GoSeed: Generating an Optimal Seeding Plan for Deduplicated Storage

Aviv Nachman and Gala Yadgar, *Technion - Israel Institute of Technology;*
Sarai Sheinvald, *Braude College of Engineering*

https://www.usenix.org/conference/fast20/presentation/nachman

This paper is included in the Proceedings of the
18th USENIX Conference on File and
Storage Technologies (FAST '20)
February 25–27, 2020 • Santa Clara, CA, USA
978-1-939133-12-0

Open access to the Proceedings of the
18th USENIX Conference on File and
Storage Technologies (FAST '20)
is sponsored by

GoSeed: Generating an Optimal Seeding Plan for Deduplicated Storage

Aviv Nachman, Gala Yadgar
Computer Science Department, Technion

Sarai Sheinvald
Braude College of Engineering

Abstract

Deduplication decreases the physical occupancy of files in a storage volume by removing duplicate copies of data chunks, but creates data-sharing dependencies that complicate standard storage management tasks. Specifically, data migration plans must consider the dependencies between files that are remapped to new volumes and files that are not. Thus far, only greedy approaches have been suggested for constructing such plans, and it is unclear how they compare to one another and how much they can be improved.

We set to bridge this gap for *seeding*—migration in which the target volume is initially empty. We present GoSeed, a formulation of seeding as an integer linear programming (ILP) problem, and three acceleration methods for applying it to real-sized storage volumes. Our experimental evaluation shows that, while the greedy approaches perform well on "easy" problem instances, the cost of their solution can be significantly higher than that of GoSeed's solution on "hard" instances, for which they are sometimes unable to find a solution at all.

1 Introduction

Data deduplication is one of the most effective ways to reduce the size of data stored in large scale systems. Deduplication consists of identifying duplicate data chunks in different files, storing a single copy of each unique chunk, and replacing the duplicate chunks with pointers to this copy. Deduplication reduces the total physical occupancy, but increases the complexity of management aspects of large-scale systems such as capacity planning, quality of service, and chargeback [53].

Another example, which is the focus of this study, is *data migration*—the task of moving a portion of a physical volume's data to another volume—typically performed for load balancing and resizing. Deduplication complicates the task of determining which files to migrate: the physical capacity freed on the source volume, as well as the physical capacity occupied on the target volume, both depend on the amount of deduplication within the set of migrated files, as well as between them and files outside the set (i.e., files that remain on the source volume and files that initially reside on the target volume). An efficient *migration plan* will free the required space on the source volume while minimizing the space occupied on the target. However, as it turns out, even *seeding*, in which the target volume is initially empty, is a computationally hard problem.

Data migration in deduplicated systems and seeding in particular are the subject of several recent studies, each focusing on a different aspect of the problem. Harnik et al. [32] address capacity estimation for general migration between volumes, while Duggal et al. [25] describe seeding a cloud-tier for an existing system. Rangoli [49] is designed for space reclamation—an equivalent problem to seeding. These studies propose greedy algorithms for determining the set of migrated files, but the efficiency of their resulting migration plans has never been systematically compared. Furthermore, in the absence of theoretical studies of this problem, it is unclear whether and to what extent they can be improved.

We present GoSeed, a new approach that bridges this gap for the seeding and, consequently, space reclamation problems. GoSeed consists of a formulation of seeding as an integer linear programming (ILP) problem, providing a theoretical framework for generating an optimal plan by minimizing its *cost*—the amount of data replicated. Although ILP is known to be NP-Hard, commercial optimizers can solve it efficiently for instances with hundreds of thousands of variables [1–4, 6]. At the same time, ILP instances representing real-world storage systems may consist of hundreds of millions of variables and constraints—too large even for the most efficient optimizers, that may require prohibitively long time to process these instances. Thus, GoSeed also includes three practical acceleration methods, each presenting a different tradeoff between runtime and optimality.

The first method, *solver timeout*, utilizes the optimizer's ability to return a feasible suboptimal solution when its runtime exceeds a predetermined threshold. A larger timeout value allows the optimizer to continue its search for the optimal solution, but increasing the timeout may yield diminishing returns. The second method, *fingerprint sampling*, is similar to the sketches used in [32], and generates an ILP instance from a probabilistically sampled subset of the system's chunks. An optimal seeding plan generated on a sample will not necessarily be optimal for the original system. Thus, increasing the sample size may reduce the plan's cost, but will necessarily increase the required processing time of the solver.

Our third method, *container aggregation*, generates an ILP instance in terms of containers—the basic unit of storage and I/O in many deduplication systems. Containers typically store several hundreds of chunks, where chunks in the same container likely belong to the same files. When they do, containers

represent the same data sharing constraints as their chunks. In addition to reducing the problem size, migrating entire containers can be done without decompressing them, and without increasing the system's fragmentation. At the same time, a container-based ILP instance may introduce "false" sharing between files, resulting in a suboptimal plan.

We implement GoSeed with the Gurobi [2] commercial optimizer, and with the three acceleration methods. We generate seeding plans for volumes based on deduplication snapshots from two public repositories [7, 47]. Our evaluation reveals the limitations of the greedy algorithms proposed for seeding thus far—while they successfully generate good plans for "easy" problems (with modest deduplication), GoSeed generates better solutions for the harder problems, for which the greedy approaches sometimes return no solution.

Our analysis further demonstrates the efficiency of the acceleration methods in GoSeed. It shows that (1) the suboptimal solution returned by GoSeed after a timeout is often better than the greedy solutions, (2) fingerprint sampling "hides" some of the data sharing in volumes with modest deduplication, but provides an accurate representation of systems with substantial deduplication, and (3) GoSeed's container-based solutions are optimal if entire containers are migrated. Our results suggest several rules of thumb for applying and combining these three methods in practical settings.

The rest of this paper is organized as follows. Section 2 provides background on deduplication and ILP, as well as related previous work. We present the ILP formulation of GoSeed in Section 3, and its acceleration methods in Section 4. Our experimental setup and evaluation are described in Section 5, with a discussion in Section 6. Section 7 concludes this work.

2 Background and Related Work

2.1 Deduplication

The smallest unit of data in a deduplication system is a *chunk*, which typically consists of 8-64KB. The incoming data is split into chunks of fixed or variable size, and the *fingerprint* of each chunk is used to identify duplicates and to replace them with pointers to existing copies. In many systems, new chunks are written to durable storage in *containers*, which are the system's I/O unit, and typically consist of hundreds of chunks [20, 29, 38, 41, 68]. New chunks are added to containers in a log structure. Thus, chunks belonging to the same file will likely reside in adjacent containers. Designs that do not employ containers typically also persist the chunks in a log structure, and thus adjacent chunks will likely belong to the same files [16, 18, 24, 54].

To recover a file, all the containers pointed to by the file *recipe* are fetched into memory, after which the file's chunks are collected. The efficiency of this process, in terms of I/O and memory usage, strongly depends on the file's *fragmentation*: the physical location of the different containers and the portion of the container's chunks that belong to the requested file [50]. Some systems reduce the amount of fragmentation by limiting the number of containers a file may point to, or their age [28, 40].

Over the last decade, numerous studies addressed the various aspects of deduplication system design, such as characterizing and estimating the system's deduplication potential [27, 33, 46, 47, 57, 61], efficient chunking and fingerprinting [9, 44, 48, 63, 64], indexing and lookups [10, 54, 68], restore performance [28, 36, 40, 67], compression [42, 65], and security [14, 34, 39, 55]. Their success (among others) has made it possible to use deduplication for primary storage and not just for archives. Additional studies explored ways to adapt the concept of deduplication to related domains such as page caching [35, 38], minimizing network bandwidth [9, 48], management of memory resident VM pages [17, 31, 62], and minimizing flash writes [16, 26, 30, 38, 52, 60].

Recently, Shilane et al. [53] described the "next" challenge in the design of deduplication systems: providing these systems with a set of management functions that are available in traditional enterprise storage systems, one of which is capacity management in deduplicated systems, and specifically, fast and effective data migration.

2.2 Data migration

Data migration is typically performed in the background, according to a *migration plan* specifying which data is moved to which new location. Typical objectives when generating a migration plan include minimizing the amount of data transferred, optimizing load balancing, or minimizing its effect on ongoing jobs.

The effectiveness of data migration and the resources it consumes may greatly affect the system's performance. Thus, efforts have been made to optimize its various aspects including service down-time, geolocation, provisioning, memory consumption, and system-specific performance requirements and constraints [43, 45, 56, 59]. Hippodrome [12] and Ergastulum [13] formulated the storage allocation problem as an instance of bin-packing, while Anderson et al. [11] experimentally evaluated several theoretical algorithms, concluding that their theoretical bounds are overly pessimistic.

The distinction between logical and physical capacity in deduplicated systems introduces additional complexity to the data migration problem. For optimal read and restore performance, the physical copies of a file's chunks must reside on the same storage volume. Thus, when migrating a file from one volume to another, this file's chunks that also belong to another file must be copied (duplicated), rather than moved. As a result, migrating data from a full volume to an empty one is likely to increase the total physical capacity of the system. Migrating data between two non-empty volumes can either increase or decrease the total physical capacity, depending on the degree of duplication between the migrated data and the data on the target volume. Intuitively, to optimize the system's overall storage utilization, a migration plan should minimize the amount of data that is duplicated as a result.

Deduplication complicates other related tasks in a similar manner. Garbage collection must consider the logical as well as the physical relationships between chunks, files, and containers. Unfortunately, specific approaches for optimizing garbage collection are not directly applicable to data migration [23, 28, 40]. As another example, online assignment of streams to servers in distributed systems must consider both content similarity and load balancing. Current solutions distribute data to servers in the granularity of individual chunks [24], super-chunks [21], files [15], or users [22], considering server load as a secondary objective. These online solutions are based on partial knowledge of the data in the system, and may result in suboptimal plans if applied directly to data migration.

2.3 Existing data migration approaches

A recent paper describes the Data Domain Cloud Tier, in which customers maintain two tightly connected deduplication domains, in an on-premises system and in a remote object store [25]. They dedicate special attention to the process of *seeding* the cloud-tier—migrating a portion of the on-premises system into an initially empty object store. While the choice of the exact files to migrate is deferred to the client, the general use-case is to keep older backups in the cloud-tier and newer ones on-premises. The authors refer to "many days or weeks possibly required to transfer a large dataset to the cloud", strongly motivating our goal to minimize the amount of data replicated during migration.

Rangoli is a greedy algorithm for space reclamation in a deduplicated system [49]. Although it predates [25] by several years, its problem formulation is equivalent: choose a set of files for migration from an existing volume to a new empty volume. Rangoli constructs a migration plan by greedily grouping files into roughly equal-sized bins according to the blocks they share, and then chooses for migration the bin whose files have the least amount of data shared with other bins. The migration objective is specified as the number of bins.

In another recent paper, Harnik et al. address migration in the broader context of load balancing [32]. Their system consists of several non-empty volumes, each operating as an independent deduplication domain. The goal is to estimate the amount of deduplication between files on different volumes, to determine the potential occupancy reduction achieved by migrating files between volumes.[1] The focus of the study is a sketching technique that facilitates this estimation. In their evaluation, the authors propose a greedy algorithm that iteratively migrates files from one volume to another, with the goal of minimizing the overall physical occupancy in the system.

Capacity planning and space reclamation in deduplicated systems are relatively new challenges. Current solutions are either naïve—migrating backups according to their age—or greedy. At the same time, migration carries significant costs in terms of physical capacity and bandwidth consumption, and it is unclear whether and how much the greedy solutions can be improved upon. This gap is the main motivation of our study.

[1] In the original paper, migration is described in terms of moving logical volumes between physical servers. Thus, their volumes are equivalent to what we refer to as files, for simplicity.

2.4 Integer linear programming (ILP)

Integer linear programming (ILP) is a well-known optimization problem. The input to ILP is a set Ax of linear constraints, each of the form $a_0x_0 + a_1x_1 \cdots + a_{n-1}x_{n-1} \leq c$, where $a_1, \ldots, a_n, c \in \mathbb{Z}$, and an objective function of the form $Tx = t_0x_0 + t_1x_1 + \cdots + t_{n-1}x_{n-1}$. The problem is finding, given Ax and Tx, an integer assignment to $x_0, x_1, \ldots, x_n$ that satisfies A_x and maximizes Tx. There is no known efficient algorithm for solving ILP. In particular, when the variables are restricted to Boolean assignments (0 or 1), then merely deciding whether Ax has a solution has been long known to be NP-Complete [37].

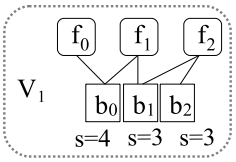
Nevertheless, ILP is used in various fields for modeling a wide range of problems [8, 51, 66, 69]. This wide use has been made possible by efficient *ILP solvers*—designated heuristic-based tools that can handle and solve very large instances. Thus, despite its theoretical hardness, ILP can in many cases be solved in practice for instances that contain hundreds of thousands and even millions of variables and constraints.

Most ILP solvers are based on the Simplex algorithm [19], which efficiently solves linear programming where the variables are not necessarily integers. They then search for an optimal integer solution, starting the search at the vicinity of the non-integer one. The wide variety of ILP solvers includes open-source solvers such as SYMPHONY [6], lp_solve [4], and GNU LP Kit [3]. Industrial tools include IBM CPLEX [1] and the Gurobi optimizer [2]. In this research, we take advantage of these highly-optimized solvers for finding the optimal migration plan in a deduplicated storage system.

3 GoSeed ILP optimization

We formulate the goal of generating a migration plan as follows. *Move physical data of size M from one volume to another, while minimizing R, the total size of the physical data that must be copied (replicated) as a result.* In a *seeding plan*, the target volume is initially empty. We refer to R as the *cost* of the migration. Note that in a seeding plan, minimizing R minimizes the total capacity of the system, as well as the amount of data *transferred* between volumes during the migration.

Problem definition. For a storage volume V, let $B_V = \{b_0, b_1, \ldots, b_{m-1}\}$ be the set of unique blocks stored on V, and let $s(b)$ be the size of block b. The storage cost of the volume is the total size of the blocks stored on it, i.e., $s(V) = \Sigma_{b_i \in B_V} s(b_i)$. Let $F_V = \{f_0, f_1, \ldots, f_{n-1}\}$ be the set of files mapped to V, and let $I_V \subseteq B_V \times F_V$ be an inclusion relation, where $(b, f) \in I_V$ means that block b is included in file f. We intentionally disregard the order of blocks in a file, or blocks that appear several times in one file. While this information is required for restoring the original file, it is irrelevant for the allocation of blocks to volumes.



(1) $0 \le x_0, x_1, x_2, m_0, m_1, m_2, r_0, r_1, r_2 \le 1$

(2) $m_0 \le x_0,\ m_0 \le x_1,\ m_1 \le x_1,\ m_1 \le x_2,\ m_2 \le x_2$

(3) $x_0 \le m_0 + r_0,\ x_1 \le m_0 + r_0,\ x_1 \le m_1 + r_1,$
$x_2 \le m_1 + r_1,\ x_2 \le m_2 + r_2$

(4) $4 \cdot m_0 + 3 \cdot m_1 + 3 \cdot m_2 = 3$

Goal: minimize $4 \cdot r_0 + 3 \cdot r_1 + 3 \cdot r_2$

Figure 1: Example system and its formulation as an ILP problem, where the goal is to migrate 30% of the physical space ($M = 3$).

We require that all the blocks included in a file are stored on the volume this file is mapped to. Thus, if a file f is *remapped* from V_1 to V_2, then every block that is included in f must be either *migrated* to V_2 or *replicated*. Similarly, if we migrate a block b from volume V_1 to volume V_2, then every file f such that $(b, f) \in I_{V_1}$ must be remapped from V_1 to V_2.

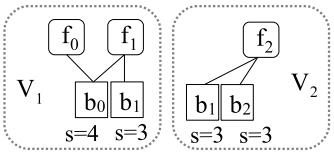
The ***seeding problem*** is to decide, given a source volume V_1 with $B_{V_1}, F_{V_1}, I_{V_1}$, an empty destination volume V_2, a target size M and a threshold size R, whether there exists a set $B' \subseteq B_{V_1}$ of blocks whose total size is M, that can be migrated from V_1 to V_2, such that the total size of blocks that are replicated is at most R. In practice, we are interested in the respective optimization problem. Namely, the *seeding optimization problem* is to find such a set B' while minimizing R. A solution to the seeding optimization problem is a migration plan: the list of files that are remapped, the list of blocks that are replicated, and B'—the list of blocks that are migrated from V_1 to V_2.

We prove that the seeding problem is NP-hard using a reduction from the Clique problem (omitted due to space considerations). Intuitively, the relationship between files and blocks influences the quality of the solution, because the decision whether to migrate a specific block depends on the decision regarding other blocks. In this aspect, seeding is similar to many other set-selection problems such as Set Cover, Vertex Cover, and Hitting Set, that are known to be NP-hard [37].

ILP formulation. We model the seeding optimization problem as an ILP problem as follows. For every file $f_i \in F_{V_1}$ we allocate a Boolean variable x_i. Assigning 1 to x_i means that f_i is remapped from V_1 to V_2. For every block $b_i \in B_{V_1}$ we allocate two Boolean variables, m_i, r_i. Assigning 1 to m_i means that b_i is migrated from V_1 to V_2, and assigning 1 to r_i means that b_i is replicated and will be stored in both V_1 and V_2.

We model the problem constraints as a set of linear inequalities, as follows.

1. All variables are Boolean: $0 \le x_j \le 1$, $0 \le m_i \le 1$, and $0 \le r_i \le 1$ for every $f_j \in F_{V_1}$ and $b_i \in B_{V_1}$.
2. If a block b is migrated, then every file that b is included in is remapped: $m_i \le x_j$ for every i, j such that $(b_i, f_j) \in I_{V_1}$.



- Block b_2 is migrated: $m_2 = 1$
- File f_2 is remapped: $x_2 = 1$
- Block b_1 is replicated: $r_1 = 1$
- The remaining files and blocks are untouched: $x_0 = x_1 = m_0 = m_1 = r_0 = r_2 = 0$
- The total cost is $R = 3 \cdot r_1 = 3$

Figure 2: The system from Figure 1 after applying the optimal migration plan with $M = 3$.

3. If a file f is rempapped, then every block that is included in f is either migrated or replicated: $x_j \le m_i + r_i$ for every i, j such that $(b_i, f_j) \in I_{V_1}$.
4. The total size of migrated blocks is M: $\Sigma_{b_i \in B_{V_1}} s(b_i) \cdot m_i = M$.

The objective function minimizes the total size of blocks that are replicated: minimize $\Sigma_{b_i \in B_{V_1}} s(b_i) \cdot r_i$.

Another intuitive constraint is that a block cannot be migrated and replicated at the same time: $m_i + r_i \le 1$ for every $b_i \in B_{V_1}$. This constraint will be satisfied implicitly in any optimal solution—if a block is migrated ($m_i = 1$) then replicating it will only increase the value of the objective function, and thus r_i will remain 0. This is also true for all the solutions in the space defined by the Simplex algorithm, and consequently for suboptimal solutions returned when the solver times out.

A solution to the ILP instance is an assignment of values to the Boolean variables. We note, however, that such an assignment does not necessarily exist. If a solution does not exist, Simplex-based solvers will return quickly—we observed a few minutes in our evaluation. If a solution to the ILP instance exists, we find B' by returning every block b_i such that $m_i = 1$, and the list of replicated blocks by returning every block b_i such that $r_i = 1$. The list of files to remap is given by every file f_i such that $x_i = 1$.

Figure 1 shows an example of a simple deduplicated system, and the formulation as an ILP instance of the respective seeding optimization problem with $M = 3$. The optimal solution, depicted in Figure 2, is to migrate b_2, replicate b_1, and remap f_2, which yields $R = 3$. Another feasible solution is to migrate b_1, whose size is also 3. However, migrating b_1 results in replicating both b_0 and b_2, which yields $R = 7$.

Refinements. The requirement to migrate blocks whose total size is exactly M may severely limit the possibility of finding a solution. Fortunately, in real settings, there is some range of acceptable migrated capacities. For example, for the file system in Figure 1, a solution exists for $M = 3$ but not for $M = 2$. In realistic systems, feasible solutions may be easier to find but their cost, R, might be unnecessarily high. Thus, we redefine

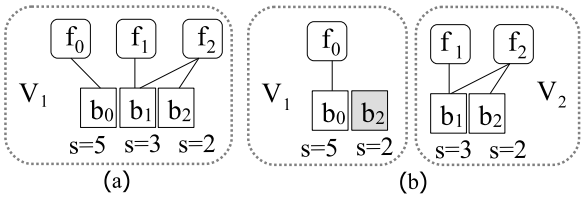


Figure 3: A migration plan with an orphan block. The goal is to migrate 30% ($M = 3$) of the system in (a). b_2 is the orphan—it was duplicated when f_2 was remapped (b).

our problem by adding a *slack value*, ε, as follows.

For a given $B_{V_1}, F_{V_1}, I_{V_1}$, target size M, and slack value ε, the *seeding optimization problem with slack* is to find $B' \subseteq B_{V_1}$ of blocks whose total size is M', $M - \varepsilon \leq M' \leq M + \varepsilon$, that can be migrated from V_1 to V_2. In the formulation as an ILP problem, we require that the total size of migrated blocks is $M \pm \varepsilon$: $M - \varepsilon \leq \Sigma_{b_i \in B_{V_1}} s(b_i) \cdot m_i \leq M + \varepsilon$. For example, for the system in Figure 1, the optimal solution for $M = 2$ and $\varepsilon = 1$, is the solution given above for $M = 3$.

Another refinement in the problem formulation is required to prevent "leftovers" on the source volume V_1. An *orphan* block is copied because a file it is included in is remapped, but no other file that includes it remains in V_1. For example, consider the system in Figure 3(a), with a migration objective of $M = 3$. For simplicity, assume that $\varepsilon = 0$. The only feasible solution is depicted in Figure 3(b), where b_1 is migrated, f_1 and f_2 are remapped, and b_2 is replicated. b_2 cannot be migrated because this would exceed the target migration size, $M = 3$. Replicating b_2 leaves an extra copy of this block in V_1, where it is not contained in any file.

Although a migration plan with orphan blocks represents a feasible solution to the ILP problem, it is an inefficient one. For example, b_2 in Figure 3(b) consists of 20% of the system's original capacity. Orphans can be eliminated by garbage collection, or even as part of the migration process [25]. This is essentially equivalent to migrating the orphan blocks, rather than replicating them, resulting in a migrated capacity which exceeds the original objective. For example, removing b_2 from volume V_2 in Figure 3(b) is equivalent to a migration plan with $M = 5$, rather than the intended $M = 3$.

We eliminate such solutions by adding the following constraint: if a block b is copied, then at least one file it is included in is not remapped: $r_i \leq \Sigma_{\{j|(b_i,f_j)\in I_{V_1}\}}(1 - x_j)$ for every $b_i \in B_{V_1}$. This additional constraint may result in the solver returning without a solution. Such cases should be addressed by increasing ε or modifying M. Nevertheless, the decision whether to prevent orphan blocks in the migration plan or to eliminate them during its execution is a design choice that can easily be realized by adding or removing the above constraint.

Complexity. The number of constraints in the ILP formulation is linear in the size of I_V—the number of pointers from files to blocks in the system. Although the size of I_V can be at most $|B_V| \cdot |F_V|$, it is likely considerably smaller in practice: the majority of the files are small, and the majority of the blocks are included in a small number of files [47].

In general, the time required for an ILP solver to find an optimal solution depends on many factors, including the number of variables, the connections between them (represented by the constraints), and the number of feasible solutions. In our context, the size of the problem is determined by the number of files and blocks, and its complexity depends on the deduplication ratio and on the pattern of data sharing between the files. It is difficult to predict how each of these factors will affect the solving time in practice. Furthermore, small changes in the target migration size and in the slack value may significantly affect the solver's performance. We evaluate the sensitivity of GoSeed to these parameters in Section 5.

4 GoSeed Acceleration Methods

The challenge in applying ILP solvers to realistic migration problems is their size. In a system with an average chunk size of 8KB, there will be approximately 130M chunks in each TB of physical capacity. Thus, the runtime for generating a migration plan for a source volume with several TBs of data would be unacceptably long. In this section, we present three methods for reducing this generation time. We describe their advantages and limitations and the ways in which they may be combined, and evaluate their effectiveness in Section 5.

4.1 Solver timeout

The runtime of an ILP solver can be limited by specifying a timeout value. When a timeout is reached before the optimal solution is found, the solver will halt and return the best feasible solution found thus far. This approach has the advantage of letting the solver process the unmodified problem. It does not require any preprocessing, and, theoretically, the solver may succeed in finding the optimal solution. The downside is that when the solver is timed out, we cannot necessarily tell how far the suboptimal solution is from the optimal one.

4.2 Fingerprint sampling

Sampling is a standard technique for handling large problems, and has been used in deduplication systems to increase the efficiency of the deduplication process [15, 16, 41], to route streams to servers [21], for estimating deduplication ratios [33], and for managing volume capacities [32]. We use sampling in the same way it is used in [32]. Given a *sampling degree* k, we include in our sample all the chunks whose fingerprint contains k leading zeroes, and all the files containing those chunks. When the fingerprint values are uniformly distributed, the sample will include $\frac{1}{2^k}$ chunks. Harnik et al. show in [32] that $k = 13$ guarantees small enough errors for estimating the capacity of deduplicated volumes larger than 100GB.

Sampling reduces the size of the ILP instance by a predictable factor: incrementing the sampling degree k by one reduces the number of blocks by half. Combining sampling and timeouts presents an interesting tradeoff: a smaller sampling factor results in a larger ILP instance that more accurately represents the sampled system. However, solving a larger instance

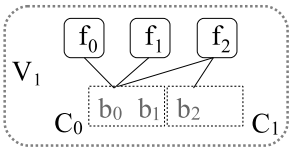


Figure 4: The system from Figure 1 with container aggregation.

is more likely to time out and return a suboptimal solution. It is not clear which combination will result in a better migration plan—a suboptimal solution on a large instance, or an optimal solution on a small instance. Our analysis in Section 5 shows how the answer depends on the original (unsampled) instance and on the length of the timeout.

4.3 Container-based aggregation

Aggregation is often employed as a first step in analysing large datasets. In deduplication systems, containers are a natural basis for aggregation. Containers are often compressed before being written to durable storage, and are decompressed when they are fetched into memory for retrieving individual chunks. Thus, generating and executing a migration plan in the granularity of containers holds the advantage of avoiding decompression as well as an increase in the fragmentation in the system by migrating individual chunks from containers.

To formulate the migration problem with containers we coalesce chunks that are stored in the same container into a single block, and remove parallel edges, i.e., pointers from the same file to different chunks in the same container. Figure 4 shows the container view of the volume from Figure 1. In a real system, formulating the migration problem with containers is more efficient than with chunks: when processing file recipes, we can ignore the chunk fingerprints and use only the container IDs for generating the variables and constraints.

In a system that stores chunks in containers, the container-based migration problem accurately represents the system's original constraints. At the same time, we can further leverage container-based aggregation as an acceleration method by artificially increasing the container size beyond the size used by the system. With *aggregation degree* K, we coalesce every K adjacent containers into one, like we do for chunks. Thus, a system with 4MB containers can be represented as one with $4K$-MB containers by coalescing every K original containers. Containers typically store hundreds of chunks, which means that the size of the resulting ILP problem will be smaller by several orders of magnitude. Furthermore, containers are allocated as fixed-size extents, which further reduces the ILP problem complexity: the optimization goal of minimizing the *total size* of migrated blocks becomes a simpler goal of minimizing their *number*.

A container-based seeding plan can be obtained more quickly than a chunk-based one. Thus, if aggregation is combined with solver timeouts, a container-based suboptimal solution will likely be closer to the optimal (container-based) solution than in an execution solving the chunk-based instance. At the same time, container-based aggregation (like any aggregation method) reduces the granularity of the solution, which affects its efficiency as an acceleration method for the original chunk-based problem. Namely, an optimal container-based migration plan is not necessarily optimal if the migration is executed in the granularity of chunks.

Consider a migration plan generated with containers, and let F_{V_2} be the set of files that are remapped to V_2 as a result of that plan. F_{V_1} is the set of files that remain on V_1. If a container is not part of the migration plan, this means that *all* of its chunks are contained only in files from F_{V_1}. When a container is marked for migration, this means that *all* of its chunks are contained only in files from F_{V_2}. When a container includes *at least one chunk* that is contained in a file from F_{V_1} as well as in a file from F_{V_2}, the entire container is marked for replication. However, this container may also contain some *"false positives"*—chunks that are contained only in files from F_{V_1} (and should not be part of the migration), or only in files from F_{V_2} (and should be migrated rather than replicated).

These false positives increase the cost of the container-based solution, and can be eliminated by performing the actual migration in the granularity of chunks, as done in [25]. However, this would eliminate the advantages of migrating entire containers, and may cause the solver to "miss" the migration plan that would have been optimal for the chunk-based ILP instance. We observe this effect in Section 5

5 Evaluation

The goal of our experimental evaluation is to answer the following questions:

- What is the difference, in terms of cost, between the ILP-based migration plan and the greedy ones?
- How do the ILP instance parameters (its size, M, and ε) affect it's complexity, indicated by the solver's runtime?
- How does timing out the solver affect the quality (cost) of the returned solution?
- How do the sampling and aggregation degrees affect the solver's runtime and the cost of the migration plan?

5.1 Experimental setup

Deduplication snapshots. We use static file system snapshots from two publicly available repositories. The UBC dataset [47] includes file systems of 857 Microsoft employees available via SNIA IOTTA [5]. The FSL dataset [7] includes daily snapshots of a Mac OS X Snow Leopard server and of student home directories at the File System and Storage Lab (FSL) at Stony Brook University [57, 58]. The snapshots include, for each file, the fingerprints calculated for each of its chunks, as well as the chunk size in bytes. Each snapshot file represents one entire file system, which is the migration unit in our model, and is represented as one file in our ILP instance.

To obtain a mapping between files and unique chunks, we emulate the ingestion of each snapshot into a simplified deduplication system. We assume that all duplicates are detected and eliminated. We emulate the assignment of chunks to containers

Volume	Files	Chunks	Dedupe	Containers	Logical
UBC-50	50	27M	0.59	122K	807 GB
UBC-100	100	73M	0.34	317K	3.5 TB
UBC-200	200	138M	0.32	570K	6.7 TB
UBC-500	500	382M	0.31	1.6M	19.5 TB
Homes	81	19M	0.13	295K	8.9 TB
MacOS-Week	102	6M	0.02	93K	20 TB
MacOS-Daily	200	6.3M	0.01	99K	43 TB

Table 1: Volume snapshots in our evaluation. The container size is 4MB. Dedupe is the deduplication ratio—the ratio between the physical and logical size of each volume. Logical is the logical size.

by assuming that unique chunks are added to containers in the order of their appearance in the original snapshot file. We create snapshots of entire volumes by ingesting several file-system snapshots one after the other, thus eliminating duplicates across individual snapshots. The resulting volume snapshot represents an independent deduplication domain.

The volume snapshots used in our experiments are detailed in Table 1. The *UBC-X* volumes contain the first X file systems in the UBC dataset. These snapshots were created with variable-sized chunks with Rabin fingerprints, whose specified average chunk size is 64KB. In practice, however, many chunks are 4KB or less. The FSL snapshots were also generated with Rabin fingerprints and average chunk size of 64KB.[2] The *MacOS-Daily* volume contains all available daily snapshots of the server between May 14, 2015 and May 8, 2016, while the *MacOS-Week* volume contains weekly snapshots, which we emulate by ingesting the snapshots from all the Fridays in repository. The Homes volume contains weekly snapshots of nine users between August 28 and October 23, 2014 (nine weeks in total).

***GoSeed* Implementation.** We use the commercial Gurobi optimizer [2] as our ILP solver, and use its `C++` interface to define our problem instances. The problem variables (x_i, m_i, r_i) are declared as Binary and represented by the `GRBVar` data type. The constraints and objective are declared as linear expressions. M and ε are given in units of percents of the physical capacity. We specify three parameters for each execution: a *timeout* value, the *parallelism* degree (number of threads), and a *random seed*. These parameters do not affect the optimality of the solution, but they do affect the solver's runtime. Specifically, the starting point for the search for an integer solution is chosen at random, which may lead some executions to complete earlier than others. If the solver times out, different executions might return solutions with slightly different costs. In our evaluation, we solve each ILP instance in three separate executions, each with a different random seed, and present the average of their execution times and costs. Our wrapper program for converting a volume snapshot into an ILP instance in Gurobi consists of approximately 400 lines of code.[3]

We ran our experiments on a server running Ubuntu 18.04.3, equipped with 64GB DDR4 RAM (with 2666 MHz bus speed), Intel® Xeon® Silver 4114 CPU (with hyper-threading functionality) running at 2.20GHz, one Dell® T1WH8 240GB TLC SATA SSD, and one Micron 5200 Series 960GB 3D TLC NAND Flash SSD. We let Gurobi use 38 CPUs, and specify a timeout of six hours, to allow for experiments with a wide range of setup and problem parameters.

Comparison to existing approaches. We use our volume snapshots to evaluate the quality of the migration plans generated by the existing approaches described in Section 2.3. We implement ***Rangoli*** according to the original paper [49]. We convert our migration objective M into a number of bins B, such that $B = \frac{1}{M}$. We modified Rangoli to comply with the restriction that the migrated capacity is between $M-\varepsilon$ and $M+\varepsilon$: when choosing one of B bins for migration, our version of Rangoli chooses only from those bins whose capacity is within the specified bounds.

For evaluation purposes, we implemented a seeding version of the greedy load balancer that was used for evaluating the capacity sketches in [32]. We refer to this algorithm as ***SGreedy***. In each iteration, SGreedy chooses one file from V_1 to remap to V_2. The remapped file is the one which yields the best space-saving ratio, i.e., the ratio between the space freed from V_1 and that added to V_2. The iterations continue until the migrated capacity is at least $M-\varepsilon$, and if, at this point, it does not exceed $M+\varepsilon$, a solution is returned. SGreedy returns a seeding plan in the form of a list of files that are remapped from V_1 to V_2. We then use a dedicated "cost calculator" to derive the cost of the migration plan on the original (unsampled) system.

Our calculator creates an array of the volume's chunks and their sizes, and two bit indicators, V_1 and V_2, that are initialized to FALSE for each chunk. It then traverses the files in the volume snapshot and updates the indicators of their blocks as follows. If a file is remapped, then the V_2 indicators of all its chunks are set to TRUE. If a file is not remapped, then the V_1 indicators of all its chunks are set to TRUE. A final pass over the chunk array calculates the replication cost by summing the sizes of all the chunks whose V_1 and V_2 indicators are both TRUE. The migrated capacity is the sum of the sizes of all the chunks whose V_2 indicator is TRUE and V_1 indicator is FALSE.

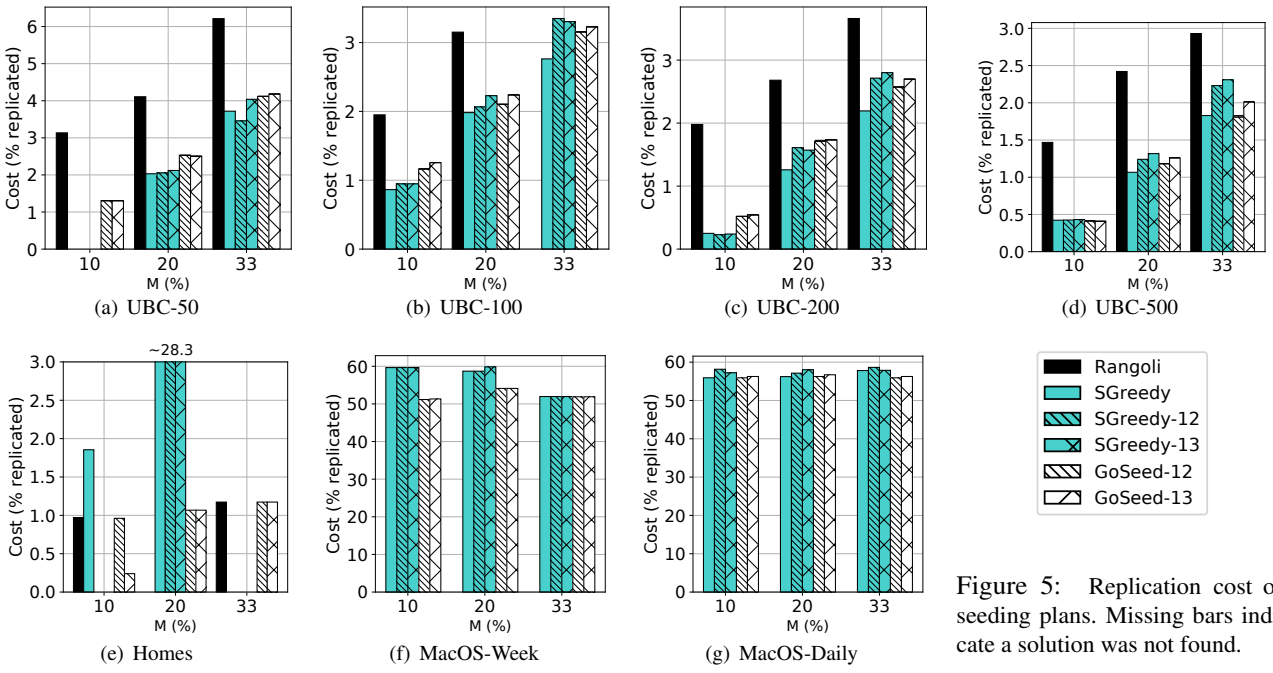
5.2 Results

Comparison of different algorithms. We first analyze the migration cost incurred by the different algorithms on the various volume snapshots. Figure 5 shows our results with three values of M (10,20,33) and $\varepsilon = 2$. A missing bar of an algorithm indicates that it did not find a solution for that instance. GoSeed-K and SGreedy-K depict the results obtained by these algorithms running on a snapshot created with sampling degree K (the cost was calculated on the original snapshot).

Rangoli does not perform well on most of the volume snapshots. It incurs the highest replication cost on the UBC snapshots, except UBC-100 with $M = 33$, for which it does not find a solution. On the FSL snapshots, it finds a good solution only

[2] Due to technical issues in our preprocessing step, we had to represent all the chunks in the FSL snapshots as chunks of *exactly* 64KB. The chunk fingerprints and the deduplication between them is unchanged.

[3] Code available at https://github.com/avivnachman1/GoSeed



(a) UBC-50 (b) UBC-100 (c) UBC-200 (d) UBC-500

(e) Homes (f) MacOS-Week (g) MacOS-Daily

Figure 5: Replication cost of seeding plans. Missing bars indicate a solution was not found.

for the Homes volume (with $M = 10$ and $M = 33$), but not for the remaining instances. The backups on the MacOS volumes share most of their data, with a very low deduplication ratio. In these circumstances, Rangoli fails because it is unable to partition the files into separate bins of the required size.

SGreedy returns a solution in all but two instances (UBC-50 with $M = 10$ and Homes with $M = 33$). For the UBC snapshots, the cost of its solution is 37%-87% lower than the cost of Rangoli's solution. When SGreedy is applied to a sampled snapshot, as it was originally intended, this cost increases by as much as 28% and 27%, for sample degrees 12 and 13, respectively. This increase is expected, as the sampled snapshot "hides" some of the data sharing in the real system. However, the increase is smaller in most instances. It is also interesting to note a few cases where SGreedy returns a better solution (with a lower replication cost) on the sampled snapshot than on the original one, such as for UBC-50 with $M = 33$. These situations can happen when "hiding" some of the sharing helps the greedy process find a solution that it wouldn't find otherwise.

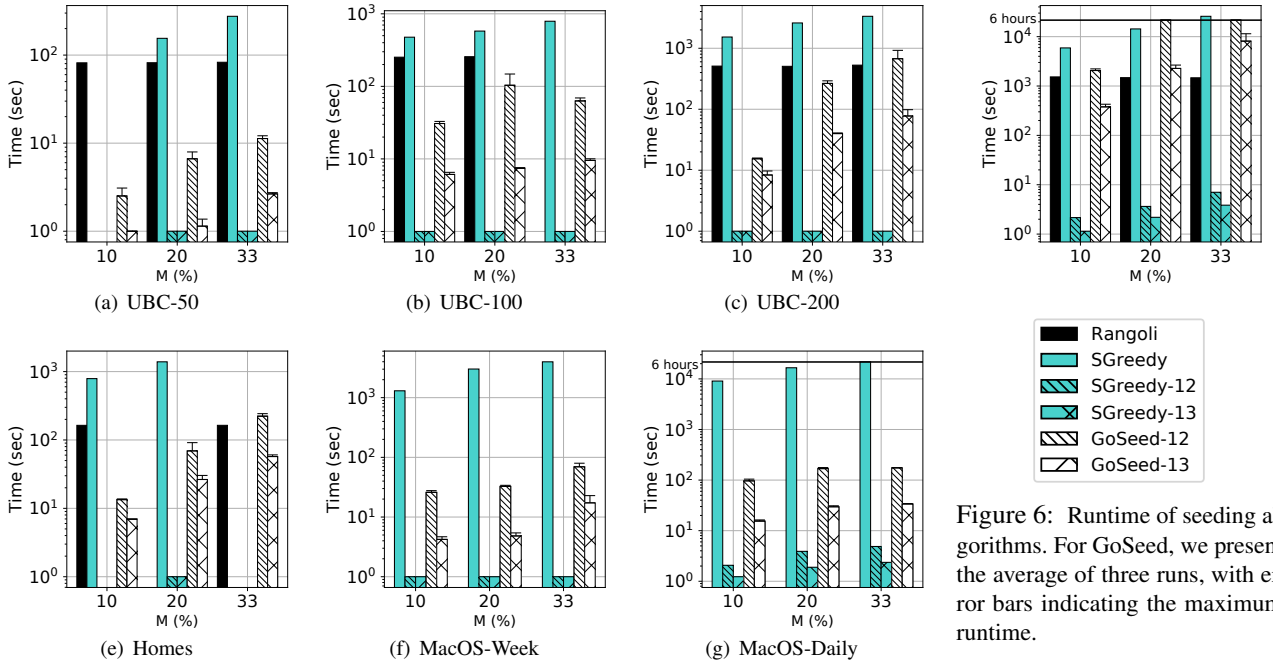
We can now classify our volumes into three rough categories. We refer to the UBC volumes as *easy*—their data sharing is modest and the greedy algorithms find good solutions for them. We refer to the Homes volume as *hard*—its data sharing is substantial and the greedy algorithms mostly return solutions with high costs (up to 29%), or don't find a solution at all. We consider the MacOS volumes to be *very hard* because of their exceptionally high degree of sharing between files. This sharing prevents Rangoli from finding any solution, and incurs very high costs (up to 60%) in the plan generated by SGreedy.

GoSeed cannot find a solution for the full snapshots, which translate to ILP instances with hundreds of millions of constraints. We thus use fingerprint sampling to apply GoSeed to the volume snapshots, with two sampling degrees, 12 and 13. Our results show that GoSeed finds a solution for all the volumes and all values of M. It generates slightly better plans with a smaller sampling degree, when more of the system's constraints are manifested in the ILP instance.

In the easy (UBC) volumes, the cost of GoSeed's migration plan is similar to that of SGreedy's plan on the sampled snapshots. It is higher for four instances (UBC-50 with $M = 20, 33$, and for UBC-100 and UBC-200 with $M = 10$) and equal or lower for the rest. This shows that greedy solutions may suffice for volumes with modest data sharing between files.

The picture is different for the hard volumes. For Homes, GoSeed consistently finds a better migration plan, while each of the greedy algorithms finds a solution for some values of M but fails to find one for others. The biggest gap between the greedy and optimal solutions occurs for $M = 20$: SGreedy (with and without sampling) replicates 27%-28% of the volume's capacity, while the replication cost of the plan generated by GoSeed is only 1%. This demonstrates a known property of greedy algorithms—their solutions are good enough most of the time, but very bad in the worst case.

Finally, for the very hard (MacOS) volumes, GoSeed finds similar or better solutions than SGreedy, with or without sampling. Although more than 50% of the volume is replicated in all of the migration plans, the replication cost of GoSeed for MacOS-Weekly with $M = 10$ and $M = 20$ is 14% and 8% lower than that of SGreedy, respectively. The exceptionally high degree of sharing in this volume indicates that better solutions likely do not exist. This conclusion was supported in our attempt to apply the "user's" migration plan from [25], remapping the oldest backups (files, in our case) to a new tier. In MacOS-Weekly and MacOS-Daily, remapping the single



(a) UBC-50 (b) UBC-100 (c) UBC-200

(e) Homes (f) MacOS-Week (g) MacOS-Daily

Figure 6: Runtime of seeding algorithms. For GoSeed, we present the average of three runs, with error bars indicating the maximum runtime.

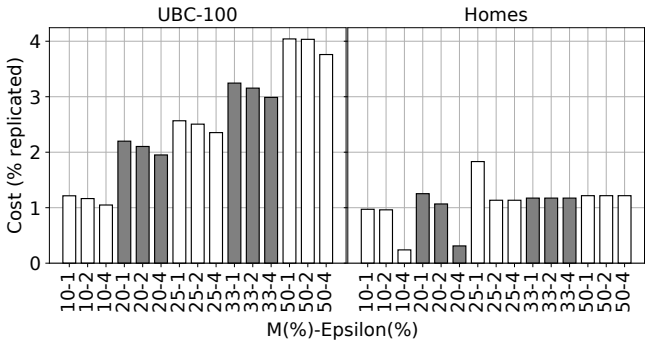


Figure 7: GoSeed plans generated with sampling degree K=12.

oldest backup to a new volume resulted in migrating 0.2% and 0.3% of the volume's capacity, and replicating 49% and 55% of it, respectively.

Figure 6 shows the runtime of the different algorithms. The runtime of GoSeed is longer than that of SGreedy on the sampled snapshot, but shorter than that of SGreedy and Rangoli on the original snapshots. GoSeed timed out at six hours only in one execution (UBC-500 and $K = 12$). The rest of the instances were solved by GoSeed in less than one hour (UBC) or five minutes (FSL). We note, though, that GoSeed utilizes 38 threads, while the greedy algorithms use only one. For a migration plan transferring several TBs of data across a wide area network or a busy interconnect, these runtimes and resources are acceptable.

Effect of ILP parameters. We first investigate how M and ε affect the solver's ability to find a good solution. We compare the cost of the plan generated by GoSeed with five values of M (used in [49]) and three values of ε on an easy (UBC-100) volume and on a hard one (Homes). The results in Figure 7 show that in the easy volume, higher values of M result in a higher cost, and that this cost can be somewhat reduced by increasing ε, which increases the number of feasible solutions. We observe a similar effect in Homes, but to a much smaller extent. We note that this effect is also shared by the greedy algorithms (not shown for lack of space), for which differences in ε often make a difference between finding a feasible solution or returning without one. Increasing M also exponentially increases the runtime of the solver—migrating more blocks results in more feasible solutions in the search space. We omit the runtimes of this experiment, but the effect can be observed in Figure 6.

We next investigate how the size of the snapshot affects the time required to solve the ILP instance. We compare problems with similar constraints and different sizes by generating sampled snapshots with K between 7 and 13 of the above two volumes. Figure 8 shows the average runtime of GoSeed on these snapshots with $M = 20$ and $\varepsilon = 2$. Error bars mark the minimum and maximum runtimes. Note that both axes are log-scaled—incrementing K by one doubles the number of blocks in the ILP instance. As we expected, the time increases exponentially with the number of blocks. The figure also shows that the runtime of the same instance with one random seed can be as much as 1.45× longer than with another seed. We discuss the implications of this difference below.

Effect of solver timeout. To evaluate the effect of timeouts on the cost of the generated plan, we generate a volume snapshot by sampling UBC-100 with $K = 8$, for which the solver's execution time is approximately four hours. We repeatedly solve this instance (with the same random seed) with increasing timeout values. We set the timeouts to fixed portions of the full runtime, after having measured the complete execution. We repeat this process for three different seeds. To eliminate the

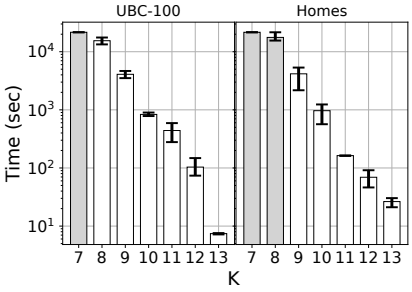


Figure 8: Solving time increases exponentially with instance size (gray bars indicate that the solver timed out).

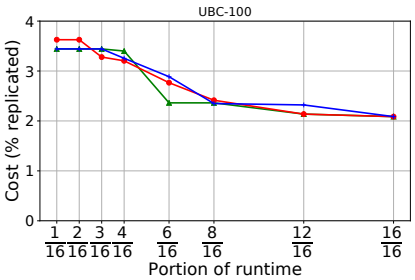


Figure 9: Migration cost decreases when timeout increases (costs are shown for three random seeds).

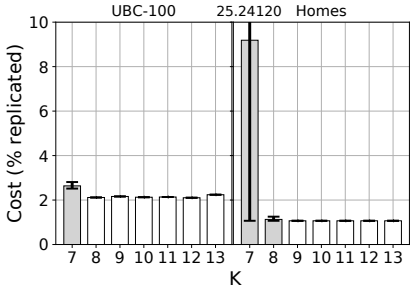


Figure 10: Cost is hardly affected by the sampling degree, unless the instance becomes too large.

effect of sampling, we present the cost of migration assuming the sample represents the entire system.

The results in Figure 9 show that the most substantial cost reduction occurs in the first half of the execution, after which the quality of the solution does not improve considerably. The three processes converge to the same optimal solution at different speeds, corresponding to the different runtimes in Figure 8. At the same time, we note that the largest differences in cost occur between suboptimal solutions returned in the first half of the execution, when the solver makes most of its progress. The cost difference is relatively small and does not exceed 22% (at $\frac{6}{16}$)—a much smaller difference than the difference in time required to find the optimal solution.

Gurobi provides an interface for querying the solver for intermediate results without halting its execution. We did not use this interface because it might compromise the accuracy of our time measurements. However, it can be used to periodically check the rate at which the intermediate solution improves. When the rate decreases and begins to converge, continuing the execution yields diminishing returns, and it can be halted.

Effect of fingerprint sampling. We evaluate the effect of the sampling degree on the cost of the solution by calculating the costs of the plans generated for UBC-100 and Homes with $M = 20$, $\varepsilon = 2$, and K between 7 and 13. Figure 10 shows that the difference between the cost of optimal solutions is very small. However, when the solver times out, the cost of the suboptimal solution can be as much as 24× higher.

Our results for varying the ILP instance parameters and sampling degrees suggest the following straightforward heuristic for obtaining the best seeding plan within a predetermined time frame. Generate a sample of the system with degree between 10 and 13—smaller degrees are better for smaller systems. If the solver times out, increase the sampling degree by one. If the solver completes and there is still time, solve instances with increasing values of ε until the end of the time frame is reached. This process results in a set of solutions that form a Pareto frontier—their cost decreases as their migrated capacity is farther from the original objective M. The final solution should be chosen according to the design objectives of the system.

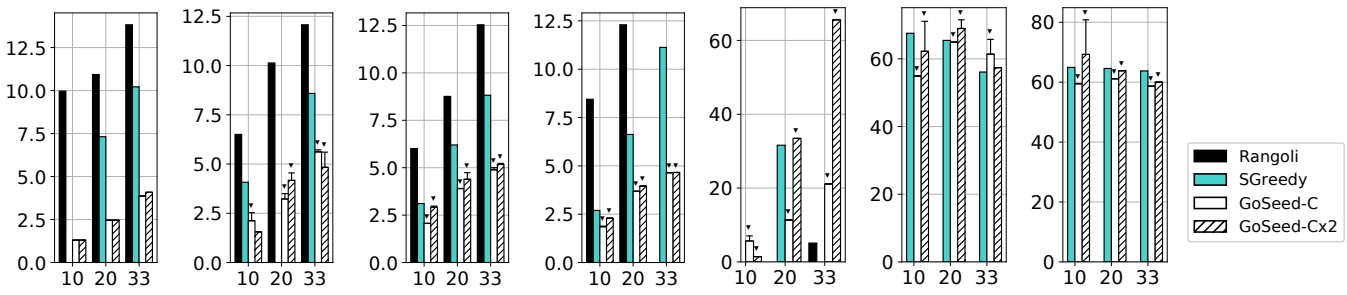
Efficiency of container-based plans. The container-based aggregation generates a reduced ILP instance which is an accurate representation of the connections between files and containers. This representation can also be used to generate container-based migration plans with Rangoli and SGreedy. Thus, our next experiment compares the costs of GoSeed and the greedy algorithms on the same instances. Our results in Figure 11 show that in these circumstances, GoSeed can reduce the migration cost obtained by Rangoli and SGreedy by as much as 87% and 66%, respectively. These results are not surprising given the size of the ILP instances—they consist of several hundred thousand variables, well within Gurobi's capabilities. As a result, even in experiments in which Gurobi times out (indicated by the small triangles in the figure), its suboptimal solutions are considerably better than the greedy ones. The costs with aggregated containers (GoSeed-C×2) are higher because of the false dependencies described in Section 4.3.

We used our cost calculator to compare the chunk-level cost of the container-based migration plan to the greedy plans generated for the original system (figures omitted due to space considerations). For the MacOS volumes and for UBC-50, GoSeed's container-based plan outperforms Rangoli and is comparable to SGreedy. However, for the larger UBC volumes and for Homes, SGreedy and Rangoli find solutions with as much as 7.6× and 13.6× lower cost, respectively. On these instances, Gurobi returned a suboptimal solution which was close to the container-based optimum, but far from the chunk-based optimum, due to the reasons described in Section 4.3. We therefore recommend using GoSeed with container-based aggregation if the migration is to be performed with entire containers, and with fingerprint sampling otherwise.

6 Discussion

Data migration within a large-scale deduplicated system can reallocate tens of terabytes of data. This data is possibly transferred over a wide area network or a busy interconnect, and some of it might be replicated as a result. The premise of our research is that the potentially high costs of data migration justify solving a complex optimization problem with the goal of minimizing these costs.

Thus, in contrast to existing greedy *heuristics* to this hard problem, GoSeed attempts to *solve* it. By formulating data migration as an ILP instance, GoSeed can "hide" its complexity



(a) UBC-50 (b) UBC-100 (c) UBC-200 (d) UBC-500 (e) Homes (f) Mac-Week (g) Mac-Daily

Figure 11: Replication cost of container-based migration plans. For GoSeed, we present the average of three runs (error bars indicate the maximum cost). Triangles indicate experiments in which the solver timed out. GoSeed outperforms the greedy solutions by as much as 87%.

by leveraging off-the-shelf highly optimized solvers. This approach is independent of specific design and implementation details of the deduplication system or the ILP solver. However, it introduces an inherent tradeoff between the time spent generating a seeding plan, and the cost of executing it. As this cost depends on the system's characteristics, such as network speed, cost of storage, and read and restore workload, the potential for cost saving by GoSeed is system dependent as well.

Our evaluation showed that the benefit of GoSeed is high in two scenarios. The first is when the problem's size allows the solver to find the optimal (or near optimal) solution within the allocated time. Container-based migration is an example of this case, where GoSeed significantly reduced the migration cost of the greedy algorithms. The second case is when a high degree of data sharing in the system makes it hard for the greedy solutions to find a good migration plan, causing them to produce a costly solution or no solution at all. At the same time, for systems with low or exceptionally high degrees of data sharing, the greedy solutions and that of GoSeed are comparable.

Accurately identifying the large instances for which GoSeed would significantly improve on the greedy solution is not straightforward, and requires further research. Fortunately, a simple hybrid approach can provide 'the best of both worlds': one can run the greedy algorithm, followed by GoSeed, and execute the migration plan whose cost is lower.

Generalizations. Seeding is the simplest form of data migration in large systems. A natural next step to this work is to generalize our ILP-based approach to more complex migration scenarios, such as migration into a non-empty volume, and migration where both source and target volumes are chosen as part of the plan. Each generalization introduces additional aspects, and might require reformulating not only the ILP constraints, but also its objective function.

For example, when the destination volume is not empty, the optimal migration plan can be the one that minimizes the total storage capacity on the source and destination volumes combined. An alternative formulation might minimize the total amount of data that must be transferred from the source volume to the destination. In the most general case, generating the migration plan also entails determining either the source or the destination volume, or both, such that the migration goal is achieved and the objective is optimized. Data migration in general introduces additional objectives, such load balancing between volumes, or optimizing the migration process under certain network conditions and limitations. The problem can be further extended by allowing some files to be split between volumes, introducing a new tradeoff between the cost of migration and that of file access.

The ILP formulation of these problems will result in considerably more complex instances than those of the seeding problem. As a result, we might need to apply our acceleration methods more aggressively, e.g., by increasing the fingerprint sampling degree, or construct new methods. Thus, each generalization of the seeding problem introduces non-trivial challenges as well as additional tradeoffs between the solving time and the cost of migration.

7 Conclusions

We presented GoSeed, an algorithm for generating theoretically optimal seeding plans in deduplicated systems, and three acceleration methods for applying it to realistic storage volumes. Our evaluation demonstrated the effectiveness of the acceleration methods: GoSeed can produce an optimal seeding plan on a sample of the system in less than an hour, even in cases where the greedy solutions do not find a feasible solution to the problem. When executed on the original system, GoSeed's solution is not theoretically optimal, but it can substantially reduce the cost of the greedy solutions.

Finally, our formulation of data migration as an ILP problem, combined with the availability of numerous ILP solvers, opens up new opportunities for additional contributions in this domain, and for making data migration more efficient.

Acknowledgments

We thank our shepherd, Dalit Naor, and the anonymous reviewers, for their helpful comments. We thank Sharad Malik for his insightful suggestions, and Yoav Etsion for his invaluable help with the evaluation infrastructure. We thank Polina Manevich, Michal Amsterdam, Nadav Levintov, Benny Lodman, Matan Levy, Yoav Zuriel, Shai Zeevi, Eliad Ben-Yishai, Maor Michaelovitch, Itai Barkav, and Omer Hemo for their help with the implementation and with processing the traces.

References

[1] CPLEX Optimizer. https://www.ibm.com/analytics/cplex-optimizer. Accessed: 2019-12-29.

[2] The fastest mathematical programming solver. http://www.gurobi.com/. Accessed: 2019-12-29.

[3] GLPK (GNU Linear Programming Kit). https://www.gnu.org/software/glpk/. Accessed: 2019-12-29.

[4] Introduction to lp_solve 5.5.2.5. http://lpsolve.sourceforge.net/5.5/. Accessed: 2019-12-29.

[5] SNIA IOTTA Repository. http://iotta.snia.org/tracetypes/6. Accessed: 2019-12-29.

[6] SYMPHONY development home page. https://projects.coin-or.org/SYMPHONY. Accessed: 2019-12-29.

[7] Traces and snapshots public archive. http://tracer.filesystems.org/. Accessed: 2019-12-29.

[8] Jeph Abara. Applying integer linear programming to the fleet assignment problem. *Interfaces*, 19(4):20–28, 1989.

[9] Bhavish Aggarwal, Aditya Akella, Ashok Anand, Athula Balachandran, Pushkar Chitnis, Chitra Muthukrishnan, Ramachandran Ramjee, and George Varghese. EndRE: An end-system redundancy elimination service for enterprises. In *7th USENIX Conference on Networked Systems Design and Implementation (NSDI 10)*, 2010.

[10] Yamini Allu, Fred Douglis, Mahesh Kamat, Ramya Prabhakar, Philip Shilane, and Rahul Ugale. Can't we all get along? Redesigning protection storage for modern workloads. In *2018 USENIX Annual Technical Conference (USENIX ATC 18)*, 2018.

[11] Eric Anderson, Joseph Hall, Jason D. Hartline, Michael Hobbs, Anna R. Karlin, Jared Saia, Ram Swaminathan, and John Wilkes. An experimental study of data migration algorithms. In *5th International Workshop on Algorithm Engineering (WAE 01)*, 2001.

[12] Eric Anderson, Michael Hobbs, Kimberly Keeton, Susan Spence, Mustafa Uysal, and Alistair Veitch. Hippodrome: Running circles around storage administration. In *1st USENIX Conference on File and Storage Technologies (FAST 02)*, 2002.

[13] Eric Anderson, Mahesh Kallahalla, Susan Spence, Ram Swaminathan, and Qiang Wan. *Ergastulum: quickly finding near-optimal storage system designs*. HP Laboratories, June 2002.

[14] Alysson Bessani, Miguel Correia, Bruno Quaresma, Fernando André, and Paulo Sousa. DepSky: Dependable and secure storage in a cloud-of-clouds. *ACM Transactions on Storage*, 9(4):12:1–12:33, November 2013.

[15] Deepavali Bhagwat, Kave Eshghi, Darrell D. E. Long, and Mark Lillibridge. Extreme binning: Scalable, parallel deduplication for chunk-based file backup. In *IEEE International Symposium on Modeling, Analysis Simulation of Computer and Telecommunication Systems (MASCOTS 09)*, 2009.

[16] Feng Chen, Tian Luo, and Xiaodong Zhang. CAFTL: A content-aware flash translation layer enhancing the lifespan of flash memory based solid state drives. In *9th USENIX Conference on File and Stroage Technologies (FAST 11)*, 2011.

[17] Licheng Chen, Zhipeng Wei, Zehan Cui, Mingyu Chen, Haiyang Pan, and Yungang Bao. CMD: Classification-based memory deduplication through page access characteristics. In *10th ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments (VEE 14)*, 2014.

[18] Austin T. Clements, Irfan Ahmad, Murali Vilayannur, and Jinyuan Li. Decentralized deduplication in SAN cluster file systems. In *2009 Conference on USENIX Annual Technical Conference (USENIX 09)*, 2009.

[19] George B. Dantzig. *Linear programming and extensions*. Rand Corporation Research Study. Princeton Univ. Press, Princeton, NJ, 1963.

[20] Biplob Debnath, Sudipta Sengupta, and Jin Li. ChunkStash: Speeding up inline storage deduplication using flash memory. In *2010 USENIX Conference on USENIX Annual Technical Conference (USENIX ATC 10)*, 2010.

[21] Wei Dong, Fred Douglis, Kai Li, Hugo Patterson, Sazzala Reddy, and Philip Shilane. Tradeoffs in scalable data routing for deduplication clusters. In *9th USENIX Conference on File and Stroage Technologies (FAST 11)*, 2011.

[22] Fred Douglis, Deepti Bhardwaj, Hangwei Qian, and Philip Shilane. Content-aware load balancing for distributed backup. In *25th International Conference on Large Installation System Administration (LISA 11)*, 2011.

[23] Fred Douglis, Abhinav Duggal, Philip Shilane, Tony Wong, Shiqin Yan, and Fabiano Botelho. The logic of physical garbage collection in deduplicating storage. In *15th USENIX Conference on File and Storage Technologies (FAST 17)*, 2017.

[24] Cezary Dubnicki, Leszek Gryz, Lukasz Heldt, Michal Kaczmarczyk, Wojciech Kilian, Przemyslaw Strzelczak, Jerzy Szczepkowski, Cristian Ungureanu, and Michal Welnicki. HYDRAstor: A scalable secondary storage. In *7th Conference on File and Storage Technologies (FAST 09)*, 2009.

[25] Abhinav Duggal, Fani Jenkins, Philip Shilane, Ramprasad Chinthekindi, Ritesh Shah, and Mahesh Kamat. Data Domain Cloud Tier: Backup here, backup there, deduplicated everywhere! In *2019 USENIX Annual Technical Conference (USENIX ATC 19)*, 2019.

[26] EMC Corporation. *INTRODUCTION TO THE EMC XtremIO STORAGE ARRAY (Ver. 4.0)*, rev. 08 edition, April 2015.

[27] Jingxin Feng and Jiri Schindler. A deduplication study for host-side caches in virtualized data center environments. In *29th IEEE Symposium on Mass Storage Systems and Technologies (MSST 13)*, 2013.

[28] Min Fu, Dan Feng, Yu Hua, Xubin He, Zuoning Chen, Wen Xia, Fangting Huang, and Qing Liu. Accelerating restore and garbage collection in deduplication-based backup systems via exploiting historical information. In *2014 USENIX Annual Technical Conference (USENIX ATC 14)*, 2014.

[29] Fanglu Guo and Petros Efstathopoulos. Building a high-performance deduplication system. In *2011 USENIX Conference on USENIX Annual Technical Conference (USENIX ATC 11)*, 2011.

[30] Aayush Gupta, Raghav Pisolkar, Bhuvan Urgaonkar, and Anand Sivasubramaniam. Leveraging value locality in optimizing NAND flash-based SSDs. In *9th USENIX Conference on File and Stroage Technologies (FAST 11)*, 2011.

[31] Diwaker Gupta, Sangmin Lee, Michael Vrable, Stefan Savage, Alex C. Snoeren, George Varghese, Geoffrey M. Voelker, and Amin Vahdat. Difference engine: Harnessing memory redundancy in virtual machines. In *8th USENIX Conference on Operating Systems Design and Implementation (OSDI 08)*, 2008.

[32] Danny Harnik, Moshik Hershcovitch, Yosef Shatsky, Amir Epstein, and Ronen Kat. Sketching volume capacities in deduplicated storage. In *17th USENIX Conference on File and Storage Technologies (FAST 19)*, 2019.

[33] Danny Harnik, Ety Khaitzin, and Dmitry Sotnikov. Estimating unseen deduplication-from theory to practice. In *14th Usenix Conference on File and Storage Technologies (FAST 16)*, 2016.

[34] Danny Harnik, Benny Pinkas, and Alexandra Shulman-Peleg. Side channels in cloud services: Deduplication in cloud storage. *IEEE Security Privacy*, 8(6):40–47, Nov 2010.

[35] Charles B. Morrey III and Dirk Grunwald. Content-based block caching. In *23rd IEEE Symposium on Mass Storage Systems and Technologies (MSST 06)*, 2006.

[36] Michal Kaczmarczyk, Marcin Barczynski, Wojciech Kilian, and Cezary Dubnicki. Reducing impact of data fragmentation caused by in-line deduplication. In *Proceedings of the 5th Annual International Systems and Storage Conference (SYSTOR 12)*, 2012.

[37] R. Karp. Reducibility among combinatorial problems. In R. Miller and J. Thatcher, editors, *Complexity of Computer Computations*, pages 85–103. Plenum Press, 1972.

[38] Cheng Li, Philip Shilane, Fred Douglis, Hyong Shim, Stephen Smaldone, and Grant Wallace. Nitro: A capacity-optimized SSD cache for primary storage. In *2014 USENIX Annual Technical Conference (USENIX ATC 14)*, 2014.

[39] Jin Li, Xiaofeng Chen, Mingqiang Li, Jingwei Li, Patrick PC Lee, and Wenjing Lou. Secure deduplication with efficient and reliable convergent key management. *IEEE Transactions on Parallel and Distributed Systems*, 25(6):1615–1625, June 2014.

[40] Mark Lillibridge, Kave Eshghi, and Deepavali Bhagwat. Improving restore speed for backup systems that use inline chunk-based deduplication. In *11th USENIX Conference on File and Storage Technologies (FAST 13)*, 2013.

[41] Mark Lillibridge, Kave Eshghi, Deepavali Bhagwat, Vinay Deolalikar, Greg Trezise, and Peter Camble. Sparse indexing: Large scale, inline deduplication using sampling and locality. In *7th Conference on File and Storage Technologies (FAST 09)*, 2009.

[42] Xing Lin, Guanlin Lu, Fred Douglis, Philip Shilane, and Grant Wallace. Migratory compression: Coarse-grained data reordering to improve compressibility. In *12th USENIX Conference on File and Storage Technologies (FAST 14)*, 2014.

[43] Chenyang Lu, Guillermo A. Alvarez, and John Wilkes. Aqueduct: Online data migration with performance guarantees. In *1st USENIX Conference on File and Storage Technologies (FAST 02)*, 2002.

[44] Udi Manber. Finding similar files in a large file system. In *USENIX Winter 1994 Technical Conference (WTEC 94)*, 1994.

[45] Keiichi Matsuzawa, Mitsuo Hayasaka, and Takahiro Shinagawa. The quick migration of file servers. In *11th ACM International Systems and Storage Conference (SYSTOR 18)*, 2018.

[46] Dirk Meister, Jürgen Kaiser, Andre Brinkmann, Toni Cortes, Michael Kuhn, and Julian Kunkel. A study on data deduplication in HPC storage systems. In *International Conference on High Performance Computing, Networking, Storage and Analysis (SC 12)*, 2012.

[47] Dutch T. Meyer and William J. Bolosky. A study of practical deduplication. In *9th USENIX Conference on File and Stroage Technologies (FAST 11)*, 2011.

[48] Athicha Muthitacharoen, Benjie Chen, and David Mazières. A low-bandwidth network file system. In *18th ACM Symposium on Operating Systems Principles (SOSP 01)*, 2001.

[49] P. C. Nagesh and Atish Kathpal. Rangoli: Space management in deduplication environments. In *6th International Systems and Storage Conference (SYSTOR 13)*, 2013.

[50] Youngjin Nam, Guanlin Lu, Nohhyun Park, Weijun Xiao, and David H. C. Du. Chunk fragmentation level: An effective indicator for read performance degradation in deduplication storage. In *2011 IEEE International Conference on High Performance Computing and Communications (HPCC 11)*, 2011.

[51] A. Richards and J. P. How. Aircraft trajectory planning with collision avoidance using mixed integer linear programming. In *Proceedings of the 2002 American Control Conference (IEEE Cat. No.CH37301)*, volume 3, pages 1936–1941, May 2002.

[52] Prateek Sharma and Purushottam Kulkarni. Singleton: System-wide page deduplication in virtual environments. In *21st International Symposium on High-Performance Parallel and Distributed Computing (HPDC 12)*, 2012.

[53] Philip Shilane, Ravi Chitloor, and Uday Kiran Jonnala. 99 deduplication problems. In *8th USENIX Workshop on Hot Topics in Storage and File Systems (HotStorage 16)*, 2016.

[54] Kiran Srinivasan, Tim Bisson, Garth Goodson, and Kaladhar Voruganti. iDedup: Latency-aware, inline data deduplication for primary storage. In *10th USENIX Conference on File and Storage Technologies (FAST 12)*, 2012.

[55] Mark W. Storer, Kevin Greenan, Darrell D.E. Long, and Ethan L. Miller. Secure data deduplication. In *ACM International Workshop on Storage Security and Survivability (StorageSS '08)*, 2008.

[56] John D. Strunk, Eno Thereska, Christos Faloutsos, and Gregory R. Ganger. Using utility to provision storage systems. In *6th USENIX Conference on File and Storage Technologies (FAST 08)*, 2008.

[57] Zhen Sun, Geoff Kuenning, Sonam Mandal, Philip Shilane, Vasily Tarasov, Nong Xiao, and Erez Zadok. A long-term user-centric analysis of deduplication patterns. In *32nd Symposium on Mass Storage Systems and Technologies (MSST 16)*, 2016.

[58] Vasily Tarasov, Amar Mudrankit, Will Buik, Philip Shilane, Geoff Kuenning, and Erez Zadok. Generating realistic datasets for deduplication analysis. In *2012 USENIX Annual Technical Conference (USENIX ATC 12)*, 2012.

[59] Nguyen Tran, Marcos K. Aguilera, and Mahesh Balakrishnan. Online migration for geo-distributed storage systems. In *2011 USENIX Conference on USENIX Annual Technical Conference (USENIX ATC 11)*, 2011.

[60] Carl A. Waldspurger. Memory resource management in VMware ESX server. *ACM SIGOPS Operating Systems Review - OSDI '02*, 36(SI):181–194, December 2002.

[61] Grant Wallace, Fred Douglis, Hangwei Qian, Philip Shilane, Stephen Smaldone, Mark Chamness, and Windsor Hsu. Characteristics of backup workloads in production systems. In *10th USENIX Conference on File and Storage Technologies (FAST 12)*, 2012.

[62] Nai Xia, Chen Tian, Yan Luo, Hang Liu, and Xiaoliang Wang. UKSM: Swift memory deduplication via hierarchical and adaptive memory region distilling. In *16th USENIX Conference on File and Storage Technologies (FAST 18)*, 2018.

[63] Wen Xia, Hong Jiang, Dan Feng, Lei Tian, Min Fu, and Yukun Zhou. Ddelta: A deduplication-inspired fast delta compression approach. *Performance Evaluation*, 79:258 – 272, 2014. Special Issue: Performance 2014.

[64] Wen Xia, Yukun Zhou, Hong Jiang, Dan Feng, Yu Hua, Yuchong Hu, Qing Liu, and Yucheng Zhang. FastCDC: A fast and efficient content-defined chunking approach for data deduplication. In *2016 USENIX Annual Technical Conference (USENIX ATC 16)*, 2016.

[65] Zhichao Yan, Hong Jiang, Yujuan Tan, and Hao Luo. Deduplicating compressed contents in cloud storage environment. In *8th USENIX Workshop on Hot Topics in Storage and File Systems (HotStorage 16)*, 2016.

[66] Yanhua Zhang, X. Sun, and Baowei Wang. Efficient algorithm for k-barrier coverage based on integer linear programming. *China Communications*, 13(7):16–23, July 2016.

[67] zhichao Cao, Hao Wen, Fenggang Wu, and David H.C. Du. ALACC: Accelerating restore performance of data deduplication systems using adaptive look-ahead window assisted chunk caching. In *16th USENIX Conference on File and Storage Technologies (FAST 18)*, 2018.

[68] Benjamin Zhu, Kai Li, and Hugo Patterson. Avoiding the disk bottleneck in the Data Domain deduplication file system. In *6th USENIX Conference on File and Storage Technologies (FAST 08)*, 2008.

[69] Charlie Shucheng Zhu, Georg Weissenbacher, and Sharad Malik. Coverage-based trace signal selection for fault localisation in post-silicon validation. In *Hardware and Software: Verification and Testing - 8th International Haifa Verification Conference (HVC 12)*, 2012.