הטכניון
מכון טכנולוגי לישראל
TECHNION
Israel Institute of Technology

ספריות הטכניון
The Technion Libraries

בית הספר ללימודי מוסמכים ע"ש ארווין וג'ואן ג'ייקובס
Irwin and Joan Jacobs Graduate School

ILP Based Load Balancing in Deduplicated Storage Systems

Ariel Kolikant

ILP Based Load Balancing in Deduplicated Storage Systems

Research Thesis

Submitted in partial fulfillment of the requirements
for the degree of Master of Science in Computer Science

Ariel Kolikant

Submitted to the Senate
of the Technion — Israel Institute of Technology
Elul 5782 Haifa September 2022

This research was carried out under the supervision of Dr. Gala Yadgar, in The Henry and Marilyn Taub Faculty of Computer Science.

Some results in this thesis have been published as articles by the author and research collaborators in conferences and journals during the course of the author's research period, the most up-to-date versions of which being:

Roei Kisous, Ariel Kolikant, Abhinav Duggal, Sarai Sheinvald, and Gala Yadgar. The what, the from, and the to: The migration games in deduplicated systems. To appear in the Special Section on FAST 22 in the Transaction on Storage.

Roei Kisous, Ariel Kolikant, Abhinav Duggal, Sarai Sheinvald, and Gala Yadgar. The what, the from, and the to: The migration games in deduplicated systems. In *20th USENIX Conference on File and Storage Technologies (FAST 22)*, Santa Clara, CA, February 2022. USENIX Association.

The generous financial help of the Technion is gratefully acknowledged. This research was supported by the Israel Science Foundation (grant No.807/20).

Contents

List of Figures

Abstract **1**

Abbreviations and Notations **3**

1 Introduction **5**

2 Background and Related Work **7**
2.1 ILP . . . 7
2.2 Data deduplication . . . 7
2.3 Data migration in distributed deduplication systems . . . 9

3 Motivation and problem statement **11**
3.1 Motivation . . . 11
3.2 Problem Statement . . . 12

4 Greedy Method **15**

5 ILP Based Method **19**

6 Evaluation **25**
6.1 Experimental Setup . . . 25
6.2 Implementation . . . 26
6.2.1 Greedy . . . 26
6.2.2 ILP . . . 27
6.3 Basic comparison between algorithms . . . 27
6.3.1 Deletion . . . 27
6.3.2 Load Balance . . . 28
6.3.3 Runtime . . . 29
6.3.4 Implications . . . 29
6.4 Sensitivity to problem parameters . . . 30
6.4.1 Sampling degree . . . 30
6.4.2 Time limit . . . 31
6.4.3 Load balancing and traffic constraints . . . 32
6.4.4 Number of volumes . . . 32

6.5 Additional sampling 33

7 Conclusions and open questions **37**

Hebrew Abstract **i**

List of Figures

2.1 Initial system (a) and alternative migration plans: with optimal balance (b), optimal traffic (c), and optimal deletion (d). All the blocks in the system are of size 1. 8

4.1 Overview of our extended greedy algorithm. 16

5.1 Sampling by sample size k and extra sample size k' 22

6.1 Reduction in system size of all systems and all algorithms (with and without load balancing constraints. $k = 13$ and $\mu = 2\%$). 28
6.2 Resulting balance of all systems and all algorithms (with and without load balancing constraints. $k = 13$ and $\mu = 2\%$). 28
6.3 Algorithm runtime for all systems and all algorithms (with and without load balancing constraints. $k = 13$ and $\mu = 2\%$). 29
6.4 Linux-skip system with 5 volumes, $\mu = 2\%$, and two sampling degrees: $k = 8, 13$. 30
6.5 Deletion in UBC-500 system, $\mu = 2\%$, increasing timeout values 31
6.6 UBC-500 system with $k = 13$ and different load balancing margins. 32
6.7 Linux-skip with different numbers of target volumes with $T_{max} = 100, k = 13, \mu = 2\%$. . 33
6.8 Deletion, balance, and runtime of UBC-500 with different additional sampling sizes with different values of $T_{max}, K, andk'$. 34
6.9 Deletion, balance, and runtime of Linux-skip with different additional sampling sizes with different values of $T_{max}, K, andk'$. The red 'x' marks represent experiments where the ILP solver determined that no solution meets the problem's constraints (infeasible). . . . 34

Abstract

Deduplication reduces the size of the data stored in large-scale storage systems by replacing duplicate data blocks with references to their unique copies. This creates dependencies between files that contain similar content and complicates the management of data in the system. In the work presented in this thesis, we have addressed the problem of data migration and load balancing, where files are remapped between different volumes because of system expansion or maintenance.

The challenge of determining which files and blocks to migrate has been studied extensively for systems without deduplication. In the context of deduplicated storage, however, only simplified migration scenarios were considered and those were not extended into the broader load balancing problem.

In our work we have formulated the general migration problem of deduplicated systems as an optimization problem whose objective is to minimize the system's size while ensuring that the storage load is evenly distributed between the system's volumes, and that the network traffic required for the migration does not exceed its allocation. We extended the migration problem to address the load balancing problem limitations.

We then created an algorithm based on the ILP formulation, to solve the migration problem. We then compared it's results to two other algorithms solving the same generated migration and load balancing problem: the greedy algorithm and the clustering algorithm. Our ILP algorithm manages to consistently obtain the best solutions to the problem though it requires significantly larger execution times.

Abbreviations and Notations

V	:	Storage volume
B_v	:	Set of unique blocks stored on V
F_v	:	Set of files mapped to V
μ	:	Load balancing tolerance margin
k	:	Sampling degree used in fingerprint sampling acceleration method

Chapter 1

Introduction

Many large-scale storage systems employ data deduplication to reduce the size of the data that they store. The deduplication process identifies duplicate data blocks in different files and replaces them with pointers to a unique copy of the block stored in the system. This reduction in the system's size comes at the cost of increased system complexity. While the complexity of reading, writing, and deleting data in deduplicated storage systems has been addressed by many academic studies and commercial systems, the high-level management aspects of large-scale systems, such as capacity planning, caching, and quality and cost of service, still need to be adapted to deduplicated storage [SCJ16].

This work focuses on the aspect of *data migration*, where files are remapped between separate deduplication domains, or *volumes*. A volume may represent a single server within a large-scale system, or an independent set of servers dedicated to a customer or dataset. Files might be remapped as a result of volumes reaching their capacity limitation or of other bottlenecks forming in the system.

Deduplication introduces new considerations when choosing which files to migrate, due to the data dependencies between files: when a file is migrated, some of its blocks may be deleted from its original volume, while others might still belong to files that remain on that volume. Similarly, some blocks need to be transferred to the target volume, while others may already be stored there. An efficient migration plan must optimize several, possibly conflicting objectives: the physical size of the stored data after migration, the load balancing between the system's volumes, and the network bandwidth generated by the migration itself.

Several recent studies address specific (simplified) cases of data migration in deduplicated systems. Harnik et al. [HHS+19] address capacity estimation and propose a greedy algorithm for reducing the system's size. Rangoli [NK13] is a greedy algorithm for *space reclamation*, where a set of files is deleted to reclaim some of the system's capacity. GoSeed [NSKY21] is an ILP (integer linear programming)-based algorithm for the *seeding* problem, in which files are remapped into an initially empty target volume. While even the seeding problem is shown to be NP-hard [NSKY21], none of these studies address the conflicting objectives involved in the full data migration problem. Namely, the tradeoff between minimizing the system size, minimizing the network traffic consumed during migration, and maximizing the load balance between the volumes in the system.

In this work, we address, for the first time, the general case of data migration. We begin by

formulating the data migration problem in its most general form, as an optimization problem whose main goal is to minimize the overall size of the system. We add the traffic and load balancing considerations as constraints on the migration plan. The degree in which these constraints are enforced directly affects the solution space, allowing the system administrator to prioritize different costs. Thus, the problem of data migration in deduplication systems maps to finding what to migrate, where to migrate from, and where to migrate to within the traffic and load balancing constraints specified by the administrator.

We then introduce two novel algorithms for generating an efficient migration plan. The first is a greedy algorithm that is inspired by the greedy iterative process in [HHS+19]. Our extended algorithm distributes the data evenly between volumes while ensuring that the migration traffic does not exceed the maximum allocation. By breaking this process into several phases, we ensure that the allocated traffic is used for both load balancing and capacity reduction, balancing between the two (possibly conflicting) goals.

Our second algorithm is inspired by the ILP-based approach of GoSeed. We reformulate the ILP problem with variables and constraints that express the traffic used during migration and the choice of volumes from which to remap files or to remap files onto. Our formulation for the general migration problem is naturally much more complex than the one required for seeding. Nevertheless, we successfully applied it to data migration in systems with hundreds of millions of blocks.

This work has been done concurrently with research done at [KKD+22] on the clustering algorithm. The Greedy and ILP algorithms presented in this work were mainly evaluated against the clustering algorithm.

We implemented our two algorithms and evaluated them on six system snapshots created from three public datasets [FSL, MB11, Lin]. Our results demonstrate that all algorithms can successfully reduce the system's size while complying with the traffic and load balancing constraints. Each algorithm has different advantages: the greedy algorithm produces a migration plan in the shortest runtime (often several seconds), although its reduction in system size is typically lower than that of the other algorithms. The ILP-based approach can efficiently utilize the allowed traffic consumption, and improve as the load balancing constraints are relaxed. However, its execution must be timed out on the large problem instances, which often prevents it from yielding an optimal migration plan. The clustering algorithm achieves comparable results to those of the ILP-based approach, and sometimes even exceeds them. It does so in much shorter runtimes.

Chapter 2

Background and Related Work

2.1 ILP

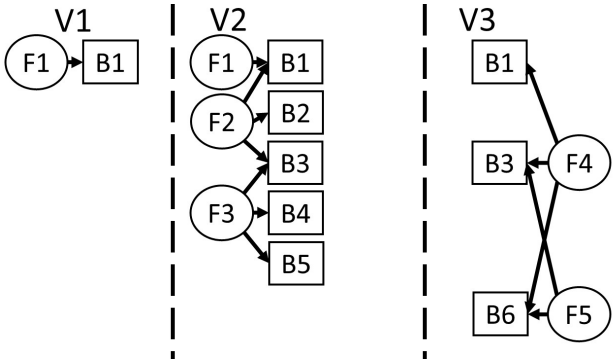
Integer linear programming (ILP) is an optimization problem. The input of an ILP problem is a vector x, a set Ax and an objective function Tx. Vector x represents the model's variables so that $x_1, x_2, ..., x_{n-1}, c \in \mathbf{Z}$. Linear constraints Ax represents the domain of legal solutions, each constraint being of the form $a_0x_0 + a_1x_1 + ... + a_{n-1}x_{n-1} \leq c$ such that $a_1, a_2, ..., a_{n-1}, c \in \mathbf{Z}$ are known parameters. The objective function Tx represents what the model seeks to optimize, and is of the form $t_0x_0 + t_1x_1 + ... + t_{n-1}x_{n-1}$.

The solution of an ILP problem is an integer assignment of vector x that satisfies constraints Ax and maximizes Tx. However, even a relaxed problem where the goal is finding whether a solution exists where x is restricted to Booleans, is long known to be NP-complete [Kar72]. The reason ILP problems have become popular despite having no theoretical efficient solver, is due to the existence of efficient ILP solvers [Gur, CPL] that can solve practical instances. Thus the formulating of problems into ILP models has been used to find practical solutions to various problems in a a wide range of fields [RH02, Aba89, ZWM13, ZSW16].

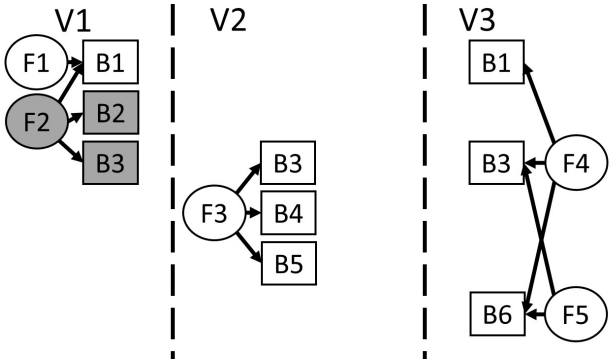
2.2 Data deduplication

In a nutshell, the deduplication process splits incoming data into fixed or variable sized chunks, which we refer to as *blocks*. The content of each block is hashed to create a *fingerprint*, which is used to identify duplicate blocks and to retrieve their unique copy from storage. Several aspects of this process must be optimized so as not to interfere with storage system performance. These include chunking and fingerprinting [Man94, XJF+14, XZJ+16, MCM01, AAA+10], indexing and lookups [ZLP08, SBGV12, ADK+18], efficient storage of blocks [LEB+09, LSD+14, DSL10, SBGV12, CLZ11, YJTL16, LLD+14], and fast file reconstruction [FFH+14, zCWWD18, LEB13, KBKD12]. Although the first commercial systems used deduplication for backup and archival data, deduplication is now commonly used in high-end primary storage.

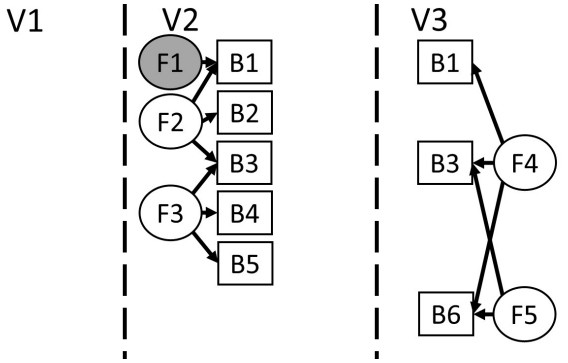
(a) Initial system: balance = 1/5



(b) Alternative 1: deletion=0, traffic=2, **balance=1**



(c) Alternative 2: deletion=1/9, **traffic=0**, balance=0



(d) Alternative 3: **deletion**=3/9, traffic=1, balance=0

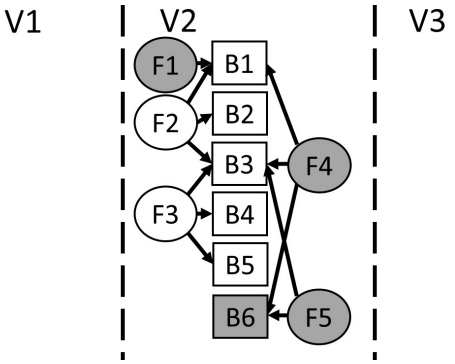


Figure 2.1: Initial system (a) and alternative migration plans: with optimal balance (b), optimal traffic (c), and optimal deletion (d). All the blocks in the system are of size 1.

2.3 Data migration in distributed deduplication systems

Numerous distributed deduplication designs were introduced in commercial and academic studies [CAVL09, DGH+09, GE11]. We focus on designs that employ a separate fingerprint index in each physical server [DDL+11, BELL09, HHS+19, BLC14, DDS+17]. This design choice maintains a small index size and a low lookup cost, facilitates garbage collection at the server level, and simplifies the client-side logic. In this design, each server (*volume*) is a separate *deduplication domain*, i.e., duplicate blocks are identified only within the same volume. Recipes of files mapped to a specific volume thus point to blocks that are physically stored in that volume.

The coupling of the logical file's location and the physical location of its blocks implies that when a file is remapped from its volume, we must ensure that all its blocks are stored in the new volume. At the same time, the file's blocks cannot necessarily be removed from its original volume, because they might also belong to other files. For example, consider the initial system depicted in Figure 2.1(a), and assume we remap file F_2 from volume V_2 to volume V_1, resulting in the alternative system in Figure 2.1(b). Block B_1 is deleted from V_2 because it is already stored in V_1. Block B_2 is deleted from V_2, but must be copied to V_1, because it wasn't there in the initial system. Block B_3 must also be copied to V_1, but is not deleted from V_2 because it also belongs to F_3. The total sizes of the initial system and of this alternative are the same: nine blocks.

Various approaches to data migration in distributed deduplication systems have been presented in previous works. The work presented in Harnik et al [HHS+19] presented a greedy iterative algorithm for reducing the total capacity of data in a system with multiple volumes. In their research, they defined the *space-saving ratio*: for volumes v, v' such that file f holds $f \in v, f \notin v'$; R is defined as the size of replicated blocks if f were to be migrated to v' and D is defined as the size of deleted blocks if f were to be migrated to v'. The space-saving ratio is defined as $\frac{R}{D}$.

Based on the definition of space-saving ratio, they developed a greedy algorithm: in each iteration, each file f has its space-saving ratio calculated for each volume v. The pair (f, v) represents the migration of file f to volume v and in each iteration the pair (f, v) with the smallest ratio is chosen to be migrated to volume v since it will result in the best space saving in that iteration. The algorithm stops when there are no legal migrations or if a predetermined deletion goal has been reached.

The work done in [NSKY21] addresses a simplified case of data migration called *seeding*, where the initial system consists of many files mapped to a single volume. The migration goal is to delete a portion of this volume's blocks by remapping files to an empty target volume [DJS+19]. GoSeed formulates the seeding problem as an ILP (integer linear programming) instance whose solution determines which files are remapped, which blocks are *moved* from the source volume to the target, and which are *replicated* to create copies on both volumes. This approach is made possible by the existence of open-source [SYM, lps, GNU] and commercial [CPL, Gur] ILP-solvers. GoSeed is applied to instances with millions of blocks with several acceleration heuristics, some of which we adapt to the generalized problem.

The Rangoli algorithm presented in [NK13] is a greedy algorithm for *space reclamation*—another specific case of data migration where a set of files is chosen for deletion in order to

delete a portion of the system's physical size. Unlike the greedy and ILP-based approaches that inspire our own algorithms, the problem solved by Rangoli is too simplified for it to be extended for general migration.

The clustering algorithm is described in [KKD+22] and was developed concurrently with our work. In our evaluation, it was the primary algorithm we compared our algorithms to. The Clustering algorithm is based on clustering [clu] which is a well known method for grouping objects based on similarity. The clustering algorithm uses *Hierarchical clustering* [GP13] an iterative clustering process that receives a set of initial clusters as input and in each iteration merges the most similar clusters into a new cluster. The initial clusters are viewed as clusters of size 1 and the merging of clusters creates a tree where the size 1 clusters constitute its leaves. In our context, the objects are files and their similarity is measured by the number of blocks shared between files. Files with similar blocks are joined into clusters. The resulting clusters represent a possible grouping of files which would have high block similarity. Those clusters are then mapped into volumes and the transition between the initial system volumes to the system volumes presented by the clusters is the migration plan produced by the algorithm.

Similarity between clusters is an essential definition for the hierarchical clustering process. In their work they used *Jaccard distance* as the metric for the cluster similarity. For sets A and B, $J(A,B)$ is defined as $\frac{|A \cap B|}{|A \cup B|}$. The *Jaccard distance* between files A and B is defined as $dist_J = \overline{J(A,B)} = 1 - J(A,B)$. Despite being a heuristic based algorithm that does not reach the mathematically optimal solution, the clustering algorithm achieves results that are close to the optimal solutions.

Both GoSeed and Rangoli address a very specific case of data migration in deduplicated systems. Harnik et al. address the general data migration problem but does not cover all data migration considerations taken in traditional (non-deduplicated) storage systems. The clustering algorithm, having been developed in conjunction with ILP and Greedy, makes similar assumptions to our work regarding the storage system. The clustering algorithm also takes similar consideration to our work, This makes it an ideal candidate for comparison in our evaluation.

Chapter 3

Motivation and problem statement

3.1 Motivation

Data migration in traditional (non-deduplicated) storage systems takes into account various often conflicting objectives when creating a migration plan. However in the various works done about data migration in deduplicated systems, only the objective of storage size reduction has been addressed. In this work we attempt to solve the migration problem for deduplicated systems with two additional objectives that are of equal importance to size reduction. The two objectives we added are: minimizing migration traffic and load balancing.

Minimizing migration traffic. In most storage systems, the amount of network bandwidth available for maintenance activities is limited [RSG+13, HSX+12]. On non-deduplicated storage systems, traditional migration plans take the bandwidth limitation into account and minimize its usage as part of their optimization [LAW02, MHS18, TAB11, DJS+19, AHK+02, AHH+01]. However, previous studies of data migration in deduplicated systems have not explicitly taken traffic minimization into account. Harnik et al. [HHS+19] do not take it into account at all. The work solving the more relaxed seeding problem implicitly minimizes the bandwidth usage by minimizing the storage size in GoSeed [NSKY21]. Our work focuses on bandwidth as it relates to data transfer between nodes. The physical layout of nodes and their precise scheduling are out of the scope of this work. Data migration costs can be divided into two types based on the bandwidth aspect. The *migration traffic* is the amount of data that would be transferred between volumes during the migration. *Replication cost* is the total number of blocks that would be duplicated as a result of the migration. Since replicated blocks are necessarily transferred between volumes, the migration traffic depends on data replication, yet they are not always equal to each other.

For example, Alternative 1 in Figure 2.1(b) results in transferring two blocks between volumes, B_2 and B_3, even though B_2 is eventually deleted from its source volume. In contrast, the alternative migration plan in Figure 2.1(c) does not consume traffic at all: file F_1 is remapped to V_2 which already stores its only block, so B_1 can be deleted from V_1. This alternative also reduces the system's size to eight blocks, making it superior to the first alternative in terms of both objectives. In some storage systems the objective of minimizing the traffic may conflict with the objective of minimizing the system's overall size.

Load Balancing. Load balancing is one of the most important objectives of any distributed

storage system, however it often conflicts with other system objectives such as utilization and management overhead [AHK+02, WBMM06, NEF+12]. Load balancing can refer to: IOPS, bandwidth requirements, or the number of files that belong to each volume. Here we refer to the *capacity load* between volumes, as in previous works [BELL09, DDL+11]. In deduplication systems where new files are routed to volumes containing existing files, capacity load balancing is particularly important. Volumes with large capacities are more likely to contain similarities to migrated files, thus creating a snowball effect as files would be migrated to the largest volumes resulting in more similarities for f uture migrations.

The load balancing objective conflicts with our first objective of size reduction. The best case for size reduction would occur if all files were migrated to a single volume, which would result in the worst load balancing possible. It means that distributed deduplicated storage systems have to weigh the benefits of remapping a file to a volume where it would save space and remapping it to a volume where it would better distribute the load. Performance load balancing is the distribution of storage in the volumes in a way that ensures similar performance across all volumes. Performance load balancing is not addressed directly in this work, yet it is implicitly improved by load balancing. Both the methods presented in this work can be extended to address performance load balancing explicitly.

In this research, we measure load balancing using the *balance* metric, which is similar to the *fairness* metric [GWM07]—the ratio between the size of the smallest volume and the largest volume in the storage system. For example, the balance of the initial system in Figure 2.1(a) is $|V_1|/|V_2| = 1/5$. Alternative 1 (Figure 2.1(b)) is perfectly balanced, with $balance = 1$, while Alternative 2 (Figure 2.1(c)) has the worst balance: $|V_1|/|V_2| = 0$.

3.2 Problem Statement

For a storage system S with a set of volumes V, let $B = \{b_0, b_1, \ldots\}$ be the set of unique blocks stored in the system, and let $size(b)$ be the size of block b. Let $F = [f_0, f_1, \ldots]$ be the set of files in S, and let $I_S \subseteq B \times F \times V$ be an inclusion relation, where $(b, f, v) \in I_S$ means that file f mapped to volume v contains block b that is stored in this volume. The expression $b \in v$ indicates that $(b, f, v) \in I_S$ for some file f, while the expression $f \in v$ indicates that $(b, f, v) \in I_S$ for some block b. The size of a volume is equal to the total size of the unique blocks stored in it, i.e., $size(v) = \Sigma_{b \in v} size(b)$. The size of the system is the total size of its volumes, i.e., $size(S) = size(V) = \Sigma_{v \in V} size(v)$.

A migration plan F_M is a set of files and target volumes $F_M \subseteq F \times V$. $(f, v) \in F_M$ indicates that the migration plan would have file f moved to volume v. Since we know the initial state of files and blocks in the volumes, we can deduce the final state of the blocks by the file movements. New system S' is defined as the result of applying migration plan F_M on system S and it has the resulting I'_S and V'

The *general migration problem* is to find F_M which results in the minimum size of S'. This is equivalent to finding F_M which maximizes the difference between *blocks that can be deleted* and *blocks that must be replicated*. Migration plan F_M must uphold the *traffic constraint* and the *load balancing constraint*. We denote T_{max} as the maximum traffic allowed during the execution of a migration plan.

The *traffic constraint* sets a limit of T_{max} on the amount of bandwidth that can be used. Migrating a block would use bandwidth in the case where it is migrated to a volume that did not contain it in the initial system. Since the storage system is a deduplicated storage system, at most one instance of a block is needed in any volume and the migration of the same blocks to a volume would only use the bandwidth once.If a block is not moved into a volume, or if a block existed in a volume before being moved, it is not required to physically copy the block and thus no bandwidth would be used in order to get a copy of the block into the the target volume and it is not counted into the traffic constraint.

The *load balancing constraint* ensures that the migration plan would result in a storage system maintaining the desired load balance. Demanding a strict equality between the final balance and desired balance limits the amount of possible migration plans, often resulting in no migration plans at all. Thus, we present a tolerance margin μ in the load balancing constraint. The load balancing constraint with a tolerance margin requires that every volume v in the new system S' would have a size within μ of the average new system volume size. $\forall v' \in V' : |\frac{size(V')}{|V'|} - size(v')| \leq size(V') * \mu$

For example, for the initial system in Figure 2.1(a), Alternative 1 (Figure 2.1(b)) is the only migration plan that satisfies the load balancing constraint (for any μ). For T_{max} lower than $2/9$, no migration is feasible. On the other hand, if we remove the load balancing constraint, the optimal migration plan will depend on the traffic constraint alone: Alternative 2 (Figure 2.1(c)) is optimal for, e.g., $T_{max} = 0$, and Alternative 3 (Figure 2.1(d)) is optimal for $T_{max} = 3$.

Chapter 4

Greedy Method

The basic greedy algorithm by Harnik et al. [HHS+19] iterates over all the files in each volume, and calculates the *space-saving ratio* from remapping a single file to each of the other volumes: the ratio between the total size of the blocks that would be replicated and the blocks that would be deleted from the file's original volume. In each iteration, the file with the lowest ratio is remapped. For example, if this basic greedy algorithm was applied to the initial system in Figure 2.1(a), it would first remap file F_1 to volume V_2, with a space-saving ratio of 0, resulting in Alternative 2 (Figure 2.1(c)). The process halts when the total capacity is reduced by a predetermined deletion goal. This algorithm is not directly applicable to the general migration problem because it does not consider traffic and load balancing.

Addressing the traffic constraint is relatively straightforward. In our extended greedy algorithm we make it the halting condition: the iterations stop when there is no file that can be remapped without exceeding the maximum allocated traffic. A small challenge is that a file might be remapped in several iterations of the algorithm, while, in the resulting migration plan, it will only be remapped from its original volume to its final destination. As a result, the sum of traffic of all the individual iterations can be (and is, in practice) higher than the traffic required when executing migration plan. This will not violate the traffic constraint, but will cause the algorithm to halt before taking advantage of the maximum allowed traffic. Thus, we heuristically allow the algorithm to use 20% more traffic than the original traffic constraint, to prevent it from halting prematurely. We include this simple extension, without a load-balancing constraint, in our evaluation.

Complying with the load-balancing constraint is more challenging. For example, if the basic greedy algorithm reached Alternative 2 (Figure 2.1(c)), it could no longer remap any single file to volume V_1 without increasing the system's capacity, and thus the system will remain unbalanced with at least one empty volume. A naive extension to this algorithm could enforce the load-balancing constraint by preventing files from being remapped if this increases the system's imbalance. However, such a strict requirement might preclude too many opportunities for optimization. For example, for the initial system in Figure 2.1(a), it would only allow to remap file F_2 to volume V_1, resulting in Alternative 1 (Figure 2.1(b)). The system would be perfectly balanced, but the basic algorithm would then terminate without reducing its size at all.

We address this challenge with two main techniques. The first is defining two iteration types:

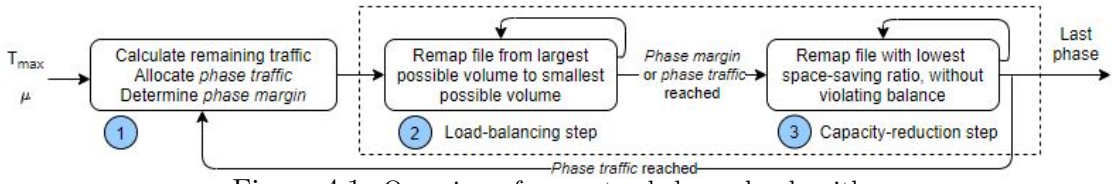


Figure 4.1: Overview of our extended greedy algorithm.

one whose goal is to balance the system's load, and another whose goal is to reduce its size. We perform these iterations interchangeably, to avoid the entire allocated traffic from being spent on only one goal. The second technique is to relax the load-balancing margin for the early iterations and continuously tighten it until the end of the execution. The idea is to let the early iterations remap files more freely, and to ensure that the iterations at the end of the algorithm result in a balanced system.

Figure 4.1 illustrates the process of our extended greedy algorithm. We divide the algorithm's process into phases. ① Each phase is allocated an even portion of the traffic allocated for migration, and is limited by a local load-balancing constraint. Each phase is composed of two steps. ② The *load-balancing step* iteratively remaps files from large volumes to small ones, until the volume sizes are within the margin defined for this phase, or its traffic is exhausted. ③ The *capacity-reduction step* uses the remaining traffic to reduce the system's size by remapping files between volumes, ensuring that volume sizes remain within the margin.

Each phase is limited by **local traffic and load-balancing constraints**, calculated at the beginning of the phase. The *phase traffic* determines the maximum traffic that can be used in each phase, and is roughly even for all the phases. The local *phase margin* determines the minimum and maximum allowed volume sizes in each phase. It is larger than the global margin, μ, in the first phase, and gradually decreases before each phase, until reaching μ in the last phase. By default, our greedy algorithm consists of $p = 5$ phases. The phase traffic for phase i, $0 \leq i < p$, is $1/(p-i)$ of the unused traffic, and the phase margin for the first phase is $\mu \times 1.5$. We have observed that increasing the number of phases correlates with better size reduction, but once the number of phases exceeds 5, there is usually too little traffic to ensure load balancing. Based on this observation, we have set the default number of phases to 5 and all our evaluations were done with this default value.

The load balancing step is the first step in each phase. In each of its iterations, the volumes are sorted according to their sizes, and we attempt to remap files from the largest volumes to the small ones. A file can be remapped only if some blocks will be deleted from its source volume as a result. Namely, we look for a file to remap between a $\langle source, target \rangle$ pair of volumes, where *source* is the largest volume and the *target* is the smallest volume for which such a file exists. In each iteration, the amount of traffic required to remap the chosen file is calculated, and the iterations halt when the maximum allowed traffic or allowed volume sizes are reached.

The capacity-reduction step uses the remaining traffic allocation of the phase. It is similar to the original greedy algorithm, but it ensures that each file remap does not cause the volumes to become unbalanced. In other words, we can remap a file only if this does not cause its source volume to become too small, or its target volume to become too large. Note that the amount of traffic that remains for the capacity-reduction step depends on the degree of

imbalance in the initial system. In the most extreme case of a highly unbalanced system, it is possible for the load balancing step to consume all the traffic allocated for the phase. In this case, the capacity-reduction step halts in the first iteration. For cases other than this extreme, a higher number of phases can divert more traffic for capacity-reduction, at the cost of longer computation time due to the increased number of iterations.

Chapter 5

ILP Based Method

Our ILP-based approach is inspired by GoSeed [NSKY21], designed for the seeding problem, where files can only be remapped from the source volume to the empty target volume. GoSeed thus defined three types of variables whose assignment specified (1) whether a file is *remapped*, (2) whether a block is *replicated* on both volumes, and (3) whether a block is deleted from the source and *moved* to the target. These limited options resulted in a fairly simple set of constraints, which cannot be directly applied to the general migration problem. The major difference is that the decision of whether or not we can delete a block from its source volume depends not only on the files initially mapped to this volume, but also on the files that will be remapped to it as a result of the migration. Thus, in our ILP-based approach, every block transfer is modeled as creating a copy of this block, and a separate decision is made whether or not to delete the block from its source volume.

The problem's constraints are defined on the set of volumes, files, and blocks, from the problem statement in Chapter 3, the maximum traffic T_{max}, and load-balancing margin μ. We define the target size of each volume v as w_v, given as percentage of the system size after migration. By default, $w_v = 1/|V|$. The constraints are expressed in terms of three types of variables that denote the actions performed in the migration: x_{fst} denotes whether file f is *remapped* from its source volume s to another (target) volume t. c_{bst} denotes whether block b is *copied* from its source volume s to another (target) volume t. Finally, d_{bv} denotes whether block b is *deleted* from volume v. The solution to the ILP instance is an assignment of 0 or 1 to these variables. The resulting migration plan remaps the set of files for which $x_{fst} = 1$ (for some volume t), transfers the blocks for which $c_{bst} = 1$ to their respective target volume, and deletes the blocks for which $d_{bv} = 1$ from their respective volumes.

Constraints and objective. We model the migration problem as an ILP problem as follows. For every two volume $v_s, v_t \in V$, for every file $f_l \in F$ we allocate a Boolean variable x_{lst}. Assigning 1 to x_i means that f_l is remapped from v_s to v_t. For every block $b_i \in B$, for every volume $v_s \in V$ we allocate a Boolean variable d_{is}. Assigning 1 to d_{is} means that b_i was deleted from v_s. For every block $b_i \in B$, for every two volumes $v_s, v_t \in V$ we allocate a Boolean variable c_{ist}. Assigning 1 to c_{ist} means that b_i was copied from v_s to v_t. We annotate the initial intersect of volumes a group of sets $b_i \in intersect_{st}$ means that in the $b_i \in v_s, b_i \in v_t$.

We annotate $size(b_i)$ as the physical size of block b_i and $size(v_s)$ as $\Sigma_{b_i \in v_s} size(b_i)$. We annotate $Traffic$ as the maximum allowed traffic for the migration plan.

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

1. All the variables are Boolean: $0 \leq x_{lst} \leq 1, 0 \leq d_{is} \leq 1, 0 \leq c_{ist} \leq 1$ for every $f_l \in F$, $b_i \in B$ and $v_s, v_t \in V$.
2. A file can be remapped to at most one volume: $\Sigma_{v_t \in V} X_{lst} \leq 1$ for every $f_l \in F$ and $v_s, v_t \in V$.
3. A block can only be deleted or copied from a volume it was originally stored in: If $b_i \notin v_s$; $c_{ist} = d_{is} = 0$ for every $b_i \in B$ and $v_s, v_t \in V$.
4. A block can be deleted from a volume only if all the files containing it are remapped to other volumes: If $b_i \in f_l$ and $f_l \in v_s$; $d_{is} \leq x_{lst}$ for every $b_i \in B$, $f_l \in F$ and $v_s, v_t \in V$.
5. A block can be deleted from a volume only if no file containing it is remapped to this volume: if $b_i \in f_l$, $f_l \in v_s$ and $f_l \notin v_t$; $d_{it} \leq 1 - x_{lst}$ for every $b_i \in B$, $f_l \in F$ and $v_s, v_t \in V$.
6. Regard all intersect as having been copied: If $b_i \in intersect_{st}$; $c_{ist} = 1$ for every $b_i \in B$ and $v_s, v_t \in V$.
7. When a file is remapped, all its blocks are either copied to the target volume, or are initially there: If $b_i \in f_l$; $x_{lst} \leq \Sigma_{v \in V} c_{ist}$ for every $b_i \in B$ and $f_l \in F$.
8. A block can be copied to a target volume only from one source volume: $\Sigma_{s, b_i \notin Intersect_{st}} c_{ist} \leq 1$ for every $b_i \in B$ and $v_t \in V$.
9. A block must be deleted if there are no files containing it on the volume: $1 - \{\Sigma_{f_{l_s}}(1 - \Sigma_{v_m \in V} X_{l_s sm}) + \Sigma_{f_{l_v}}(x_{l_m ms})\}$ for every $v_s, v_m \in V$ and every $f_{lm} \in v_m$ and $f_{ls} \in v_s$.
10. A block cannot be copied to a target volume if no file will contain it there: $\Sigma_{v_s \in V} c_{ist} \leq \Sigma_{v_s \in V} \Sigma_{f_l \in V_s} X_{lst}$ for every $v_t \in V$, every $b_i \in B$ and $f_l \in F$ so that $b_i \in f_l$
11. A file cannot be migrated to its initial volume: $X_{lst} = 0$ for every $f_l \in F$ and $v_s, v_t \in V$.
12. *Traffic constraint:* The size of all the copied blocks is not higher than the maximum allowed traffic: $\Sigma_{v_s \in V} \Sigma_{t \in V} \Sigma_{b_i \notin intersect_{st}} c_{ist} * size(b_i) \leq Traffic$.
13. *Load balancing constraint:* For each volume v, $(w_v - \mu) \times Size(S') \leq size(v') \leq (w_v + \mu) \times Size(S')$, where $size(v')$ is the volume size after migration, i.e., the sum of its non-deleted blocks and blocks copied to it: $(W_t \mu) * (size(V) + TotalVolumeChange) \leq newBlocks + oldBlocks \leq (W_t + \mu) * size(V) + TotalVolumeChange$. for every volume $v_t \in V$
    (a) $TotalVolumeChange$: $\Sigma_{b_i \in B} * \Sigma_{v_s \in V}[-d_{is} + \Sigma_{v_t \in V, b_i \notin intersect_{st}} c_{ist}]$
    (b) $newBlocks$: $\Sigma_{v_s \in V, b_i \notin intersect_{st}} c_{ist} * size(b_i)$
    (c) $oldBlocks$: $\Sigma_{b_i \in v_t}(1 - d_{it} * size(b_i))$

▶ Objective: maximize the sum of sizes of all blocks that are deleted minus all blocks that are copied. This is equivalent to minimizing the overall system size: $Min(\Sigma_i b_i * \Sigma_{v_s \in V}[-dis + \Sigma_{v_t \in V, b_i \notin intersect_{st}} c_{ist}])$

Constraints 12 and 13 formulate the traffic and load-balancing goals, and constraints 8, 9, and 10 ensure that the solver does not create redundant copies of blocks to artificially comply with the load balancing constraint. This is similar to the constraint that prevents *orphan* blocks in the seeding problem [NSKY21]. For evaluation purposes, we will also refer to a relaxed

formulation of the problem without the load-balancing constraint. In that version, constraints 8, 9, 10, and 13 are removed, considerably reducing the problem complexity.

The ILP formulation given in this paper is designed for the most general case of data migration, where any file can be remapped to any volume. In reality, the migration goal might restrict some of the remapping options, potentially simplifying the ILP instance. For example, we can limit the set of volumes that files can be migrated to by eliminating the x_{fst} and c_{bst} variables where t is not in this set. We can similarly restrict the set of volumes files can be migrated from, or require that a set of specific files are (or are not) remapped.

Complexity and run time. The complexity of the ILP instance depends on $|B|$, $|F|$, and $|V|$—the number of blocks, files, and volumes, respectively. The number of variables is $|V|^2|F| + |V|^2|B| + |V| \times |B|$, corresponding to variable types x_{fst}, c_{bst}, and d_{bv}. Each of the constraints defined on these variables contributes a similar order of magnitude. An exception is constraint 13, which reformulates the system size, twice, to ensure each individual volume's size is within the required margin. Indeed, the relaxed formulation without this constraint is significantly simpler than the full formulation.

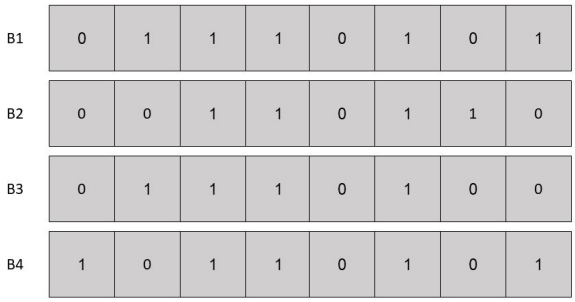
We use two of the acceleration methods suggested by GoSeed to address the high complexity of the ILP problem. The first is *fingerprint sampling*, where the problem is solved for a subset of the original system's blocks. This subset (*sample*) is generated by preprocessing the block fingerprints and including only those that match a predefined pattern. Specifically, as suggested in [HHS+19], a sample generated with sampling degree k includes only blocks whose fingerprints consist of k leading zeroes, reducing the number of blocks in the problem formulation by $1/2^k$ on average.

The second acceleration method is *solver timeout*, which halts the ILP solver's execution after a predetermined runtime. As a result, the server returns a feasible solution that is not necessarily optimal. We do not repeat the detailed analysis of the effectiveness of these heuristics, which were shown to be effective in earlier studies. Namely, the analysis of GoSeed showed that most of the solver's progress happens in the beginning of its execution (hence, timing out does not degrade its quality too much), and that it is more effective to reduce the sample size than to run the solver longer on a larger sample, as long as the sampling degree is not higher than $k = 13$.

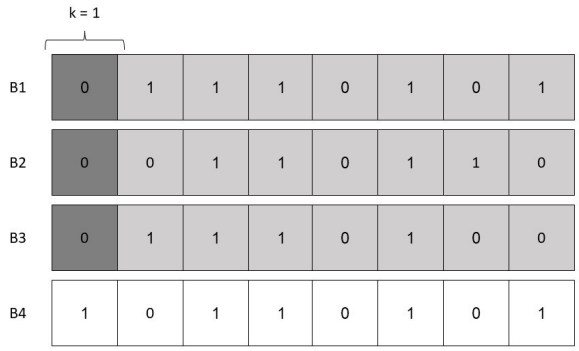
We have attempted a third acceleration method which we called *additional sampling*. The third acceleration method is based on two observations. Firstly we observe that the more solutions the ILP algorithm can check before a timeout, the closer the output solution would be to the optimal solution. Secondly we observe that larger ILP models require more time from the ILP solver to check each solution. Therefor large constraints that increase the size of the model result in solutions with lower size reduction and reducing the size of the constraints would allow for better solutions. While most constraints share the same complexity, constraint 11 (the load balancing constraint) is more complex than the other constraints by an order of magnitude since it reformulates the system size twice. By reducing the size of constraint 11 the ILP solver could check more solutions before reaching a timeout and result in a solution closer to the optimal solution.

Given the set of the original blocks of the system $B_{original}$ we create the sampled set of

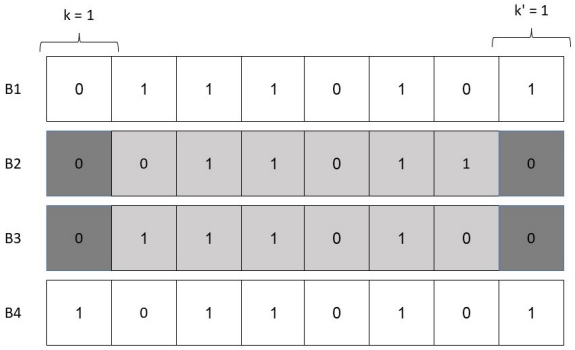
(a) Original group of blocks



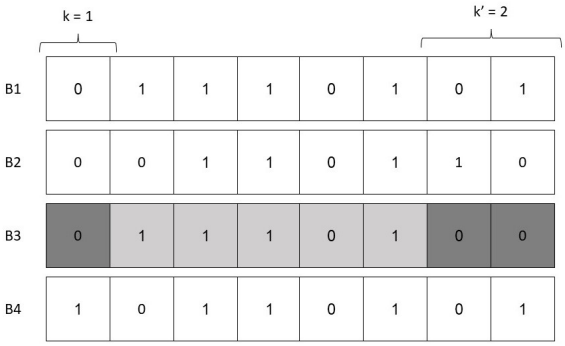
(b) sampling $k = 1$



(c) sampling and extra sampling $k = 1$, $k' = 1$



(d) sampling and extra sampling $k = 1$, $k' = 2$



(e) sampling and extra sampling $k = 2$, $k' = 1$

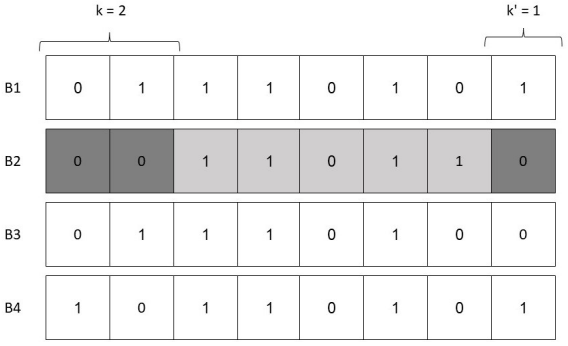


Figure 5.1: Sampling by sample size k and extra sample size k'

blocks according to the first acceleration method $B_{sampled} \subseteq B_{original}$, we then create a third set $B_{sampled'} \subseteq B_{sampled} \subseteq B_{original}$ that we use only for constraint 11. This additional sampling samples $B_{sampled}$ to create $B_{sampled'}$ in the same way the first acceleration method samples $B_{original}$ to create $B_{sampled}$. We perform the sketching of the fingerprints based on the trailing bits rather than the leading bits, to maintain the randomness of the sampling in a similar way to the first acceleration method. Figure 5.1 shows an example of 4 block fingerprints. Given $B_{original} = \{B_0, B_1, B_2, B_3\}$ and a sampling size of $k = 1$: $B_{sampled} = \{B_1, B_2, B_3\}$ since those are the only blocks in $B_{original}$ whose fingerprints have one leading zero. The additional sampling of $k' = 1$ would result in $B'_{sampled} = \{B_2, B_3\}$ since among the blocks in $B_{sampled}$ only B_2 and B_3 have one trailing zero. Using $k = 1, k' = 2$ will result in $B_{sampled'} = \{B_3\}$ while $k = 2, k' = 1$ will result in $B_{sampled'} = \{B_2\}$.

This method did not achieve the desired result of improving the migration plans' quality or reducing the runtimes. In our implementation, by default, this acceleration method is not used. We further explain our results in section 6.5.

Chapter 6

Evaluation

There are essentially two questions we want to answer: How do the algorithms compare on the final system size, load balancing, and runtime? And how are the various system and problem parameters affecting the performance of different algorithms? Here we describe our evaluation setup and the experiments we conducted in order to answer those questions.

6.1 Experimental Setup

We performed our experiments on a server that has the following specifications: Ubuntu 18.04.3, 128GB DDR4 RAM (with 2666 MHz bus speed), an Intel Xeon Silver 4114 processor running at 2.20GHz, one Dell 240GB TLC SATA SSD, and one Micron 5200 Series 960GB 3D TLC NAND Flash SSD.

The file system snapshots we used for creating the systems in our evaluation were derived from four different datasets. Two of these were used in GoSeed [NSKY21]: the UBC dataset [MB11, SNI] and the FSL dataset [FSL].UBC's dataset contains the file systems of 857 Microsoft employees, and we used the first 500 file systems of this dataset (UBC-500). The FSL dataset consists of snapshots of Stony Brook University's FSL Lab's student home directories. The data we used (Homes) comes from nine weekly snapshots from nine students between August 28 and October 23, 2014. They were created using variable sized chunks with Rabin fingerprints, with an average chunk size of 64KB.

For the purpose of this evaluation, we took the remaining sets of snapshots from [ESSY22] where two snapshots were created based on the Linux version archive [Lin]. Linux-all includes all versions from 2.0 to 5.9.14, with each snapshot presented as a file in our model. Linux-skip contains only every fifth version, meaning that it contains 5× less logical data, but has similar physical data size. We have created these two snapshots with an average chunk size of 8KB.

The six systems we compared are built to emulate a deduplication system where duplicates are detected only within a single volume, the systems are listed in Table 6.1. For the systems created from UBC or Linux snapshots, an equal number of arbitrary snapshots were assigned to each volume. For example, in the case of 5 volumes using UBC-500, each volume was assigned 100 snapshots as files. Using FSL (Homes) snapshots, systems were created so that each volume contains all the snapshots of either a group of users or a group of weeks. In Homes-weeks each volume presents a set of three weeks and contains all users' snapshots of those three weeks. For

System	Files	$\|V\|$	Chunks	Dedupe	Logical
UBC-500	500	5	382M	0.39	19.5 TB
Homes-week	81	3	19M	0.38	8.9 TB
Homes-user	81	3	19M	0.16	8.9 TB
Linux-skip	662	5 / 10	1.76M	0.12 / 0.19	377 GB
Linux-all	2703	5	1.78M	0.03	1.8 TB

Table 6.1: System snapshots in our evaluation. $|V|$ is the number of volumes, Chunks is the number of unique chunks, and Dedupe is the deduplication ratio—the ratio between the physical and logical size of each system. Logical is the logical size.

Homes-users, each volume contains all snapshots of three users over a given week.

6.2 Implementation

We ran the three algorithms using a sample of fingerprints [HHS+19] in order to reduce memory consumption and runtime. We calculated all the results with a sampling degree of $k = 13$ unless otherwise stated. We took the migration plans created on the sampled systems and emulated them on the original system using our *calculator* which we implemented in a similar fashion to the one used in [NSKY21]. First, the calculator calculates the initial volume sizes by their blocks. Second, the calculator simulates the file movements according to the migration plan. Third, the calculator calculates the final volume sizes by their blocks. Finally, the calculator infers for each volume the size that has been removed and the size that has been added. We made our evaluations with T_{max} values of 20%, 40%, and 100% of the correlating system's initial size, and μ values of 2%, 5%, 10%, and 15% of the system's final size. Unless explicitly stated, we've done all the evaluations with a timeout of 6 hours, however this timeout was only ever reached by the ILP algorithm.

Load balancing is the most limiting constraint and often in conflict with size reduction. For that reason, in addition to the evaluations we have done on the algorithms as described in chapters 5, 4, and in section 2.3, each algorithm has a *relaxed* version in which load balancing is not taken into account. These are marked in the evaluation with the (R) symbol.

6.2.1 Greedy

In order to calculate space-saving ratios efficiently, we maintain a *reference-count* matrix of size $V \times B$. The matrix cell $[v, b]$ contains the number of files in volume v that includes block b. Using this data structure, calculating the space-saving ratio for migrating a file to a volume can be done by reading the file's original snapshot and simulating the migration of blocks in accordance to the migration of files. Every iteration either terminates the algorithm or chooses one file for migration. Each time a file is chosen for an iteration, $[v, b]$ is updated according to the snapshot of that file for all blocks included in the file and for both the source and target volumes.

The Greedy algorithm works as follows: first, the input file systems are parsed into the reference count matrix representing the initial state of the system. After creating the initial state of the system, the algorithm alternates between load balancing phases and optimization phases. We have implemented greedy in C++ and are providing the source code online in [KK]. The greedy algorithm uses few resources, requiring only a single thread to run and maintaining

a simple data structure in memory.

6.2.2 ILP

As previously explained in Chapter 5, our ILP based method parses the input system into an ILP model and then uses an ILP solver to solve it. The ILP algorithm then takes the solver's solution and parses it into a migration plan. In our implementation we used the commercial Gurobi optimizer [Gur] as the algorithm's ILP solver. We used Gurobi's *C++* interface for the creation of the ILP model and the translation of the solution to migration plans. Many of the model's constraints require block intersections between volumes. A data structure called *intersects_source_target_blocks* is created for the intersections to improve model creation. In *intersects_source_target_blocks* cell $[v_1, v_2]$ contains an array of all the IDs of all the blocks that volume v_1 and v_2 share in the initial stage. In our evaluations we show the average results of three random seeds. These seeds effect the order in which the Gurobi solver checks the possible solutions to the model, resulting in different solutions in cases of timeout. The ILP program consists of 1860 lines of code. The code includes the ingestion of the system into a model and the translation of the solver's solution to a migration plan. The code is available online [KK]. ILP requires a lot of resources and scales well with a large number of threads. We used 38 threads in our evaluation. The ILP model is extremely memory intensive and each addition to the amount of blocks or files increases the model size and the memory requirements of the algorithm to create the model.

6.3 Basic comparison between algorithms

6.3.1 Deletion

A migration plan's *deletion* corresponds to the deleted physical size's percentage of the initial system's physical size. Figure 6.1 show the deletions in our experiments. In cases of timeout in ILP the optimal solution may not have been found. This is indicated in the graph with a red x next to the ILP column. The graph has been split into two groups: systems starting with a high load balancing score: Linux and Homes-week systems, and systems with a low load balancing score: Homes-users and UBC-500 systems. A higher load balancing score in the initial state of the system results in better deletion.

As expected, Greedy produces the smallest deletions. Greedy even increases the system's size by replicating blocks to ensure load balancing for Home-users. There is an interesting case in UBC-500 with $T_{max} = 100\%$, where Greedy achieves better deletion than both ILP and Cluster. This occurs because UBC-500 contains few dependencies between files, negating the advantages of Cluster's heuristics and ILP's search for an optimal solution over Greedy.

Despite ILP providing the theoretically optimal solution, there are cases where the clustering algorithm outperforms ILP in terms of deletion. In cases where ILP reached the timeout, like in Linux-all, this can be explained by ILP returning the best solution before the timeout, which is not yet the optimal solution, as ILP had terminated (had reached timeout) before finding it. However in Homes-week and Linux-skip this is because ILP's optimal solution to the sampled system translates worse to the original system than Cluster's heuristically based solution.

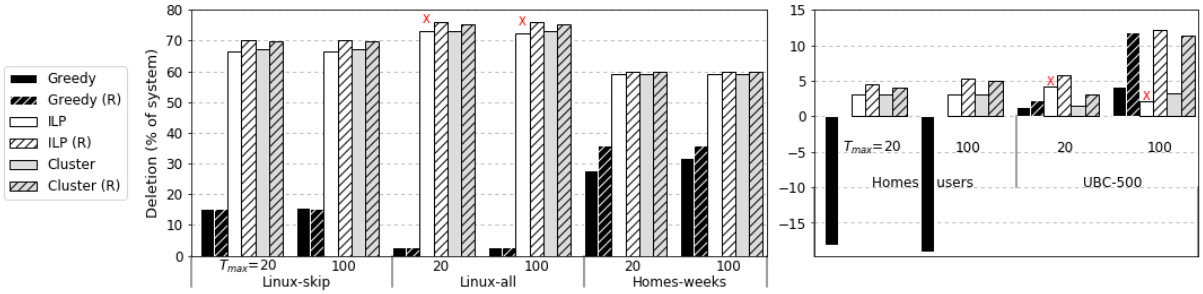


Figure 6.1: Reduction in system size of all systems and all algorithms (with and without load balancing constraints. $k = 13$ and $\mu = 2\%$).

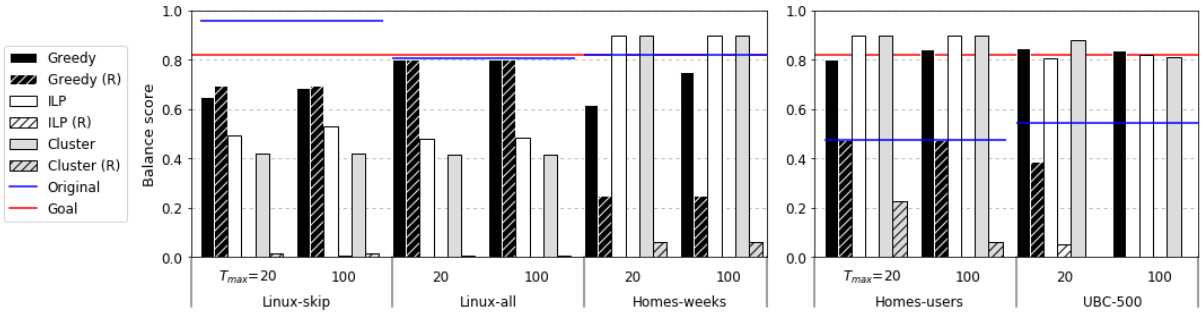


Figure 6.2: Resulting balance of all systems and all algorithms (with and without load balancing constraints. $k = 13$ and $\mu = 2\%$).

suffering more from over fitting to the sampled system than Cluster. ILP seeks the optimal solution, whereas Cluster uses the sampled system heuristically. This solution upon translation to a migration plan in the original system is outperformed by Cluster's less over fitted solution.

In UBC-500 with $T_{max} = 20\%$, ILP shows a clear advantage over the other algorithms. This case is both the largest in terms of data and contains the least amount of migrations that maintain the constraints. The limited options are better utilized in the ILP algorithm which searches for the optimal solution rather than searching heuristically. In the case of UBC-500 with $T_{max} = 100\%$ the effect of the timeout becomes even more apparent. The high traffic allocation allows for significantly more solutions to examine, resulting in a migration plan that is further than the optimal one when timeout is reached.

The relaxed version of the algorithms consistently shows better deletions. This is due to not having to trade deletion for load balancing, two aspects of migration that are at odds with each other. The most extreme example of this is seen in Homes-users, where the Greedy algorithm had to increase the size of the system to maintain the load balance. In the relaxed algorithm where no load balancing constraint existed, the best migration plan was found to be migrating nothing, as any migration would increase the size of the system (thus resulting in 0% deletion).

6.3.2 Load Balance

In figure 6.2 we show the balance achieved by each algorithm. With a tolerance margin of $\mu = 2\%$ and five volumes, the balance should be at least $^{18}/_{22} = 0.82$ yet this value is not necessarily reached. In the case of the Greedy algorithm, greedy might exhaust its maximum allowed traffic, creating an incomplete migration plan that can be unbalanced. In the cases of ILP and Cluster, the migration plans given as output do meet the required balance, however they do so for the sampled system. When applied to the original system the load value is not necessarily

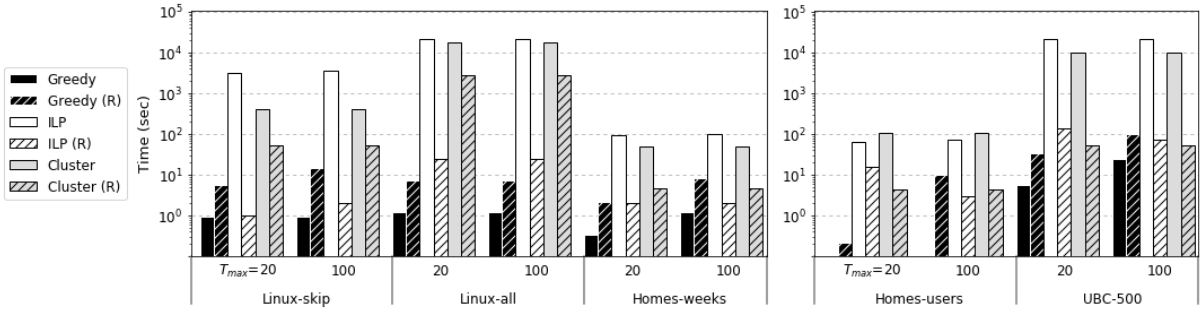


Figure 6.3: Algorithm runtime for all systems and all algorithms (with and without load balancing constraints. $k = 13$ and $\mu = 2\%$).

maintained. This is most prominent in the Linux systems where some files (i.e, entire Linux versions) are represented in the sample by only one or two blocks. Despite limiting the solutions to be within the load balancing tolerance margin, the solution applied to the original system not only deteriorates the load balancing score but violates the required load balancing tolerance margin. Their relaxed versions result in highly unbalanced systems, sometimes resulting in empty volumes. In the relaxed greedy algorithm, extreme load imbalances are avoided. This is an unintentional advantage of the greedy algorithm's inability to plan ahead. Objectively, if no load balance constraint exists, the migration of data to a single volume would result in the most size reduction. The greedy algorithm which works with one file at a time does not necessarily converge to transferring all files it can to a single volume.

6.3.3 Runtime

In Figure 6.3 we show the runtime of each algorithm (note the log scale of the y-axis). Greedy is able to generate a migration plan in 20 seconds or less, in all our experiments. ILP takes longer than an hour and often reaches our six hour timeout, because ILP attempts to solve an NP-hard problem. An exception to this are the Homes systems which contain the least amount of files, decreasing the size of the ILP model considerably. Cluster usually has shorter runtimes than ILP, with the exception of Home-users, yet still has relatively long runtimes. However, Cluster's runtime can be reduced by decreasing the number of executions in the clustering process (180 in our evaluations).

The relaxed version of Greedy has longer runtimes than its original version. This is due to two reasons: 1) fewer limitations allow for more iterations before no legal migration remains; 2) without a load balancing constraint all phases become optimization phases, in these phases Greedy checks all legal migrations, resulting in longer average iteration runtimes. In the case of ILP and Cluster the relaxed versions terminate faster by one or two orders of magnitude. In ILP this is due to reducing the model's complexity. In Cluster the removal of the load balance constraint allows it to terminate with a single attempt rather than attempting multiple times to meet the load balance constraint.

6.3.4 Implications

Our basic comparison leads to several notable observations

(1) ILP has a clear advantage over Greedy. This was not the case in previous studies

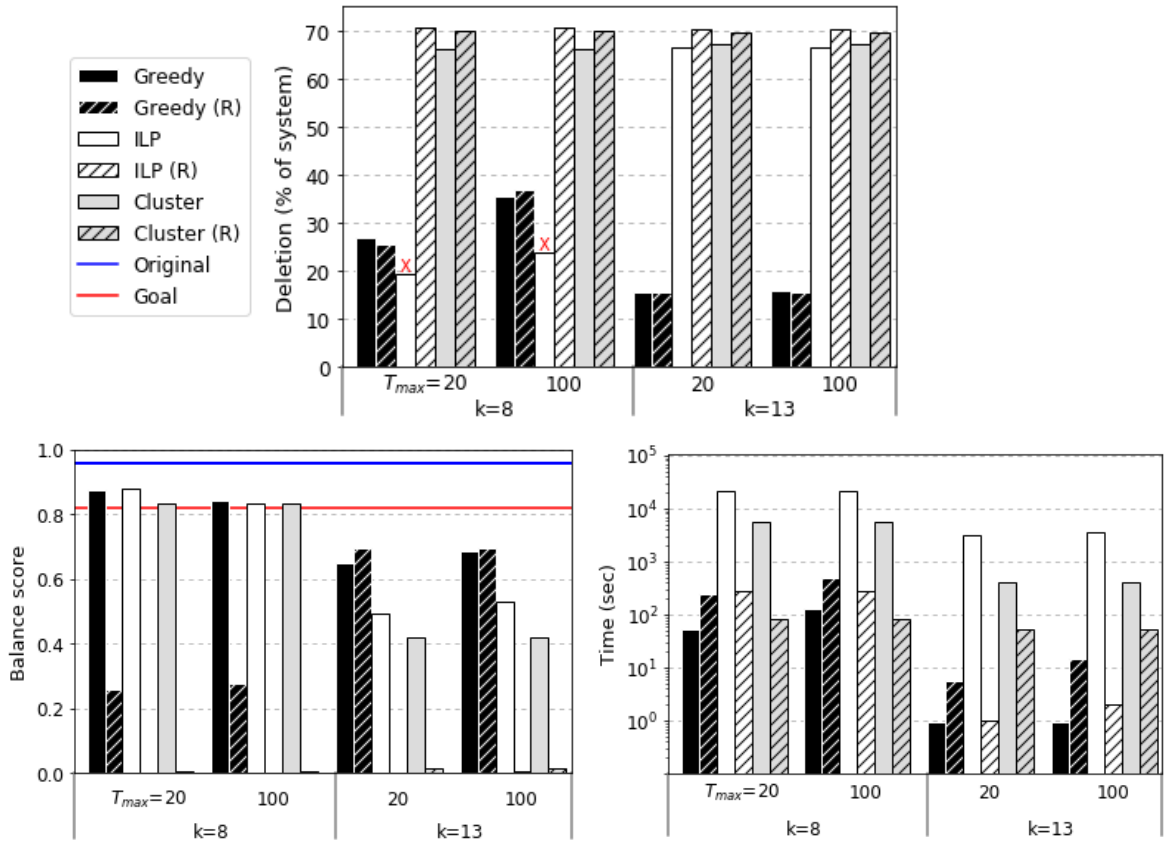


Figure 6.4: Linux-skip system with 5 volumes, $\mu = 2\%$, and two sampling degrees: $k = 8, 13$.

that examined simple cases of migration, i.e., seeding [NSKY21] and space reclamation [NK13]. When we examine the general migration plan, it seems the increased complexity of the problem increases the gap between the greedy solutions and the optimal solutions.

(2) Despite the premise of optimality in the ILP-based approach, it fails to yield better results than Cluster. Cluster has comparable results and may even outperform ILP. We conclude that in most cases Cluster is the more efficient approach. However, in cases where a migration plan must be created with highly restrictive constraints ILP finds significantly better solutions in terms of size reduction.

(3) While the load balancing constraint conflicts with the size reduction objective, its effect on the size reduction is usually small. The effect the load balancing constraint has on size reduction depends on the file similarity degree and load balance of the initial system.

6.4 Sensitivity to problem parameters

6.4.1 Sampling degree

In figure 6.4 we show the deletion, load balance, and runtime of all algorithms with two samples of the Linux-skip system. We used the sampling degree of $k = 13$ and $k = 8$ to generate the small and big samples, respectively. The Greedy algorithm benefits the most from having larger samples, achieving better deletion since larger sampling sizes allow it to detect more size reduction opportunities. This has the opposite effect on ILP-increasing the sample size reduces the size reduction; this is because increasing the model size results in less solution being checked before a timeout is reached.

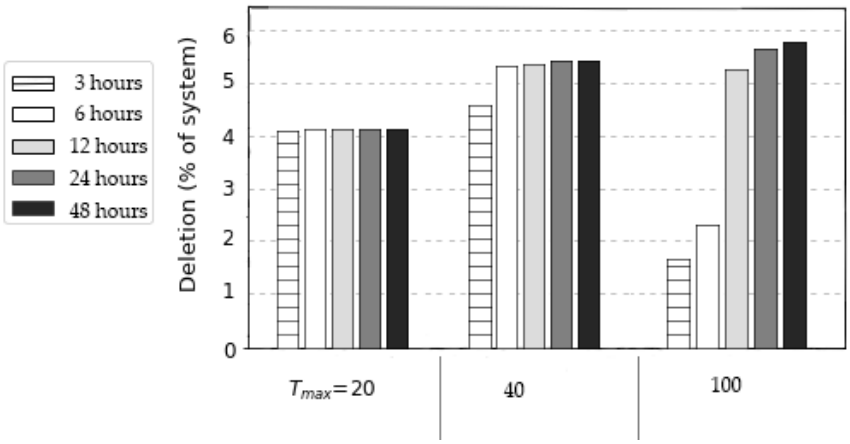


Figure 6.5: Deletion in UBC-500 system, $\mu = 2\%$, increasing timeout values

We repeated the ILP execution on the large sample with an increased timeout of twelve hours - figure 6.5. This resulted in a minor improvement of the output quality in cases where constraints were limiting $T_{max} = 20$ and in a major improvement in cases where the constraints were lenient $T_{max} = 100$. This is due to lenient constraint allowing for more solutions which the ILP solver needs to go over thus increasing the impact of the timeout. Cluster returns similar results for both sample sizes.

All the algorithms achieve better load balancing when run on the larger sample size (k=8), because the load-balancing constraint is enforced on more blocks, and thus more accurately translates to the original system. However increasing the sample size increases the runtime by several orders of magnitude. Greedy's runtime scales better than ILP and Cluster when the sample size is larger.

6.4.2 Time limit

To analyze the effect of the timeout value on ILP, we generated a migration plan for the UBC-500 system with $\mu = 2\%$ and different values of T_{max}, repeating the experiment with increasing timeout values, between 3 and 48 hours. The results, presented in e 6.5, show that the effect of the timeout depends on the space of feasible solutions. In this example, increasing T_{max} increases the number of solutions that meet the traffic constraint, respectively increasing the number of solutions that the ILP solver must consider when searching for the optimal solution. With $T_{max} = 20$, approximately eight hours are required to find the optimal solution, and increasing the timeout beyond this time had no effect. The solutions found within three and six hours were already very close to the optimal one. With $T_{max} = 40$, increasing the timeout beyond six hours carried diminishing returns, indicating that the solution is likely very close to the optimal one. In contrast, with $T_{max} = 100$, the solution keeps improving even after 48 hours, due to the very large solution space. These results are consistent with the analysis of GoSeed in [NSKY21] which showed that the majority of the solver's progress is typically achieved in the first half of its overall runtime. As we increased the problem's complexity (by increasing T_{max} we increased the time required for the solver to complete its execution, thus increasing its benefit from longer timeouts.

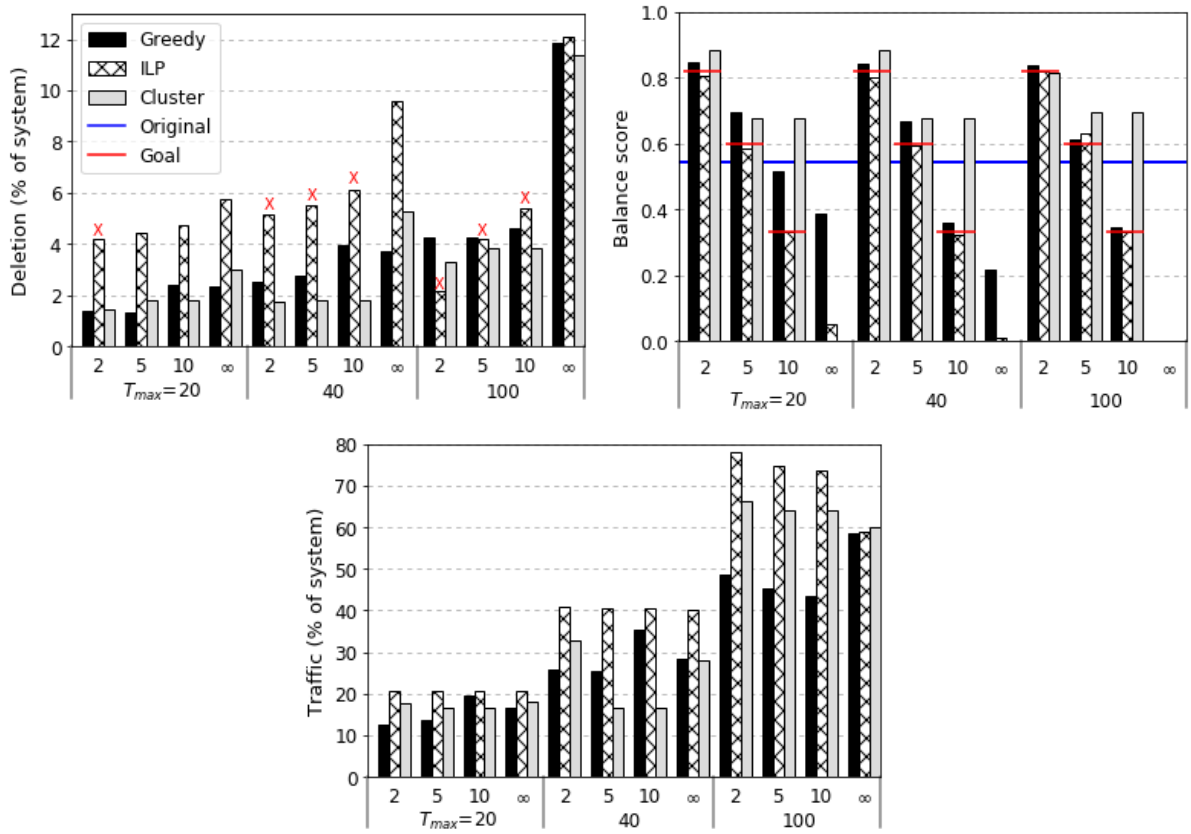


Figure 6.6: UBC-500 system with $k = 13$ and different load balancing margins.

6.4.3 Load balancing and traffic constraints

In figure 6.6 we show the deletion, load balance, and traffic consumption of all three algorithms on the UBC-500 system with different values of T_{max} and μ. UBC-500 shows the highest sensitivity to changes in the constraints due to the low similarities between its files. Increasing T_{max} resulted in improved deletion across all three algorithms and resulted in more traffic consumption. Allowing more traffic allows for more migration to occur. While a lower load balancing tolerance μ does achieve better load balancing, it has a diminishing effect on the size reduction, albeit only by a small amount.

6.4.4 Number of volumes

In figure 6.7 we show the deletion and runtime of all three algorithms on the Linux-skip. The columns named 4, 5, 6, 10 refer to four different distributions of Linux-skip's files into columns. The system presented under the name four is a system with an initial amount of 5 volumes, which is required to migrate to a final state with 4 volumes. The system presented under the name five is the default system in our evaluations where there are 5 initial volumes and 5 final volumes. The system presented under the name six is a case where an additional empty volume is provided to the system for the migration. The system presented under the name ten is a case where the initial system consists of 10 volumes instead of our default 5, and 10 final volumes. Similar deletions are achieved when one volume is added or removed, due to the high similarity between Linux versions. When the initial number of volumes is 10 more duplicates exist in the initial stage which allows for more deletion opportunities.

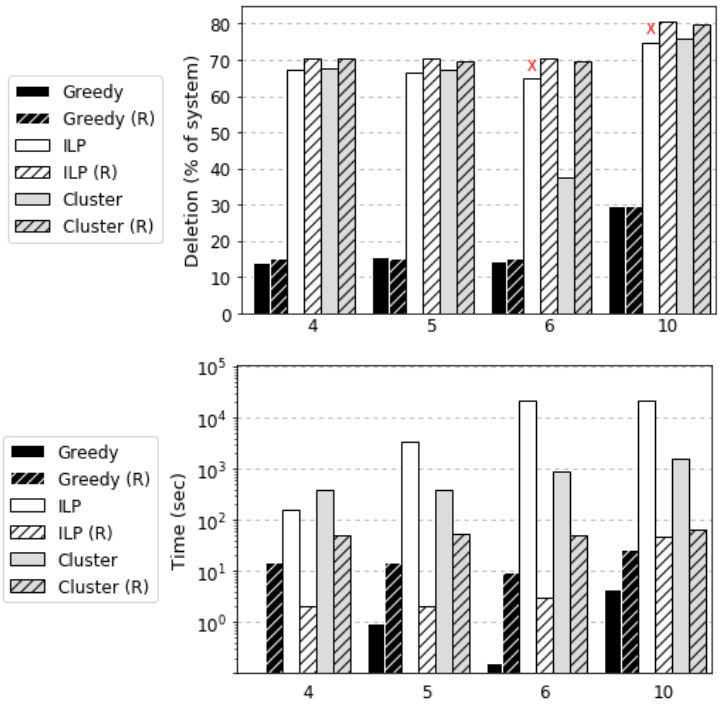


Figure 6.7: Linux-skip with different numbers of target volumes with $T_{max} = 100, k = 13, \mu = 2\%$.

We can see an increase in runtime when there is an increase in the number of volumes. As before, the Greedy algorithm scales better in terms of runtime when we increase the problem complexity. In the case of adding or removing a volume Greedy spends more time on the faster load-balancing step and requires less time. In all cases the runtime for 10 volumes was longer than for 5 volumes because there are more file migration options to consider. This is most significant in ILP where increasing the volume size increases the model size and the runtime. All algorithms managed to generate migration plans for varying volumes.

6.5 Additional sampling

We evaluated the effectiveness of additional sampling on two systems, UBC-500 and Linux-skip, with four initial sampling degrees (k) and four additional sampling degrees (k). We use $\mu = 20\%$ as in the rest of the experiments. Figure 6.8 shows the results for the UBC-500 system. With $T_{max} = 20$, the solution obtained with $k = 13$ was close to optimal. Nevertheless, the additional sampling relaxed the constraints to allow more efficient solutions. Note that increasing the initial sampling to $k = 14, 15$ had a similar effect.

With $T_{max} = 100$, where the solution space is initially very large, the effect of additional sampling was different with different initial sampling degrees. With $k = 12$ the solution space became so large that the deletion decreased with k' as a result of the solver timing out farther from the optimal solution. With $k = 13$, the solver found a better solution with an internal sample of $k = 1$ but increasing k' reduced the quality of the solution. With $k = 14, 15$ the initial sample was small enough to prevent these negative effects. In general, increasing the additional sampling degree increased the solver's runtime and reduced the system's balance, when compared to migration plans without additional sampling.

Figure 6.9 shows the results for the Linux-skip system. Recall that this system is much

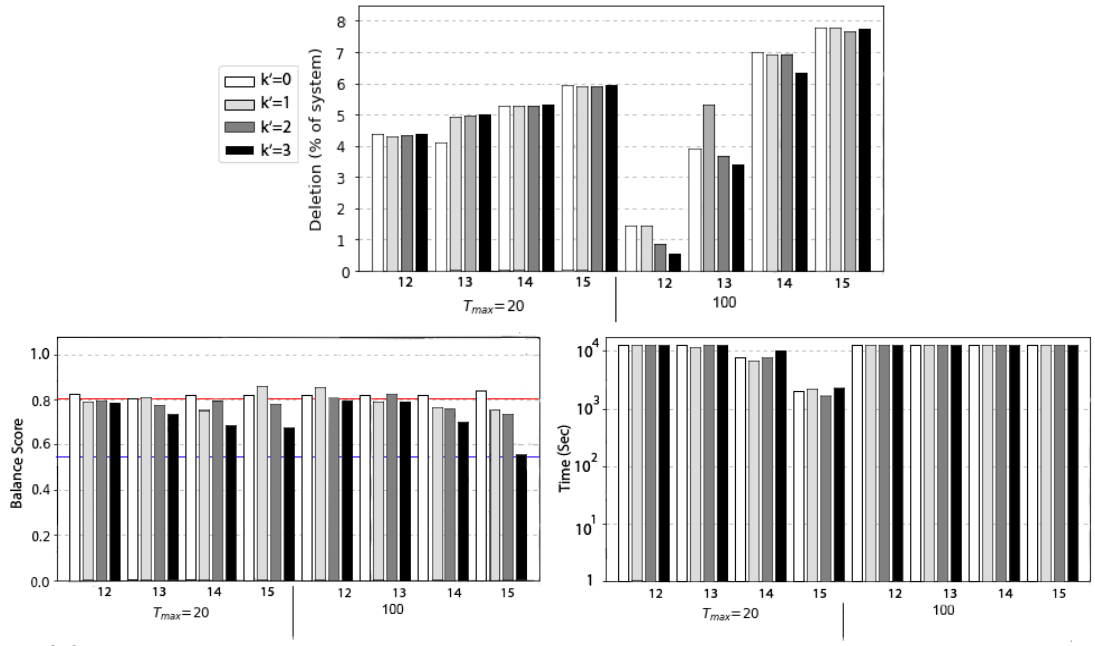


Figure 6.8: Deletion, balance, and runtime of UBC-500 with different additional sampling sizes with different values of $T_{max}, K, and k'$.

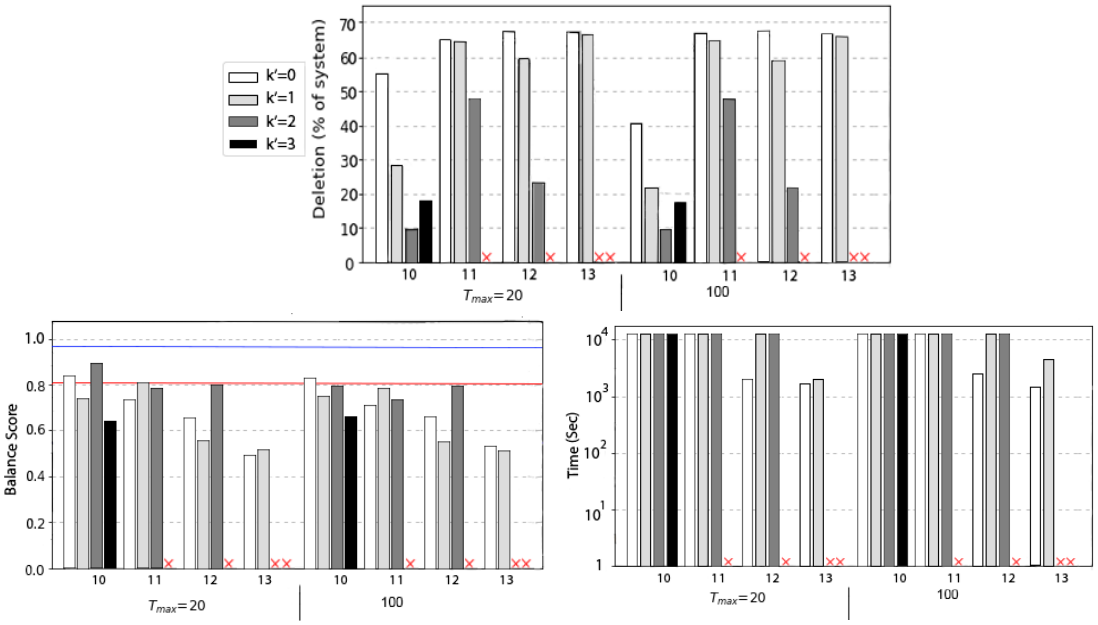


Figure 6.9: Deletion, balance, and runtime of Linux-skip with different additional sampling sizes with different values of $T_{max}, K, and k'$. The red 'x' marks represent experiments where the ILP solver determined that no solution meets the problem's constraints (infeasible).

smaller, with some files represented by as few as one or two blocks in the initial sample with $k = 13$ We thus used smaller initial sampling degrees for this system. Nevertheless, additional sampling resulted in an unfeasible ILP problem (i.e., there is no solution that satisfies its constraints) when the combined sampling degrees were too high: $k = 11, 12$ with $k' = 3$ and $k = 13$ with $k' = 2, 3$. This is the result of some files having a size of zero in the load-balancing constraint. As in the UBC-500 system, increasing the additional sampling degree increased the solver's runtime and reduced the system's balance, except some anomalies due to the aggressive sampling.

We conclude that additional sampling is not an effective acceleration heuristic: it increases the space of feasible solutions without reducing the number of variables in the ILP instance. As a result, it increases the time required to find an optimal solution instead of reducing it. For a large system, it is more effective to increase the initial sampling degree. Doing so reduces the size of the problem (rather than its complexity), resulting in shorter runtimes and better migration plans. However, in systems with small files, care must be taken not to reduce the size of the sampled system excessively, as this might result in an unfeasible ILP instance, or have negative effects on the system's balance after migration.

Chapter 7

Conclusions and open questions

We formulated the general migration problem for storage systems with deduplication, and presented two algorithms for generating an efficient migration plan. Our evaluation showed that the greedy approach is the fastest but least effective, and that despite ILP's premise of optimally, the clustering-based approach has comparable results. Greedy's migration plans can be improved by using larger sample sizes. This is because Greedy both has short run times and scales well in terms of run times. In the case of highly restrictive constraints on traffic and load balancing, ILP results in the most optimal migration plan given enough time. In most cases it is preferable to use the clustering-based approach.

Both our algorithms can be applied to more specific cases of migration, allowing further optimizations in the future. For example, thanks to its short runtime, we can use Greedy to generate multiple plans with different traffic constraints to allow multiple choices between the tradeoffs these constraints have. In another specific case, files need to be migrated together since they are logically dependant, for example all the files of a specific user. This specific case can be solved without modifying the algorithm, simply by representing all the files of a single user as a single file. In a more difficult case where the logically dependant files do not start in the same volume, an addition of a constraint for dependant files can be inserted, forcing the solution to migrate dependant files to the same volume.

Our ILP model created for the problem we presented in this work is highly restrictive in terms of possible migrations. However, relaxations could be applied to the model in order to reduce the size of the model. Future works could explore relaxations to our model and their effects on ILP both in runtime and in possible improvement of the results due to a more heuristic approach, as we've seen is preferable.

Despite the fact that the migration problem presented in this work makes no assumptions regarding the system other than deduplication, there is a more general problem scenario where data is dynamically added or deleted to the system, as opposed to remaining static as it is in our model. The migration problem of dynamic systems is likely to be more difficult to apply to the ILP based approach than to the Greedy approach, since it requires a model for the entire system to be applied.

The different advantages and disadvantages in the three algorithms presented in this work might motivate a hybrid algorithm for migration, one that utilizes the strengths of all three algorithms. The simplest implementation of this would be prepossessing the system to infer

which algorithm would be best suited for it and then using the unaltered algorithm that was chosen.

Bibliography

[AAA+10] 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.

[Aba89] Jeph Abara. Applying integer linear programming to the fleet assignment problem. *Interfaces*, 19(4):20–28, 1989.

[ADK+18] 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.

[AHH+01] 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.

[AHK+02] 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.

[BELL09] 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.

[BLC14] Bharath Balasubramanian, Tian Lan, and Mung Chiang. SAP: Similarity-aware partitioning for efficient cloud storage. In *IEEE Conference on Computer Communications (INFOCOM 14)*, 2014.

[CAVL09] 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.

[clu] Cluster analysis. `https://en.wikipedia.org/wiki/Cluster_analysis`. Accessed: 2020-10-24.

[CLZ11] 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.

[CPL] CPLEX Optimizer. https://www.ibm.com/analytics/cplex-optimizer. Accessed: 2018-10-24.

[DDL+11] 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.

[DDS+17] 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.

[DGH+09] 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.

[DJS+19] 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.

[DSL10] 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.

[ESSY22] Nadav Elias, Philip Shilane, Sarai Sheinvald, and Gala Yadgar. DedupSearch: Two-Phase deduplication aware keyword search. In *20th USENIX Conference on File and Storage Technologies (FAST 22)*, pages 233–246, Santa Clara, CA, February 2022. USENIX Association.

[FFH+14] 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.

[FSL] Traces and snapshots public archive. http://tracer.filesystems.org/. Accessed: 2018-10-24.

[GE11] Fanglu Guo and Petros Efstathopoulos. Building a high-performance deduplication system. In *2011 USENIX Conference on USENIX Annual Technical Conference (USENIX ATC 11)*, 2011.

[GNU] GLPK (GNU Linear Programming Kit). https://www.gnu.org/software/glpk/. Accessed: 2018-10-24.

[GP13] Michael Greenacre and Raul Primicerio. *Hierarchical Cluster Analysis*. Fundación BBVA, Bilbao, 2013.

[Gur] The fastest mathematical programming solver. http://www.gurobi.com/. Accessed: 2018-10-24.

[GWM07] Ron Gabor, Shlomo Weiss, and Avi Mendelson. Fairness enforcement in switch on event multithreading. 4(3):15–es, September 2007.

[HHS+19] 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.

[HSX+12] Cheng Huang, Huseyin Simitci, Yikang Xu, Aaron Ogus, Brad Calder, Parikshit Gopalan, Jin Li, and Sergey Yekhanin. Erasure coding in Windows Azure Storage. In *2012 USENIX Annual Technical Conference (USENIX ATC 12)*, 2012.

[Kar72] Richard M. Karp. *Reducibility among Combinatorial Problems*, pages 85–103. Springer US, Boston, MA, 1972.

[KBKD12] 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.

[KK] Roei Kisous and Ariel Kolikant. Source code of migration algorithms. `https://github.com/roei217/DedupMigration`. Accessed: 2022-02-22.

[KKD+22] Roei Kisous, Ariel Kolikant, Abhinav Duggal, Sarai Sheinvald, and Gala Yadgar. The what, the from, and the to: The migration games in deduplicated systems. In *20th USENIX Conference on File and Storage Technologies (FAST 22)*, Santa Clara, CA, February 2022. USENIX Association.

[LAW02] 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.

[LEB+09] 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.

[LEB13] 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.

[Lin] Linux Kernel Archives. `https://mirrors.edge.kernel.org/pub/linux/kernel/`.

[LLD+14] 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.

[lps] Introduction to lp_solve 5.5.2.5. http://lpsolve.sourceforge.net/5.5/. Accessed: 2018-10-24.

[LSD+14] 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.

[Man94] Udi Manber. Finding similar files in a large file system. In *USENIX Winter 1994 Technical Conference (WTEC 94)*, 1994.

[MB11] Dutch T. Meyer and William J. Bolosky. A study of practical deduplication. In *9th USENIX Conference on File and Stroage Technologies (FAST 11)*, 2011.

[MCM01] Athicha Muthitacharoen, Benjie Chen, and David Mazières. A low-bandwidth network file system. In *18th ACM Symposium on Operating Systems Principles (SOSP 01)*, 2001.

[MHS18] Keiichi Matsuzawa, Mitsuo Hayasaka, and Takahiro Shinagawa. The quick migration of file servers. In *11th ACM International Systems and Storage Conference (SYSTOR 18)*, 2018.

[NEF+12] Edmund B. Nightingale, Jeremy Elson, Jinliang Fan, Owen Hofmann, Jon Howell, and Yutaka Suzue. Flat datacenter storage. In *10th USENIX Symposium on Operating Systems Design and Implementation (OSDI 12)*, 2012.

[NK13] P. C. Nagesh and Atish Kathpal. Rangoli: Space management in deduplication environments. In *6th International Systems and Storage Conference (SYSTOR 13)*, 2013.

[NSKY21] Aviv Nachman, Sarai Sheinvald, Ariel Kolikant, and Gala Yadgar. GoSeed: Optimal seeding plan for deduplicated storage. *ACM Trans. Storage*, 17(3), August 2021.

[RH02] Alexander Richards and Jonathon P. How. Aircraft trajectory planning with collision avoidance using mixed integer linear programming. *Proceedings of the 2002 American Control Conference (IEEE Cat. No.CH37301)*, 3:1936–1941 vol.3, 2002.

[RSG+13] K. V. Rashmi, Nihar B. Shah, Dikang Gu, Hairong Kuang, Dhruba Borthakur, and Kannan Ramchandran. A solution to the network challenges of data recovery in erasure-coded distributed storage systems: A study on the Facebook warehouse cluster. In *5th USENIX Workshop on Hot Topics in Storage and File Systems (HotStorage 13)*, 2013.

[SBGV12] 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.

[SCJ16] 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.

[SNI] SNIA IOTTA Repository. `http://iotta.snia.org/tracetypes/6`. Accessed: 2018-10-24.

[SYM] SYMPHONY development home page. https://projects.coin-or.org/SYMPHONY. Accessed: 2018-10-24.

[TAB11] 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.

[WBMM06] Sage A. Weil, Scott A. Brandt, Ethan L. Miller, and Carlos Maltzahn. CRUSH: Controlled, scalable, decentralized placement of replicated data. In *ACM/IEEE Conference on Supercomputing (SC 06)*, 2006.

[XJF+14] 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.

[XZJ+16] 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.

[YJTL16] 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.

[zCWWD18] 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.

[ZLP08] 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.

[ZSW16] Yanhua Zhang, Xingming Sun, and Baowei Wang. Efficient algorithm for k-barrier coverage based on integer linear programming. *China Communications*, 13(7):16–23, 2016.

[ZWM13] Charlie Shucheng Zhu, Georg Weissenbacher, and Sharad Malik. Coverage-based trace signal selection for fault localisation in post-silicon validation. In *Hardware and Software*, Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics), pages 132–147, Germany, 2013. Springer Verlag. Copyright: Copyright 2021 Elsevier B.V., All rights reserved.; 8th International on Hardware and Software: Verification and Testing, HVC 2012 ; Conference date: 06-11-2012 Through 08-11-2012.

לפעמים הינן תוצאות טובות יותר מבחינת הקטנת גודל האחסון. האלגוריתם מבוסס המקבצים אינו מסיים את ריצתו במספר שניות כמו האלגוריתם החמדן, אך אינו מגיע לזמני הריצה הארוכים של האלגוריתם המבוסס על ILP.

משפיעה באופן ישיר על מרחב הפתרונות, מה שמאפשר למנהל מערכת לתעדף בין השיקולים. בעיית המגרציה אם כך היא למצוא איזה קבצים להעביר מאיזה חלק מקור ולאיזה חלק יעד, בכדי לעמוד באילוצי איזון עומסים ורוחב פס וכדי להקטין כמה שיותר את גודל האחסון של המערכת המשמש לאחסון נתונים.

אנחנו מציגים בעבודה זו שני אלגוריתמים חדשים ליצירת תוכנית מגרצייה יעילה. הראשון הוא אלגוריתם חמדן הבנוי על בסיס האלגוריתם החמדן האיטרטיבי שהוצג בעבודה קודמת בנושא מגרציה תחת דדופליקציה. האלגוריתם החמדן המורחב שלנו מחלק את הנתונים באופן שווה בין חלקי המערכת בזמן שהוא מבטיח שרוחב הפס המשומש בההעברות הללו אינו עולה על רוחב הפס המקסימלי המורשה להן. על ידי פיצול התהליך הזה לשני שלבים: שלב איזון העומסים ושלב מיטוב האחסון, אנחנו מבטיחים שרוחב הפס המוקצה משמש הן לאיזון עומסים והן להפחתת גודל האחסון המשמש לאחסון נתונים.

האלגוריתם השני שלנו נבנה על בסיס האלגוריתם המבוסס על ILP בGoSeed, אלגוריתם אשר פותר את בעיית הזריעה. אנחנו מנסחים כאן מחדש את בעיית הILP בGoSeed עם משתנים ואילוצים המבטאים את רוחב בפס המשמש בזמן ההעברות ואת בחירת החלקים מהם ואליהם קבצים יועברו. הניסוח שלנו לבעיית מגרציית הנתונים הכללית הוא באופן טבעי יותר מורכב מהניסוח אשר היה נדרש לבעיית הזריעה אשר הינה מקרה פרטי מאוד מוגבל של בעיית מגרציית הנתונים. למרות מורכבות זו, יישמנו בהצלחה את הניסוח שלנו על בעיית מגרציית נתונים של מערכות בעלות מאות מיליוני גושי נתונים ייחודיים.

עבודה זו נעשתה במקביל לעבודה דומה לפתרון בעיית מגרציית הנתונים למערכות תחת דדופליקציה בעזרת אלגוריתם מבוסס מקבצים: Cluster. האלגוריתם החמדן ואלגוריתם ILP המוצגים בעבודה זו הושוו בעיקר לאלגוריתם מבוסס המקבצים, זאת מאחר וכל האלגוריתמים מבצעים את אותן הנחות על הבעיה.

אנחנו יישמנו והשוונו את שני האלגוריתמים שלנו על שש מערכות שנוצרו משלושה מערכי נתונים: 1) UBC 2) FSL 3) גרסאות מערכות הפעלה של Linux. התוצאות שלנו מראות שכל האלגוריתמים מסוגלים להפחית באופן מוצלח את גודל המערכות בזמן שהם שומרים על אילוצי רוחב הפס ואיזון העומסים. ההשוואה בין האלגוריתמים מראה שלכל אחד משלושת האלגוריתמים ישנם ייתרונות וחסרונות שונים.

האלגוריתם החמדן מסוגל ליצור במספר שניות תוכניות מגרציה להעברת הקבצים. זמני ריצה אלו הם זמני ריצה משמעותית נמוכים יותר מזמני הריצה שך האלגוריתמים האחרים אשר דורשים לפחות מספר שעות ריצה ויכולים אף להגיע למספר ימים של ריצה. זמני הריצה של האלגוריתם החמדן מושפעים הכי פחות מהגדלת המערכת בזמן שזמני הריצה של האלגוריתמים האחרים גדלים בסדרי גודל עם הגדלת גודל המערכת. החסרון הבולט של האלגוריתם החמדן הוא שהפחתת גודל המערכת המושגת מתוכניות המיגרציה של האלגוריתם החמדן היא בדרך כלל הקטנה ביותר מבין על האלגוריתמים.

האלגוריתם המבוסס על ILP יכול לנצל באופן היעיל ביותר את רוחב הפס. האלגוריתם המבוסס על ILP מסוגל לפתור בצורה טובה מקרים בהם ישנם אילוצים מאוד מגבילים על המערכת, בזמן שהאלגורתמים האחרים מציעים פתרונות משמעותית פחות טובים במקרים אלו. הקלה של מרחב הבעיה לבעיה בה לא נדרש איזון עומסים מביאה לשיפור המשמעותי ביותר אצל ILP. החסרון של האלגוריתם המבוסס על ILP הוא זמני ריצה משמעותית יותר גדולים מהאלגוריתמים האחרים. זמני הריצה של האלגוריתם המבוסס על ILP הם כל כך גבוהים שהם מצריכים שימוש במגבלות זמן לריצת האלגוריתם. מגבלות זמן אלו, במידה והאלגוריתם מגיע אליהן, מכריחות את האלגוריתם להחזיר את תוכנית המיגרציה הטובה ביותר שמצא עד אותה נקודה בה נעצר ולא את תוכנית המיגרציה האופטימלית אותה יכול היה להשיג ללא מגבלת זמן.

האלגוריתם מבוסס המקבצים מצליח להשיג איזון בין היתרונות והחסרונות של האלגוריתם המבוסס על ILP והאלגוריתם החמדן. התוצאות של האלגוריתם מבוסס המקבצים משתוות לאלו המושגות על ידי ILP ואף

תקציר

מערכות אחסון רחבות היקף רבות משתמשות בדדופליקציה כדי להקטין את גודל הנתונים שהן מאחסנות. תהליך הדדופליקציה הוא תהליך אשר מזהה בלוקים כפולים של נתונים בקבצים השונים ומחליף אותם במצביעים להעתק ייחודי של הבלוק השמור במערכת. שימוש בדדופליקציה לצורך הפחתה בגודל המערכת גורר עליה במורכבות המערכת. בזמן שמורכבות הקריאה, כתיבה ומחיקה של נתונים במערכות אחסון המשתמשות בדדופליקציה טופלה בהרבה מאוד מחקרים אקדמאים ומערכות מסחריות, התחום הניהולי של מערכות אחסון רחבות היקף, כמו תכנון קיבולת, מטמונים, איכות ועלות שירות, עדיין דורש התייחסות.

עבודה זו מתמקדת במגרציית נתונים: העברה של קבצים בין חלקים שונים של מערכת אחסון המשתמשת בדדופליקציה. "חלק" יכול לייצג שרת יחיד בתוך מערכת אחסון רחבת היקף או מספר שרתים עצמאים המיועדים ללקוח או למערך נתונים. קבצים יכולים לעבור כתוצאה מהגעתם של חלקים למגבלת קיבולת או להיווצרותו של צוואר בקבוק אחר במערכת האחסון.

דדופליקציה מציגה שיקולים חדשים בבחירת הקבצים למגרצייה אשר אינם קיימים במערכות אחסון קלאסיות. במערכות אחסון תחת דדופליקציה, בשל התלויות בין הקבצים, כאשר קובץ עובר מקום, חלק מהבלוקים יכולים להימחק ממיקומו המקורי בזמן שבלוקים אחרים עלולים עדיין להיות שייכים לקובץ אחר באותו מיקום. באופן דומה, חלק מהבלוקים דורשים העברה למיקום היעד של המגרצייה בזמן שבלוקים אחרים עשויים כבר להיות שמורים שם לפני ההעברה כך שלא רק שלא יצריכו העתקה, גם ייתכן שניתן למחוק את העותק השמור במיקום המקורי. תוכנית מגרצייה יעילה נדרשת לבצע אופטימזציה בין מספר שיקולים סותרים. השיקול הראשון אליו התייחסו רבות המחקרים קודמים הוא השיקול של הגודל הפיזי של הנתונים המאוחסנים אותו יש להפחית. השיקול השני הוא השיקול של איזון העומסים בין חלקי המערכת השונים. השיקול השלישי הוא השיקול של רוחב הפס שיעשה בו שימוש בזמן ההעברות עצמן, קיים גודל מסויים של רוחב פס שיתאפשר להשתמש בו לצורך ביצוע תכנית המיגרציה.

בזמן שאכן קיימים מספר מחקרים המתייחסים לבעיית מגרציית הנתונים במערכות תחת דדפוליקציה, הפתרונות הקיימים או פותרים מקרים מאוד מוגבלים של הבעיה, כמו בעיית הזריעה, או פותרים את הבעיה הכללית ללא התייחסות לכלל השיקולים ומתמקדים במקום זאת רק בהקטנת הגודל הפיזי של הנתונים המאוחסנים.

בעיית הזריעה הינה מקרה מוגבל של בעיית מגרציית הנתונים הכללית. בבעיית הזריעה קיימים שני חלקים במערכת כאשר חלק אחד מהווה מקור וחלק שני מהווה יעד. היעד מתחיל ללא נתונים וקבצים יכולים לעבור רק מהמקור ליעד ולא בכיוון השני.

בעבודה זו אנו מתייחסים בפעם הראשונה הן למקרה הכללי של בעיית מגרציית הנתונים במערכות תחת דדופליקציה והן לשיקולי איזון עומסים ורוחב פס. אנחנו מתחילים בניסוח פורמלי של בעיית המגרצייה באופן הכללי ביותר שלה, כבעיית מגרציה שמטרתה העיקרית היא הקטנת גודל האחסון של המערכת. אנו מוסיפים את שיקולי רוחב הפס ואיזון העומסים כאילוצים לתוכניות המגרצייה. הנוקשות שבהן המגבלות הללו נאכפות

המחקר בוצע בהנחייתה של דוקטור גלה ידגר, בפקולטה למדעי המחשב ע"ש הנרי ומרילין טאוב.

חלק מן התוצאות בחיבור זה פורסמו כמאמרים מאת המחבר ושותפיו למחקר בכנסים ובכתבי-עת במהלך תקופת המחקר של המחבר, אשר גרסאותיהם העדכניות ביותר הינן:

Roei Kisous, Ariel Kolikant, Abhinav Duggal, Sarai Sheinvald, and Gala Yadgar. The what, the from, and the to: The migration games in deduplicated systems. To appear in the Special Section on FAST 22 in the Transaction on Storage.

Roei Kisous, Ariel Kolikant, Abhinav Duggal, Sarai Sheinvald, and Gala Yadgar. The what, the from, and the to: The migration games in deduplicated systems. In *20th USENIX Conference on File and Storage Technologies (FAST 22)*, Santa Clara, CA, February 2022. USENIX Association.

אני מודה לטכניון על התמיכה הכספית הנדיבה בהשתלמותי. מחקר זה נתמך על ידי הקרן הלאומית למדע (מענק מס. 807/20) .

איזון עומסים מבוסס ILP במערכות אחסון עם דדופליקציה

חיבור על מחקר

לשם מילוי חלקי של הדרישות לקבלת התואר
מגיסטר למדעים במדעי המחשב

אריאל קוליקנט

הוגש לסנט הטכניון — מכון טכנולוגי לישראל
תמוז התשפ"ב חיפה ספטמבר 2022

איזון עומסים מבוסס ILP במערכות אחסון עם דדופליקציה

אריאל קוליקנט