detected: either we find a not-yet-incorporated new flatstore – then we delete it and remove any defragmentation flags we find –, or we delete any flagged flatstore. Interruptions during the dictabase update are not critical: the address of every copied chunk will be carried over the next time the chunk store is opened (see Sec. 3.4). Remaining addresses of garbage chunks will be detected easily (upon lookup) because the whole address space will have vanished together with the old flatstore.

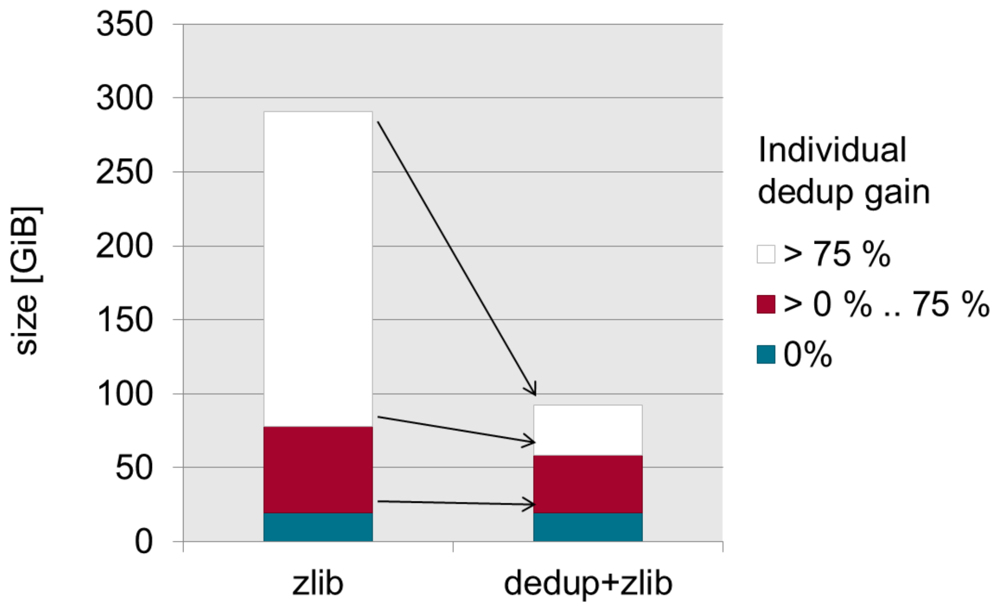

Figure 4: File classes in deduplicated vault grouped by their individual deduplication gain.

3.4 Data integrity

There is a multitude of reasons why data integrity may be at risk: when the disk is full, files may be written partially. Or when the power goes out, write buffers in the operating system do not make it onto the disk. Or the application crashes in the midst of adding a file to the store, before the recipe is complete, leaving behind a number of garbage chunks or even inaccurate reference counts.

Naturally, we want to detect and handle any of these situations. To that end, flatstores include CRC-32 checksums for chunks and CRC-8 checksums for metadata. Any read operation is anticipated to be partial. The composite flatstore detects incomplete defragmentation operations (see 3.3).

Upon opening the chunk store, we perform maintenance: we try to read any chunks that were newly appended since the last “sync”, i.e., when the operating system buffers were forced on disk. If we find an inconsistency, we truncate the respective flatstore. Also, we keep information in the dictabase about the address space it covered at the last sync, and we update the dictabase should it miss any addresses.

Finally, we replay the journal transactions since the last sync, rolling back any incomplete transaction. Each step is idempotent by design; therefore, maintenance can safely be repeated after an interruption.

3.5 Encryption

We use the authenticated model described in Ref. [9, Sec. 4.2]. Our main concern has been to protect the local storage, and we only support one key pair: that of the (fictitious) local storage owner. For an end-to-end encrypted deduplicated transfer, we would need a globally valid key pair per user.

4 Experiments and results

We implemented our deduplicated storage in Python (with the chunking in Cython), and we incorporated it into the SDM client software LoCo [10], which runs on both Linux and Windows. Subjectively, no difference in the performance between the conventional and the deduplicated storage could be noticed.

Intuitively, it is clear that some files are more amenable to deduplication than others. More precisely, if we partition the set of files in a data storage into classes and deduplicate each class on its own, we obtain different deduplication gains. For our analysis, we selected the partition such that two files are in the same class precisely when they have a chunk in common. On a real-world dataset with 260 K files, 292 GiB (zlib compressed), and a total deduplication gain of 75 %, we found 208 K classes.

These classes can be divided into three groups (see Fig. 4): almost 200 K classes consist of only one file each; these make up 19.6 GiB. There are 8 K further classes whose individual gain is at most 75 %; these have 40 K files and 59.8 GiB in total. The remaining 323 classes consist of 20 K files and 213.0 GiB. The three groups have total deduplication gains of 0 %, 35 %, and 93 %, respectively, corresponding to deduplication rates of 1, 1.5, and 14, respectively.

In summary, the deduplication gain that can be achieved very much depends on the data at hand. For instance, the first group contained a lot of preview images and no simulation models whatsoever, while most simulation models were in the third group. Therefore, we surmise that a deduplication rate in the range of 3 to 8 is possible for mixed SDM data; for pure simulation models 12 and more is possible.

5 Summary

We considered an advanced compression technique – data deduplication – and how to apply it to SDM models. There is no off-the-shelf solution that can be used in the SDM scenario.

For data deduplication, we solved challenges such as choice of parameters, storage, deletion, data integrity, and encryption. We implemented our

procedure in Python and incorporated it into the SDM client LoCo. The runtime performance is completely adequate for an SDM client. We measured the deduplication gain on several datasets. We achieve a deduplication gain of 75 % for mixed SDM data and 87–93 % for pure simulation models (which corresponds to deduplication rates of 4 and 8–14, respectively).

6 Acknowledgements

The work on data deduplication has been developed in the big data project VAVID (reference number: 01 IS14005 C), which is partly funded by the German ministry of education and research (BMBF).

7 References

[1] http://leveldb.org/.

[2] http://7-zip.org/sdk.html.

[3] M. Ghosh. Scaling Deduplication in Pcompress – Petascale in RAM. Apr. 2014. URL: https ://moinakg.wordpress.com/2014/04/06/scaling-deduplication-in-pcompress-petascale-in-ram/.

[4] F. Guo and P. Efstathopoulos. “Building a High-performance Deduplication System”. In: Proceedings of the 2011 USENIX Conference on USENIX Annual Technical Conference. USENIXATC’11. Portland, Oregon: USENIX Association, 2011, pp. 25–25.

[5] R. M. Karp and M. O. Rabin. “Efficient randomized pattern-matching algorithms”. In: IBM Journal of Research and Development 31.2 (Mar. 1987), pp. 249–260.

[6] M. Lillibridge et al. “Sparse Indexing: Large Scale, Inline Deduplication Using Sampling and Locality”. In: Proceedings of the 7th Conference on File and Storage Technologies. FAST ’09. San Francisco: USENIX Association, 2009, pp. 111–123.

[7] J. Paulo and J. Pereira. “A Survey and Classification of Storage Deduplication Systems”. In: ACM Comput. Surv. 47.1 (June 2014), 11:1–11:30. ISSN: 0360-0300.

[8] N. Stander and A. Basudhar. “LS-OPT - Status and Outlook”. In: 14th International LS-DYNA Users Conference. Livermore Software Technology Corporation. Detroit, June 2016.

[9] M. W. Storer et al. “Secure Data Deduplication”. In: Proceedings of the 4th ACM International Workshop on Storage Security and Survivability.

StorageSS ’08. Alexandria, Virginia: ACM, 2008, pp. 1–10. ISBN: 978-1-60558-299-3.

[10] M. Thiele, T. Landschoff, and A. J. Beck. “LoCo – An Innovative Process and Team Data Management Solution for Simulation”. In: NAFEMS European Simulation Process and Data Management Conference. 2015.

[11] A. Tridgell and P. Mackerras. The rsync algorithm. Tech. rep. TR-CS-96-05. The Australian National University, 1996.